1 Robot Localization with the Viterbi Algorithm

Consider the following semi-realistic model of a robot's motion: we say that the state variable X_t represents the robot's location in a grid-like environment. Suppose the grid has r rows and c columns. Then, there are $N = r \cdot c$ total states and the state space can be written as $\mathcal{S} = \{(i, j) \mid 0 \leq i < r,\ 0 \leq j < c\}$, i.e., each $X_t \in \mathcal{S}$ is represented by a tuple of row and column indices.

This environment is enclosed by walls and has both horizontal and vertical walls scattered throughout between pairs of adjacent cells. Let NEIGHBORS(q) denote the set of empty grid cells that are adjacent to state q and are reachable from it (i.e., not blocked from q by a wall). Then, we model the robot's motion in this environment as a random walk where it transitions to one of its unblocked neighboring cells uniformly at random, i.e.,

$$P(X_{t+1} = q' \mid X_t = q) = \frac{1}{|\text{NEIGHBORS}(q)|} \text{ if } q' \in \text{NEIGHBORS}(q) \text{ else } 0$$

We don't know where the robot starts, so we will assume that the probability distribution π over the initial states is uniform; that is $P(X_0 = (i, j)) = 1/N$.

The robot has 4 sensors mounted on it, one pointing in each of the 4 cardinal directions. Each sensor returns a single bit output: 1 when a sensor reads a wall directly in front of it and 0 if it reads open space. The observation at a timestep t is a 4-bit sequence returning the sensor values from the left, right, up and down sensors respectively, **in that particular order**. For example, for the environment in the figure above,

- if the robot is present in the top left cell (i.e., $X_t = (0, 0)$), its sensors should read 1011 because there is a wall in the left, up and down directions.
- if the robot is present in the top right cell (i.e., $X_t = (0, 15)$), its sensors should read 0110 because there are walls in the right and up directions.

We note that the number of 0s that a sensor should read in state q is precisely |NEIGHBORS(q)|.

However, the sensors are faulty! Each sensor, independent of the others and the timestep, can fail with probability ϵ and incorrectly flip its output. Therefore, there are 2^4 possible observations that we can read off of the robot, each occurring with different probabilities: if there are d discrepancies between the true sensor reading and the returned sensor reading, the probability of this observation will be $(1 - \epsilon)^{4-d}\epsilon^d$.

The true robot states X_t are hidden from us, but we can observe the faulty sensor readings, say denoted by Y_t at timestep t, every time the robot moves to a new state. As such, we can model the scenario above using a hidden markov model.

(a) Our goal is to predict the true robot hidden states from the observed sensor readings. Given a sequence of T observations $o_1, \ldots, o_T$, **implement the Viterbi algorithm to decode the most likely sequence of hidden states.** You have all the information required to compute the transition probabilities $P(X_{t+1} = q' \mid X_t = q)$ and observation emission probabilities $P(Y_t = o_t \mid X_t = q)$.

The starter code contains two files:

- `env.py`: This file encodes the dynamics of the environment. In particular, it defines the `Env` class which creates the underlying environment, tracks the true robot hidden state and emits sensor readings. You may call any pre-defined methods of the `Env` class but not access any of its instance variables from `hmm.py`.
- `hmm.py`: This is where all of your code should go.

Since your goal is to return the best set of hidden states, **don't forget to also backtrace your way through the viterbi probabilities** to find the most likely sequence. Feel free to reference **this handout on HMMs** for more information about the Viterbi algorithm.

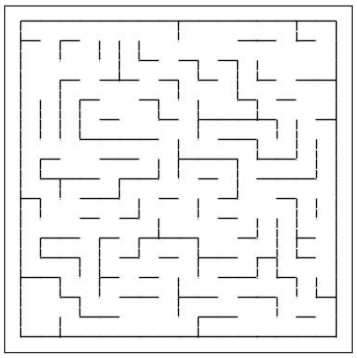
Include your code in the appendix and select the appropriate pages when submitting to Gradescope. Also submit `hmm.py` to the Gradescope coding assignment.

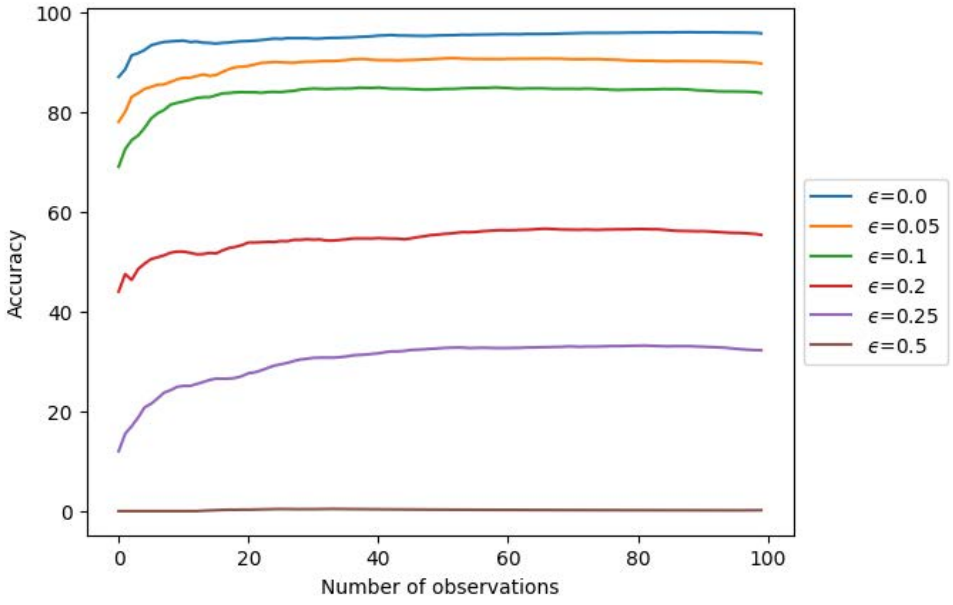
(b) Now, we will vary the error parameter ϵ to values like 0, 0.05, 0.1, 0.2, 0.25 and 0.5. For each value of epsilon, we will generate and decode 100 sequences of 100 observations each. We also plot the cumulative accuracy (i.e., the accuracy of the first i predictions when compared to the first i hidden states, for each $i \in [1, 100]$, for a single trajectory), averaged over all 100 trajectories, for each ϵ. Attach this plot in your writeup. Describe any trends that you see in the plot: what happens when ϵ increases from 0 to 0.5? for a fixed epsilon, how does the cumulative accuracy change over time? Do these trends align with your intuition?

Solution:

As ϵ increases, the cumulative accuracy decreases. This is expected because higher ϵ means the sensors are more likely to provide faulty readings, making it harder for the algorithm to decode the correct hidden states.

For a fixed ϵ, the cumulative accuracy improves as the number of observations increases. This is expected because the Viterbi algorithm can better predict the most likely sequence of states as it gathers more data points and utilizes the inherent structure of the environment and transition probabilities.





2 Graphical Model Potpourri

(a) Show that $a \perp\!\!\!\perp (b, c) \mid d$ implies $a \perp\!\!\!\perp b \mid d$.

Solution:

Start from the assumption $a \perp (b, c) \mid d$, which implies:

$$P(a \mid b, c, d) = P(a \mid d).$$

The marginal probability $P(a \mid b, d)$ can be written as:

$$P(a \mid b, d) = \sum_c P(a \mid b, c, d) P(c \mid b, d).$$

Using the independence assumption $P(a \mid b, c, d) = P(a \mid d)$, we substitute:

$$P(a \mid b, d) = \sum_c P(a \mid d) P(c \mid b, d).$$

Since $P(a \mid d)$ is independent of c, it factors out of the summation:

$$P(a \mid b, d) = P(a \mid d) \sum_c P(c \mid b, d).$$

The term $\sum_c P(c \mid b, d)$ represents the marginalization of $P(c \mid b, d)$ over all possible values of c, which sums to 1. Therefore:

$$P(a \mid b, d) = P(a \mid d).$$

(b) Suppose you have a set of d binary random variables $\{X_1, ..., X_d\}$.

(i) Without making any independence assumptions, what is the minimum number of parameters needed to fully specify the joint distribution?
Hint: You need one fewer parameter compared to the number of outcomes, due to the constraint that the total probability should sum to 1.

Solution: $2^d - 1$

(ii) Suppose the random variables are structured as a Markov chain, where each X_i only depends on X_{i-1}. What is the minimum number of parameters needed now?

Solution:

The joint distribution can be written as:

$$P(X_1, X_2, \ldots, X_d) = P(X_1) \prod_{i=2}^{d} P(X_i \mid X_{i-1}).$$

$P(X_1)$: The initial distribution requires 1 parameter, since $P(X_1 = 1) + P(X_1 = 0) = 1$.

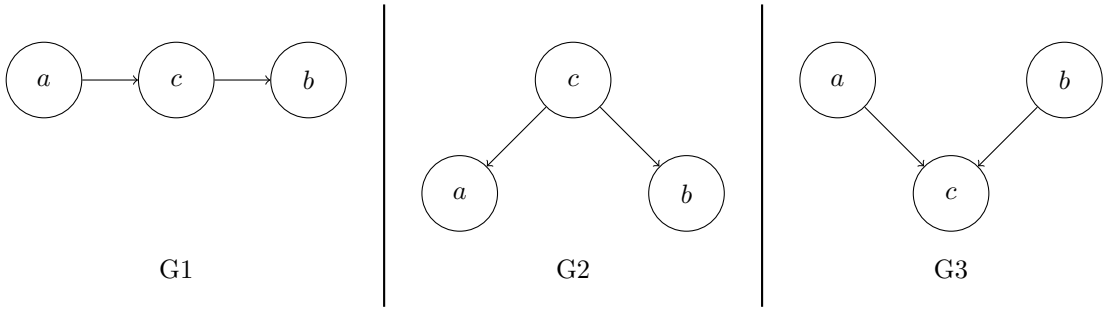
$P(X_i \mid X_{i-1})$: For each i, there are 2 conditional probabilities ($X_{i-1} = 0$ and $X_{i-1} = 1$), but each requires only 1 parameter because the probabilities must sum to 1.

Therefore, Minimum Number of Parameters: 1 + (d-1) = d

(iii) How does the parameter complexity change going from no independence assumption to a Markovian assumption?

Solution: The parameter complexity is significantly reduced from exponential ($2^d - 1$) to linear (d) due to the Markovian assumption. This is because the latter simplifies the dependency structure by only considering local dependencies between consecutive variables.

(c) Consider the following three graphical models with distinct structures.



Match each scenario to the most appropriate graphical model.

(i) A family's decision on where to travel (c) involves considerations such as the parents' work schedules (a) and the children's school holidays (b).

Solution: G3

(ii) A record-breaking snowfall (c) spurs both ski resort bookings (a) and demand for winter clothing (b).

Solution: G2

(iii) A person's workout routine (a) affects their energy levels (c), which then influences their work productivity (b).

Solution: G1

(iv) The climate of a region (a) impacts the type of vegetation that grows (c), which in turn determines the wildlife population (b).

Solution: G1

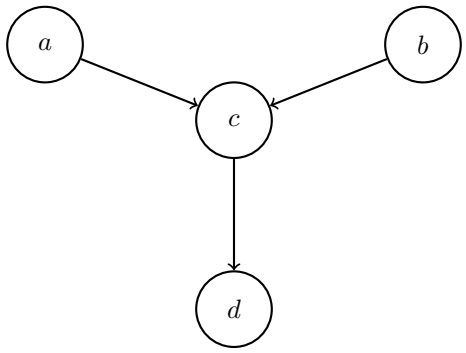
(v) A country's economic stability (c) affects both employment rates (a) and consumer spending habits (b).

Solution: G2

(vi) A restaurant's popularity (c) hinges on both the quality of its food (a) and its social media presence (b).

Solution: G3

(d) Consider the following directed graph, where none of the variables are observed.



(i) What is the joint distribution $p(a, b, c, d)$ in terms of the marginal and conditional distributions given in the graph?

Solution: $P(a, b, c, d) = P(a) \cdot P(b) \cdot P(c \mid a, b) \cdot P(d \mid c)$

(ii) Assuming that a, b, c, d are binary random variables, what is the minimum number of parameters required to specify the joint distribution?

Solution: 1+1+4+2=8 parameters.

(iii) Show that a and b are independent, or $a \perp\!\!\!\perp b$.

Solution: a and b do not share a direct edge, nor do they have a common ancestor or any active path connecting them in the absence of c and d.

(iv) Assume that for any given node x in the graph, $p(x \mid \text{pa}(x)) \neq p(x)$, where $\text{pa}(\cdot)$ denotes the parent nodes of x. Show that when d is observed, a and b are no longer independent, or $a \not\perp\!\!\!\perp b \mid d$.

Solution:

When d is observed, it opens the path $a \rightarrow c \rightarrow d \leftarrow c \leftarrow b$. This is because observing d creates a v-structure at d, making a and b conditionally dependent through d.

The joint distribution now involves a dependency between a and b through c and d:

$$p(a, b \mid d) \neq p(a \mid d) \cdot p(b \mid d).$$

Thus, observing d introduces a dependency between a and b, breaking their marginal independence. Hence, $a \not\perp\!\!\!\perp b \mid d$.

3 Langevin Dynamics Demo

For this question, go to the Google Colab notebook provided ***here*** [1] to complete the code.

In this notebook, we will walk through a simple demo for Langevin dynamics, where the goal is to sample from a distribution p(x) using only its score function ∇_xlogp(x). Here we assume a toy setting where p(x) is known. In most practical cases we only have access a dataset of samples $\mathcal{D} = \{x_0, x_1, \ldots, x_n\} \sim p(x)$, in which case we might use a technique called score matching to estimate the score function[2]. For more background, you can reference Chapter 14.3 on Langevin Sampling in Bishop and Bishop, 2024.

Recall that the Langevin update equation at timestep t with step size η and random noise $\epsilon \sim \mathcal{N}(0, I)$ is

$$x_{t+1} = x_t + \eta \nabla_x \log p(x) + \sqrt{2\eta}\epsilon$$

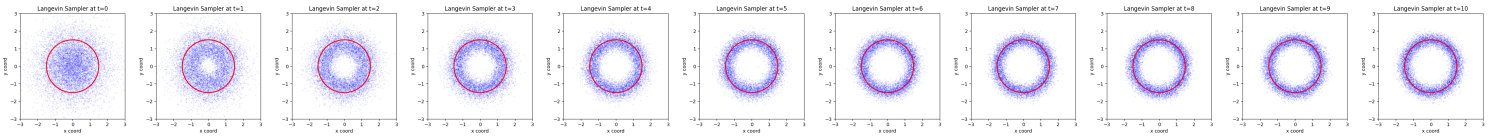
Deliverables: Complete the written questions below, which will walk you through the notebook. In addition, submit the .ipynb file to the code assignment.

1. Complete the functions `langevin_update` and `sample_langevin`. Include a PDF export of the completed code in your write-up.

   Solution: See Appendix

2. Using the given default settings, visualize the samples over time. Include an image of the plots and describe what you see. Which is closer to the target shape, early or late timesteps? How does the variance of the samples change over time?
   Note: To simplify the runtime and plotting, throughout this problem you will only run the Langevin sampler for a few iterations. In practice, however, you would typically run the sampler for longer (e.g., several thousand iterations), to ensure the Markov chain has converged.

   Solution:



At early timesteps (e.g., $t = 0, 1, 2$), the samples are more scattered and do not closely align with the red target shape (ellipse). At later timesteps (e.g., $t = 8, 9, 10$), the samples align more closely with the target shape, showing that the Langevin dynamics sampler effectively moves the particles toward the correct distribution.

At early timesteps (e.g., $t = 0, 1, 2$), the variance is high because the particles are still converging to the target distribution. At later timesteps (e.g., $t = 8, 9, 10$), the variance reduces as the particles approach the target shape. This is expected because the particles settle into regions of high probability density, dictated by the log-pdf function.

[1] https://colab.research.google.com/drive/1fvcWjRJYRLQM83RPKu-uQ4sqAPGTuuxA?usp=sharing
[2] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. NeurIPS 2019.

These behavior aligns with the theoretical expectation of Langevin dynamics. Early timesteps are dominated by noise and gradient-driven movement. As iterations increase, the gradient-driven term dominates, guiding the particles toward the target distribution, and the noise stabilizes the exploration.

3. Now let's adjust the radius of the elliptical logpdf. Set `rx, ry = 1.0, 2.5`. How do the final samples compare with the samples for the circular logpdf? Does Langevin sampling fit better to the elliptical or the circular logpdf?
Hint: Even coverage over the entire target distribution is more desirable.

Solution:

When adjusting the radii of the elliptical log-pdf ($r_x = 1.0, r_y = 2.5$), the final samples adapt to the elongated elliptical shape, compared to the circular log-pdf where samples distribute uniformly around a circular boundary. The elliptical case results in denser sampling along the ellipse's boundary, aligning with the target shape, while the circular log-pdf produces isotropic coverage.

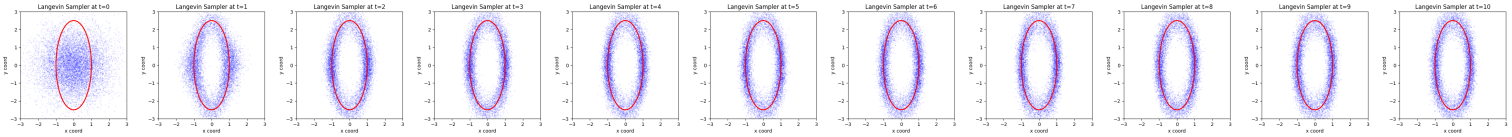
Langevin sampling fits both shapes effectively, but in the elliptical case, the variance reflects the anisotropy of the ellipse, with greater spread along the major axis (r_y). For the circular log-pdf, the variance remains uniform across all directions, matching the symmetry of the target distribution.

Over time, the variance increases initially as particles explore the space, then stabilizes as the sampler converges to the target distribution. The elliptical shape requires more iterations to achieve even coverage along both axes due to its anisotropic nature, while the circular log-pdf converges more uniformly.

In conclusion, Langevin sampling adapts well to both distributions, but elliptical shapes require finer tuning of the step size (η) and more iterations to ensure adequate coverage along all dimensions of the target distribution.

4. Let's see if we can get a better fit to the elliptical logpdf in (c) by tuning each dimension of `eta`. What is a better selection of `eta`?

Solution: eta = torch.tensor([1e-2, 100e-3])



5. Here let's move the center of the logpdf away from the origin. Set `rx, ry = 1.5, 1.5` and `cx, cy = 1.0, 1.0`. How do the final samples compare with centered logpdf? Does Langevin sampling fit better to the centered or off-center logpdf?

Solution:

When the center of the elliptical log-pdf is shifted to $(c_x, c_y) = (1.0, 1.0)$ with radii $r_x = r_y = 1.5$, Langevin sampling adjusts the particle density toward the off-center distribution over successive time steps. The particles converge reasonably well to the elliptical shape, but early steps exhibit lag as the sampling adapts to the translation. Compared to centered distributions, convergence to off-center distributions requires additional iterations due to the need to shift particles from their initialization.

While the variance aligns well with the elliptical shape, under-representation in certain regions suggests that the step size (η) may require tuning. Overall, Langevin sampling adapts effectively to off-center log-pdfs, though centered log-pdfs allow for faster and more uniform convergence.

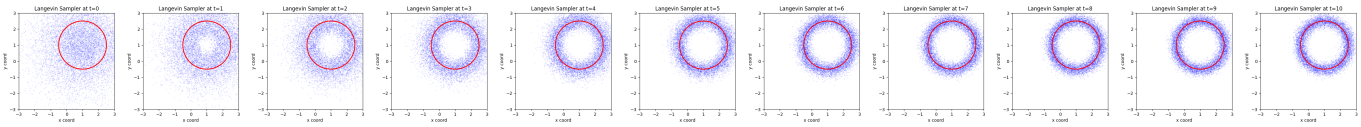
6. For the off-centered logpdf in (e), let's try to get a better fit by tuning the initialization `init_particles`. What can you do to improve the initialization?

Solution:

```
init_particles = torch.randn((langevin_kwargs["num_particles"], langevin_kwargs["num_dims"]))
* torch.tensor([1.5, 1.5]) + torch.tensor([1.0, 1.0])
```

The initialization succeeded because it strategically aligns the particles with the target elliptical distribution. By using `torch.randn` to generate random particles from a standard normal distribution, followed by scaling with the ellipse's radii $[1.5, 1.5]$ and shifting by the center $[1.0, 1.0]$, the particles were distributed close to the high-probability regions of the target.

This approach reduces convergence time as the particles start near the target's optimal regions, minimizing unnecessary drift. Additionally, scaling ensures proper coverage of the ellipse, avoiding sparse or over-concentrated regions, while centering aligns the particles with the target's center. The visual progression confirms this effectiveness, as the particles quickly converge to the elliptical shape with even coverage by the final timesteps.



6 Honor Code

1. **List all collaborators. If you worked alone, then you must explicitly state so.**

   Solution: N/A

2. **Declare and sign the following statement**:

   "I certify that all solutions in this document are entirely my own and that I have not looked at anyone else's solution. I have given credit to all external sources I consulted."

   Signature : _____

   While discussions are encouraged, *everything* in your solution must be your (and only your) creation. Furthermore, all external material (i.e., *anything* outside lectures and assigned readings, including figures and pictures) should be cited properly. We wish to remind you that the consequences of academic misconduct are *particularly severe*!

   Solution: Zhe Wee (Derrick) Ng

4 Appendix

1. **Q1**

```
import matplotlib
matplotlib.use('Agg')  # Use a non-interactive backend

import random
from env import Env
import numpy as np
import matplotlib.pyplot as plt

def observation_probability(env: Env, i: int, j: int, observation: list[int], epsilon: float) -> float
    """
    Helper function to compute the emission probabiility of observing the sensor reading ``observation
    in state (i, j) of the environment.

    Hint: You may want to call env.get_neighbors(i, j) to get the neighbors of (i, j).
    """
    # TODO: compute the emission probability
    # Get the true sensor reading for the state
    node = env._nodes[(i, j)]
    sensorReading = node._true_sensor_reading()

    # Compute probability for each sensor
    prob = 1.0
    for t, o in zip(sensorReading, observation):
        if t == o:
            prob *= (1 - epsilon)
        else:
            prob *= epsilon

    return prob

def viterbi(observations: list[list[int]], epsilon: float) -> np.ndarray:
    """
    Params:
    observations: a list of observations of size (T, 4) where T is the number of observations and
    1. observations[t][0] is the reading of the left sensor at timestep t
    2. observations[t][1] is the reading of the right sensor at timestep t
    3. observations[t][2] is the reading of the up sensor at timestep t
    4. observations[t][3] is the reading of the down sensor at timestep t
    epsilon: the probability of a single sensor failing

    Return: a list of predictions for the agent's true hidden states.
    The expected output is a numpy array of shape (T, 2) where
    1. (predictions[t][0], predictions[t][1]) is the prediction for the state at timestep t
    """
    # TODO: implement the viterbi algorithm
```

```
T = len(observations)  # Number of timesteps
rows, cols = env._rows, env._columns

# Initialize DP table and backpointer
dp = np.zeros((T, rows, cols))
backPointer = np.zeros((T, rows, cols, 2), dtype=int)  # e.g. backPointer[5,2,3]=[1,3]

# Initialization step
for i in range(rows):
    for j in range(cols):
        dp[0, i, j] = observation_probability(env, i, j, observations[0], epsilon) / (rows * cols)

# Recursion step
for t in range(1, T):
    for i in range(rows):
        for j in range(cols):
            maxProb = -69
            bestPrevState = None

            for ni, nj in env.get_neighbors(i, j):  # Transition from neighbors
                prob = dp[t - 1, ni, nj] / len(env.get_neighbors(ni, nj))
                if prob > maxProb:
                    maxProb = prob
                    bestPrevState = (ni, nj)

            dp[t, i, j] = maxProb * observation_probability(env, i, j, observations[t], epsilon)
            backPointer[t, i, j] = bestPrevState

# Backtrace
predictions = np.zeros((T, 2), dtype=int)
finalState = np.unravel_index(dp[T - 1].argmax(), dp[T - 1].shape)
predictions[T - 1] = finalState
for t in range(T - 2, -1, -1):
    predictions[t] = backPointer[t + 1, predictions[t + 1, 0], predictions[t + 1, 1]]

return predictions
```

2. **Q3.1**

```
def langevin_update(grad_func, current_particles, noise, eta):
    """
    Runs a single Langevin update step on current_particles.
    Returns a tensor of shape (num_particles, 2).
    """
    next_particles = None
    ### start langevin_update ###
    # Compute the gradient of log p(x) at the current_particles
    grad = grad_func(current_particles)
    # Perform the Langevin update step
    next_particles = current_particles + eta * grad + torch.sqrt(2 * eta) * noise
    ### end langevin_update ###
    return next_particles

def sample_langevin(grad_func, particles, num_steps, eta):
    """
    Takes randomly initialized particles and runs them through a Langevin sampler.
    Returns a tensor of shape (num_steps, num_particles, 2).
    """
    particles_over_time = [particles]
    ### start sample_langevin ###
    current_particles = particles

    for step in range(num_steps):
        # Generate noise for each particle
        noise = torch.randn_like(current_particles)  # Same shape as current_particles
        # Update particles using Langevin dynamics
        current_particles = langevin_update(grad_func, current_particles, noise, eta)
        particles_over_time.append(current_particles)
    ### end sample_langevin ###
    particles_over_time = torch.stack(particles_over_time)
    return particles_over_time
```

Intro to Langevin Dynamics

This notebook is view-only. You will need to make a copy.

In this notebook, we will walk through a simple demo for Langevin dynamics, where the goal is to sample from a distribution $\mathrm{p}(x)$ using only its score function $\nabla_x \log \mathrm{p}(x)$. Here we assume a toy setting where $\mathrm{p}(x)$ is known. In most practical cases we only have access a dataset of samples $\mathcal{D} = \{x_0, x_1, \ldots, x_n\} \sim p(x)$, in which case we might use a technique called score matching to estimate the score function [1].

```
from IPython import display
from functools import partial
import imageio
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import torch
import torch.nn.functional as F
from tqdm import tqdm
```

Here we define the log pdf and the gradient of the log pdf (i.e., the score function). We also provide a function for plotting the target shape corresponding for this specific elliptical logpdf.

```
def logpdf(x, rx=2.5, ry=2.5, cx=0.0, cy=0.0):
    shifted_x = x - torch.tensor([cx, cy])
    scaled_x = shifted_x / torch.tensor([rx, ry])
    r = torch.linalg.norm(scaled_x, axis=-1)
    return -(r - 1)**2 / 0.033

def create_grad_func(logpdf, **kwargs):
  def grad_logpdf(x):
    x.requires_grad_(True)
    log_prob = logpdf(x, **kwargs)
    return torch.autograd.grad(log_prob.sum(), x)[0]
  return grad_logpdf

def create_shape(rx=2.5, ry=2.5, cx=0.0, cy=0.0):
  return  {
      "class": "Ellipse",
      "kwargs": {
          "width": 2 * rx,
          "height": 2 * ry,
          "xy": (cx, cy)
      }
  }
```

Here we define some utility functions for visualizing results.

```
def plot_frame(particles, step, shape, figsize=(4, 4), lim=(-3, 3)):
    particles_np = particles.detach().cpu().numpy()
    fig, ax = plt.subplots(figsize=figsize)
    ax.scatter(particles_np[step, :, 0], particles_np[step, :, 1], alpha=0.1, s=1, color='blue')
    ax.set_xlim(*lim)
    ax.set_ylim(*lim)
    ax.set_xlabel("x coord")
    ax.set_ylabel("y coord")
    ax.set_aspect('equal')
    ax.set_title(f'Langevin Sampler at t={step}')

    shape_cls = shape["class"]
    shape_patch = getattr(matplotlib.patches, shape_cls)(
        edgecolor='red',
        facecolor='none',
        linewidth=2,
        **shape["kwargs"]
    )
    ax.add_patch(shape_patch)

    fig.canvas.draw()
    buf = fig.canvas.buffer_rgba()
    image = np.asarray(buf)
```

```
    plt.close()
    return image

def plot_trajectory(particles, particle_idx, axis_names=["x", "y"], figsize=(10, 3), lim=(-3, 3)):
    particles_np = particles.detach().cpu().numpy()
    fig, ax = plt.subplots(1, len(axis_names), figsize=figsize)
    for axis, axis_name in enumerate(axis_names):
        trajectory = particles_np[:, particle_idx, axis]
        ax[axis].plot(trajectory)
        ax[axis].set_ylim(*lim)
        ax[axis].set_title(f"Trajectory of particle {particle_idx} along {axis_name}-axis")
        ax[axis].set_xlabel("timestep")
        ax[axis].set_ylabel(f"{axis_names[axis]} coord")

def frames_to_image(frames):
  w, h = frames[0].shape[1], frames[0].shape[0]
  collated = Image.new('RGB', (w * len(frames), h))
  for i, frame in enumerate(frames):
      collated.paste(Image.fromarray(frame), (i * w, 0))
  return collated

def frames_to_gif(frames, filename="temp.gif"):
  imageio.mimsave(filename, frames, fps=5, loop=0)
  return filename
```

Part (a)

Finish the implementation of `langevin_update` and `sample_langevin`.

Recall that the update equation at timestep t with step size η and random noise $\epsilon \sim \mathcal{N}(0, I)$ is

$$x_{t+1} = x_t + \eta \nabla_x \log p(x) + \sqrt{2\eta}\epsilon$$

```
def langevin_update(grad_func, current_particles, noise, eta):
    """
    Runs a single Langevin update step on current_particles.
    Returns a tensor of shape (num_particles, 2).
    """
    next_particles = None
    ### start langevin_update ###
    # Compute the gradient of log p(x) at the current_particles
    grad = grad_func(current_particles)
    # Perform the Langevin update step
    next_particles = current_particles + eta * grad + torch.sqrt(2 * eta) * noise
    ### end langevin_update ###
    return next_particles

def sample_langevin(grad_func, particles, num_steps, eta):
    """
    Takes randomly initialized particles and runs them through a Langevin sampler.
    Returns a tensor of shape (num_steps, num_particles, 2).
    """
    particles_over_time = [particles]
    ### start sample_langevin ###
    current_particles = particles

    for step in range(num_steps):
        # Generate noise for each particle
        noise = torch.randn_like(current_particles)  # Same shape as current_particles
        # Update particles using Langevin dynamics
        current_particles = langevin_update(grad_func, current_particles, noise, eta)
        particles_over_time.append(current_particles)
    ### end sample_langevin ###
    particles_over_time = torch.stack(particles_over_time)
    return particles_over_time
```

Now that you've completed your implementation, run the sampler and visualize results!

For Langevin sampling, you can control the number of particles (`num_particles`), the dimension of each particle (`num_dims`), the number of update steps (`num_steps`), and the step size (`eta`). You can also control the shape of the base logpdf, e.g. the radii of the x and y axes of the ellipse (`rx` and `ry`) and the center (`cx` and `cy`).

```
def sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs, init_particles=None):
  if init_particles is None:
    # Initialize particles
    init_particles = torch.randn(
        langevin_kwargs["num_particles"],
        langevin_kwargs["num_dims"],
        device=device
    )
  # Run langevin sampling
  data = sample_langevin(
      create_grad_func(logpdf, **ellipse_kwargs),
      init_particles,
      langevin_kwargs["num_steps"],
      langevin_kwargs["eta"]
  )
  # Plot results
  frames = []
  for t in tqdm(range(data.shape[0])):
    frames.append(plot_frame(data, t, create_shape(**ellipse_kwargs)))
  return frames
```

Part (b)

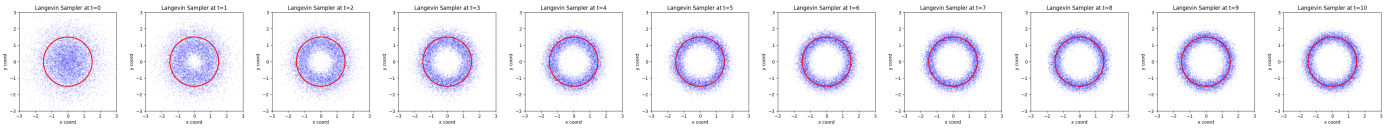
First, run sampling with the default hyperparameters.

Note: To simplify the runtime and plotting, throughout this problem you will only run the Langevin sampler for a few iterations. In practice, however, you would typically run the sampler for longer (e.g., several thousand iterations), to ensure the Markov chain has converged.

```
device = "cpu"
langevin_kwargs = {
    "num_particles": 10000,
    "num_dims": 2,
    "num_steps": 10,
    "eta": torch.tensor([1e-2, 1e-2])
}
ellipse_kwargs = {
    "rx": 1.5,
    "ry": 1.5,
    "cx": 0.0,
    "cy": 0.0
}

frames = sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs)
frames_to_image(frames)
```

```
100%|██████████| 11/11 [00:01<00:00,  7.42it/s]
```



Part (c)

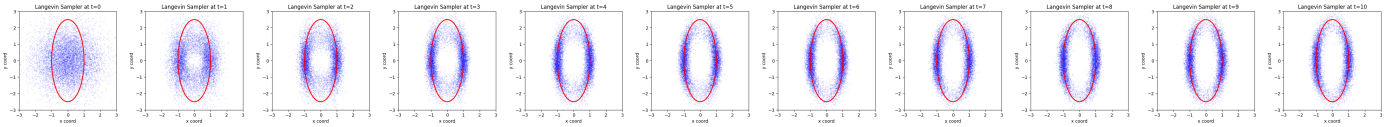
Now let's adjust the radius of the elliptical logpdf.

```
device = "cpu"
langevin_kwargs = {
    "num_particles": 10000,
    "num_dims": 2,
    "num_steps": 10,
    "eta": torch.tensor([1e-2, 1e-2])
}
ellipse_kwargs = {
    "rx": 1.0,
    "ry": 2.5,
    "cx": 0.0,
```

```
    "cy": 0.0
}

frames = sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs)
frames_to_image(frames)
```

```
100%|██████████| 11/11 [00:01<00:00,  6.62it/s]
```



Part (d)

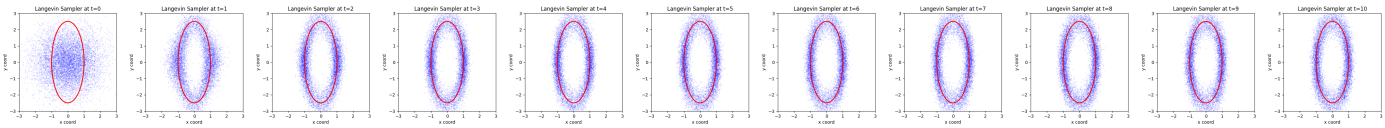
Let's see if we can get a better fit to the elliptical logpdf in (c) by tuning each dimension of `eta`.

```
eta = None
### start adjust eta ###
eta = torch.tensor([1e-2, 100e-3])
### end adjust eta ###

device = "cpu"
langevin_kwargs = {
    "num_particles": 10000,
    "num_dims": 2,
    "num_steps": 10,
    "eta": eta
}
ellipse_kwargs = {
    "rx": 1.0,
    "ry": 2.5,
    "cx": 0.0,
    "cy": 0.0
}

frames = sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs)
frames_to_image(frames)
```

```
100%|██████████| 11/11 [00:01<00:00,  7.38it/s]
```



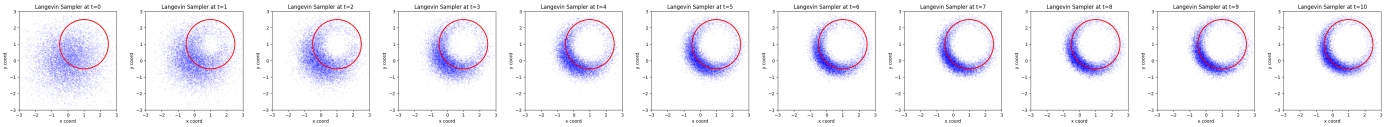
Part (e)

Here let's move the center of the logpdf away from the origin.

```
device = "cpu"
langevin_kwargs = {
    "num_particles": 10000,
    "num_dims": 2,
    "num_steps": 10,
    "eta": torch.tensor([1e-2, 1e-2])
}
ellipse_kwargs = {
    "rx": 1.5,
    "ry": 1.5,
    "cx": 1.0,
    "cy": 1.0
}

frames = sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs)
frames_to_image(frames)
```

```
100%|██████████| 11/11 [00:01<00:00,  8.58it/s]
```



Part (f)

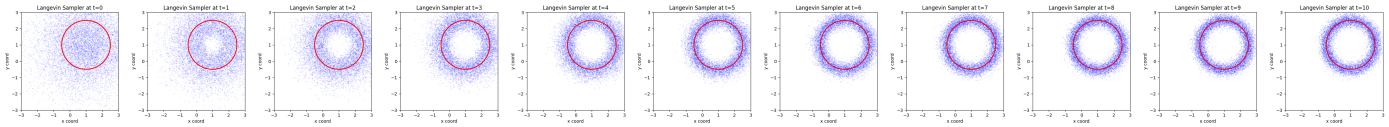
For the off-centered logpdf in (e), let's try to get a better fit by tuning the initialization `init_particles`.

```
init_particles = None
### start adjust init_particles ###
init_particles = torch.randn((langevin_kwargs["num_particles"], langevin_kwargs["num_dims"])) * torch.tensor([1.5, 1.5]) + torch.tensor([1.0,
### end adjust init_particles ###

device = "cpu"
langevin_kwargs = {
    "num_particles": 10000,
    "num_dims": 2,
    "num_steps": 10,
    "eta": torch.tensor([1e-2, 1e-2])
}
ellipse_kwargs = {
    "rx": 1.5,
    "ry": 1.5,
    "cx": 1.0,
    "cy": 1.0
}

frames = sample_and_viz_langevin(device, langevin_kwargs, ellipse_kwargs, init_particles=init_particles)
frames_to_image(frames)
```

```
100%|██████████| 11/11 [00:01<00:00,  8.95it/s]
```



Conclusion

That's it! Congratulations on finishing the notebook.

Contributors

Grace Luo, Saeed Saremi

Acknowledgements

This demo is based on Shreyas Kapur's blog post.

References

[1] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. NeurIPS 2019.

[2] Shreyas Kapur. Code Focused Guide on Score-Based Image Models. Blog Post 2023.

[3] Yang Song. Generative Modeling by Estimating Gradients of the Data Distribution. Blog Post. Blog Post 2021.