2.1 Flow Models

---

1. Fundamental Definition

Flow models are constructed from ODEs (Ordinary Differential Equations), which consist of three important elements:

- Trajectory
- Vector Filed (VF)
- Flow

Trajectory describes the path in the ODE space, which can be visualized as the path of a particle moving through space over time. It has the following form:

$$X : [0, 1] \to \mathbb{R}^d, \quad t \mapsto X_t$$

VF represents the direction (or can be thought of as the direction vector) in which particles move through space over time. We assume that all particles follow the direction specified by the VF. It has the following form:

$$u : \mathbb{R}^d \times [0, 1] \to \mathbb{R}^d, \quad (x, t) \mapsto u_t(x)$$

By defining **Trajectory** and **VF**, we can formulate the following ODE problem: given the initial condition $X_0 = x_0$. Then

$$\frac{d}{dt} X_t = u_t(X_t)$$

Flow is the solution to the above ODE problem. Flow ψ is a function

$$\psi : \mathbb{R}^d \times [0, 1] \mapsto \mathbb{R}^d, \quad (x_0, t) \mapsto \psi_t(x_0)$$

This solution has the following form: given the initial condition $\psi_0(x_0) = x_0$, then

$$\frac{d}{dt} \psi_t(X_t) = u_t(\psi_t(x_0))$$

Keypoint: ***Vector fileds define ODEs whose solutions are flows.***

---

2. Simulating an ODE

Numerical differential equations provide a fundamental simulation method, the **Euler method**.

Given the initial condition $X_0 = x_0$, the Euler method has the following iterative formula:

$$X_{t+h} = X_t + hu_t(X_t), \quad t = 0, h, 2h, \ldots, 1-h,$$

where $h = 1/n$ is the step size。

Note: We can directly plug in $u_t(X_t)$ here because $\frac{d}{dt}X_t = u_t(X_t)$.

---

3. Flow Models

Flow models are described by ODEs, which is

$$X_0 \sim p_{\text{init}}$$
$$\frac{d}{dt}X_t = u_t^\theta(X_t),$$

where p_{init} is an arbitrary distribution, and $u_t^\theta(X_t)$ is a VF constructed by a neural network with parameters θ.

The goal of flow models is to make the final X_1 as similar as p_{data} through the above learning process; that is, we believe X_1 comes from p_{data}。

Algorithm 1 Sampling from a Flow Model with Euler method

Require: Neural network vector field u_t^θ, number of steps n
1: Set $t = 0$
2: Set step size $h = \frac{1}{n}$
3: Draw a sample $X_0 \sim p_{\text{init}}$
4: **for** $i = 1, \ldots, n$ **do**
5: $\quad X_{t+h} = X_t + hu_t^\theta(X_t)$
6: $\quad$ Update $t \leftarrow t + h$
7: **end for**
8: **return** X_1

---

4. Simulation

The following simulation demonstrates the trajectory of a single particle and the trajectories of multiple particles under the action of a flow model.

In [1]:
```
# import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

from typing import Optional

import torch
```

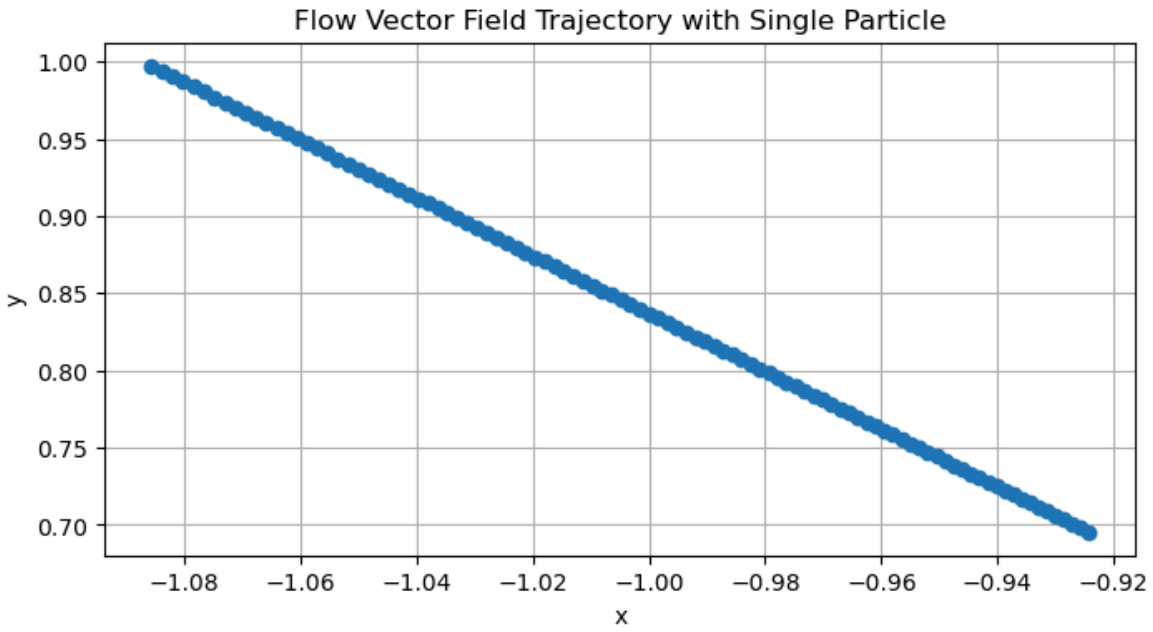
```
import torch.nn as nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [2]:
```
# set up an arbitrary neural network for flow step
class VF_step_flow(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(VF_step_flow, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

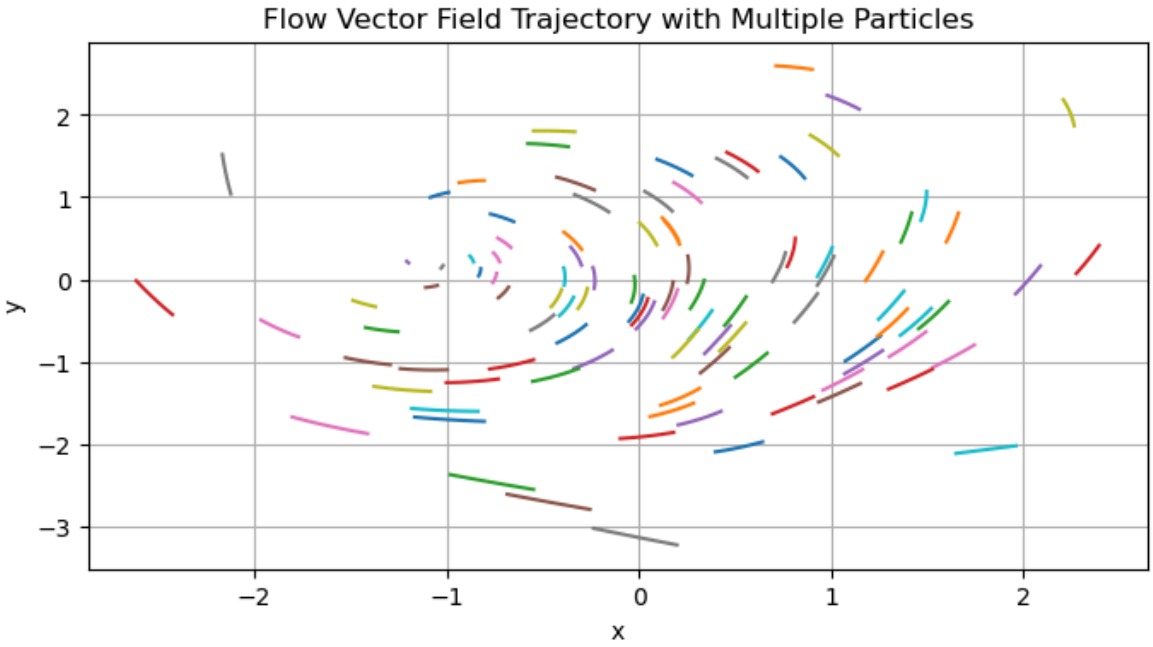
In [3]:
```
def flow_model(num_particle: int, step_size: int, random_seed: Optional[i

    """
    Simulate the flow field of particles using a neural network model.
    Args:
        num_particle (int): Number of particles to simulate.
        step_size (int): Number of steps for the simulation.
        random_seed (int, optional): Random seed for initial positions. D
    Returns:
        list: A list of numpy arrays representing the history of particle
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    # generate initial positions of particles
    X = np.random.normal(0, 1, size=(num_particle, 2))
    X_input = torch.tensor(X, dtype=torch.float32, requires_grad=False).t
    X_history = [X]

    model = VF_step_flow(input_size=2, hidden_size=10, output_size=2).to(

    t = 0   # time step
    h = 1 / step_size

    # main
    while t < 1:
        for _ in range(step_size):
            with torch.no_grad():
                X_input = X_input + h * model(X_input)
            X_history.append(X_input.cpu().detach().numpy())
            t = t + h

    return X_history
```

In [4]:
```
# simulate the trajectory of a single particle in the flow model
single_particle_flow = flow_model(num_particle=1, step_size=100, random_s

x = [single_particle_flow[i][0][0] for i in range(len(single_particle_flo
y = [single_particle_flow[i][0][1] for i in range(len(single_particle_flo
```

```
plt.figure(figsize=(8, 4))
plt.plot(x, y, marker="o")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Flow Vector Field Trajectory with Single Particle")
plt.show()
```



In [5]:
```
# simulate the trajectory of multiple particles in the flow model
multiple_particle_flow = flow_model(num_particle=100, step_size=100, rand

particle = {
    f"particle_{j}": [multiple_particle_flow[i][j] for i in range(len(mul
    for j in range(len(multiple_particle_flow[0]))
}

plt.figure(figsize=(8, 4))
for key in particle.keys():
    plt.plot(
        [particle[key][i][0] for i in range(len(particle[key]))],
        [particle[key][i][1] for i in range(len(particle[key]))]
    )
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Flow Vector Field Trajectory with Multiple Particles")
plt.show()
```



2.2 Diffusion Models

---

1. Fundamental Definition

Diffusion models are models constructed from SDEs (Stochastic Differential Equations). Their elements are similar to flow models, with the difference being that the trajectories in diffusion models have randomness.

For such trajectories with randomness, we can view them as a stochastic process $(X_t)_{0\leq t\leq 1}$, that is:

$$X_t \text{ is a random variable for every } 0 \leq t \leq 1$$

$$X : [0,1] \to \mathbb{R}^d, \quad t \mapsto X_t \text{ is a random trajectory for every draw of } X$$

Brownian Motion

In SDEs, **Brownian motion** is commonly used to describe the stochastic term. For a stochastic process $(W_t)_{0\leq t\leq 1}$, we have the following properties:

1. $W_0 = 0$.
2. **normal increments**: $W_t - W_s \sim N(0, t-s)$ for all $0 < s < t$.
3. **independent increments**: For any $0 \leq t_0 \leq t_1 \leq \ldots \leq t_n = 1$, the increments $W_{t_1} - W_{t_0}, \ldots, W_{t_n} - W_{t_{n-1}}$ are independent random variables.

A stochastic process satisfying these three properties is called Brownian motion or a Wiener process.

Brownian motion can be simulated using the following iterative formula. Given the initial condition $W_0 = 0$, then:

$$W_{t+h} = W_t + \sqrt{h}\varepsilon, \quad \varepsilon \sim N(0,1).$$

From ODEs to SDEs

Since SDEs contain stochastic terms, they cannot be differentiated like ODEs, so we need to perform some appropriate treatment.

First, we need to discretize the ODEs:

$$\begin{aligned} &\frac{d}{dt}X_t = u_t(X_t) \\ \Leftrightarrow\ &\frac{1}{h}(X_{t+h} - X_t) = u_t(X_t) + R_t(h) \\ \Leftrightarrow\ &X_{t+h} = X_t + hu_t(X_t) + hR_t(h), \end{aligned}$$

where $R_t(h)$ is the error term, satisfying $\lim_{h\to 0} R_t(h) = 0$.

Next, we add a stochastic term, namely Brownian motion, to the discretized ODEs:

$$X_{t+h} = X_t + hu_t(X_t) + \sigma_t(W_{t+h} - W_t),$$

where $hu_t(X_t)$ is the deterministic term, $W_{t+h} - W_t$ is the stochastic term, and σ_t is the diffusion coefficient. From the properties of Brownian motion, we can obtain $W_{t+h} - W_t \sim N(0, h)$.

Finally, we can formulate the SDE problem. Given the initial conditions $X_0 = x_0$ and $W_0 = 0$, then:

$$dX_t = u_t(X_t)dt + \sigma_t dW_t.$$

Note: There is no flow solution here; we can only rely on numerical solutions.

---

2. Simulation an SDE

Numerical differential equations provide a fundamental simulation method, the **Euler-Maruyama method**.

Given the initial condition $X_0 = x_0$, the Euler-Maruyama method has the following iterative formula:

$$X_{t+h} = X_t + hu_t(X_t) + \sqrt{h}\sigma_t\varepsilon, \quad \varepsilon \sim N(0,1) \text{ and } t = 0, h, 2h, \ldots, 1-h,$$

where $h = 1/n$ is the step size。

---

3. Diffusion Models

Diffusion models are described by SDEs, which is:

$$X_0 \sim p_{\text{init}}$$
$$dX_t = u_t^\theta(X_t)dt + \sigma_t dW_t,$$

where p_{init} is an arbitrary distribution, and $u_t^\theta(X_t)$ is a VF constructed by a neural network with parameters θ.

The goal of flow models is to make the final X_1 as similar as p_{data} through the above learning process; that is, we believe X_1 comes from p_{data}。

Algorithm 2 Sampling from a Diffusion Model (Euler-Maruyama method)

Require: Neural network u_t^θ, number of steps n, diffusion coefficient σ_t
1: Set $t = 0$
2: Set step size $h = \frac{1}{n}$
3: Draw a sample $X_0 \sim p_{\text{init}}$
4: **for** $i = 1, \ldots, n$ **do**
5: Draw a sample $\epsilon \sim \mathcal{N}(0, I_d)$
6: $X_{t+h} = X_t + hu_t^\theta(X_t) + \sigma_t\sqrt{h}\epsilon$
7: Update $t \leftarrow t + h$
8: **end for**
9: **return** X_1

---

4. Simulation

The following simulation demonstrates:

1. The path process of 1-D Brownian motion.
2. The trajectory of a single particle and the trajectories of

multiple particles under the action of a diffusion model.

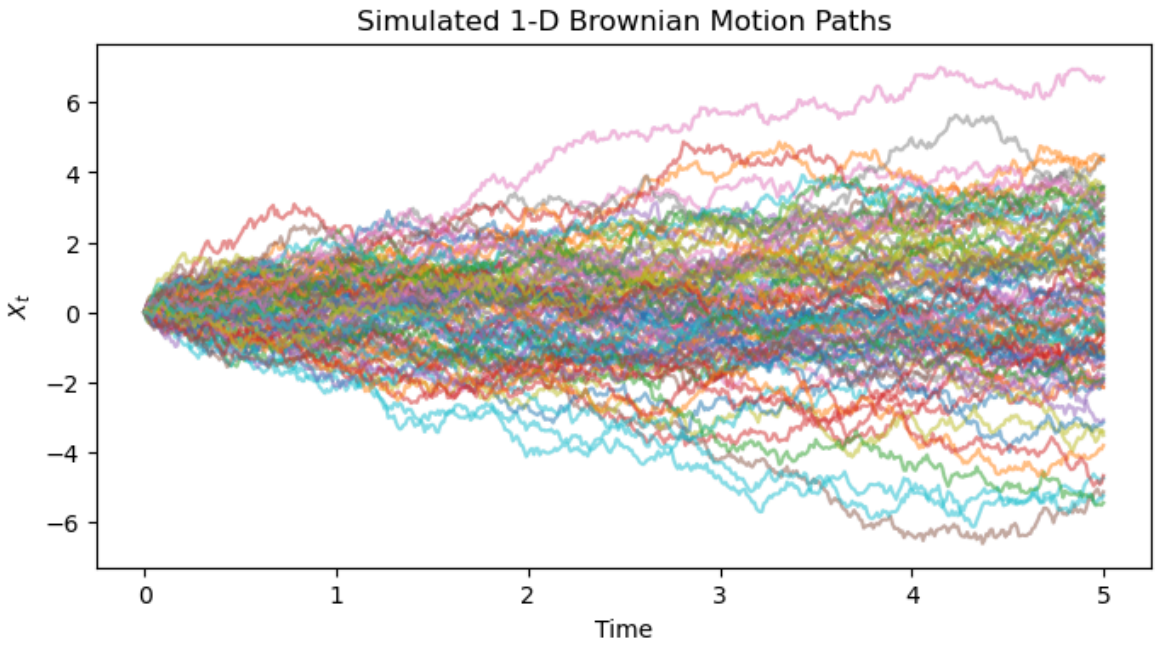
3. The effect of different diffusion coefficients on trajectories under the action of a diffusion model.

In [6]:
```
# import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

from typing import Optional

import torch
import torch.nn as nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [7]:
```
def brownian_motion(num_path: int, num_period: float, step_size: int, ran

    """
```

```
    Simulate the 1-D Brownian motion.
    Args:
        num_path (int): Number of paths to simulate.
        num_period (float): Total time duration.
        step_size (int): Number of discrete time steps.
        random_seed (int, optional): Random seed for reproducibility. Def
    Returns:
        np.ndarray: A 1D array of shape (num_path, step_size) representin
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    dt = num_period / step_size
    dW = np.sqrt(dt) * np.random.randn(num_path, step_size)
    X = np.cumsum(dW, axis=1)
    X = np.hstack((np.zeros((num_path, 1)), X))  # add initial position a

    return X
```

In [8]:
```
# simulate the 1-D Brownian motion
num_path = 100
num_period = 5
step_size = 500
BM = brownian_motion(num_path, num_period, step_size, random_seed=123)

time_points = np.linspace(0, num_period, step_size + 1)

plt.figure(figsize=(8, 4))
for path in range(BM.shape[0]):
    plt.plot(time_points, BM[path], alpha=0.5)
plt.xlabel("Time")
plt.ylabel("$X_t$")
plt.title("Simulated 1-D Brownian Motion Paths")
plt.show()
```



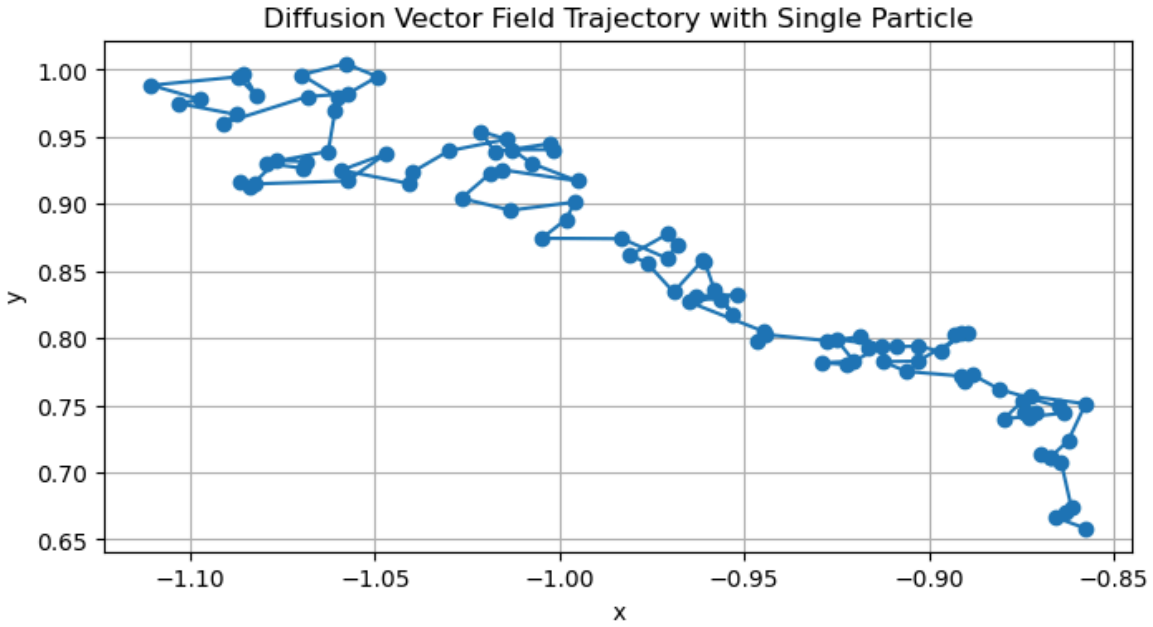
In [9]:
```
# set up an arbitrary neural network for diffusion step
class VF_step_diffusion(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
```

```
        super(VF_step_diffusion, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

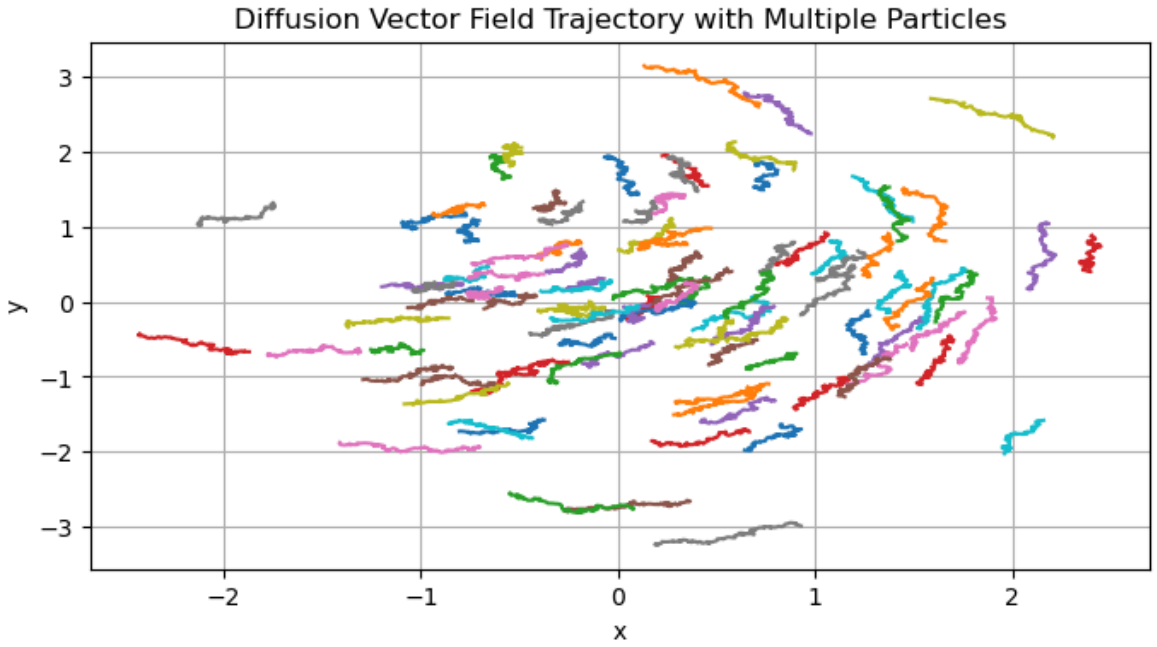
In [10]:
```
def diffusion_model(num_particle: int, step_size: int, sigma: float, rand

    """
    Simulate the diffusion field of particles using a neural network mode
    Args:
        num_particle (int): Number of particles to simulate.
        step_size (int): Number of steps for the simulation.
        sigma (float): Standard deviation for the diffusion term.
        random_seed (int, optional): Random seed for initial positions. D
    Returns:
        list: A list of numpy arrays representing the history of particle
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    # generate initial positions of particles
    X = np.random.normal(0, 1, size=(num_particle, 2))
    X_input = torch.tensor(X, dtype=torch.float32, requires_grad=False).t
    X_history = [X]

    model = VF_step_diffusion(input_size=2, hidden_size=10, output_size=2

    t = 0  # time step

    # main
    while t < 1:
        for _ in range(step_size):
            h = 1 / step_size
            with torch.no_grad():
                epsilon = np.random.normal(0, 1, size=(num_particle, 2))
                diffusion_term = torch.tensor(sigma * np.sqrt(h) * epsilo
                X_input = X_input + h * model(X_input) + diffusion_term
            X_history.append(X_input.cpu().detach().numpy())
            t = t + h

    return X_history
```

In [11]:
```
# simulate the trajectory of a single particle in the diffusion model
single_particle_diffusion = diffusion_model(num_particle=1, step_size=100

x = [single_particle_diffusion[i][0][0] for i in range(len(single_particl
y = [single_particle_diffusion[i][0][1] for i in range(len(single_particl

plt.figure(figsize=(8, 4))
plt.plot(x, y, marker="o")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
```

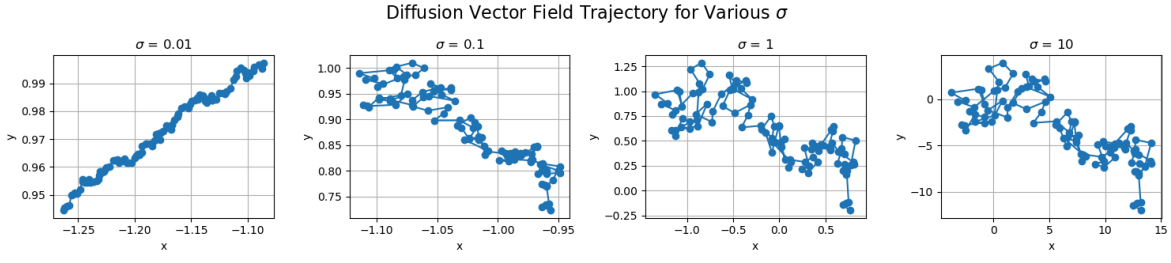
```
plt.title("Diffusion Vector Field Trajectory with Single Particle")
plt.show()
```



In [12]:
```
# simulate the trajectory of multiple particles in the diffusion model
multiple_particle_diffusion = diffusion_model(num_particle=100, step_size

particle = {
    f"particle_{j}": [multiple_particle_diffusion[i][j] for i in range(le
    for j in range(len(multiple_particle_diffusion[0]))
}

plt.figure(figsize=(8, 4))
for key in particle.keys():
    plt.plot(
        [particle[key][i][0] for i in range(len(particle[key]))],
        [particle[key][i][1] for i in range(len(particle[key]))]
    )
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Diffusion Vector Field Trajectory with Multiple Particles")
plt.show()
```

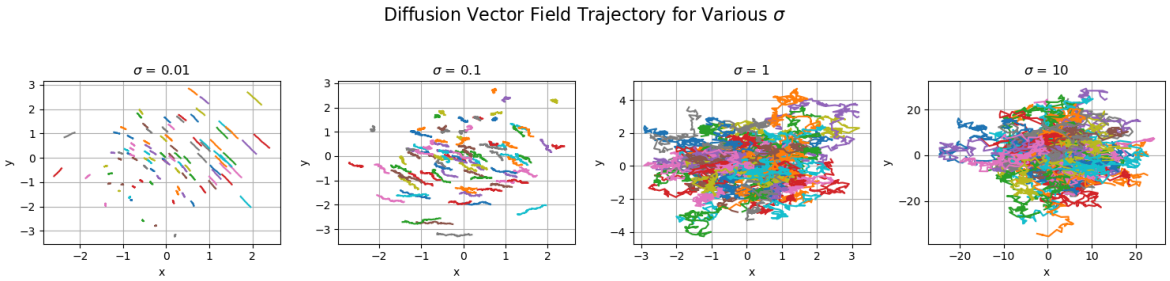


In [13]:
```
# simulate the trajectory of a single particle in the diffusion model for
sigma_list = [0.01, 0.1, 1, 10]
fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, sigma in enumerate(sigma_list):
    single_particle_diffusion = diffusion_model(num_particle=1, step_size

    x = [single_particle_diffusion[i][0][0] for i in range(len(single_par
    y = [single_particle_diffusion[i][0][1] for i in range(len(single_par

    axs[idx].plot(x, y, marker="o")
    axs[idx].grid()
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
    axs[idx].set_title(rf"$\sigma$ = {sigma}")

plt.tight_layout()
plt.suptitle(r"Diffusion Vector Field Trajectory for Various $\sigma$", f
plt.show()
```



In [14]:
```
# simulate the trajectory of multiple particles in the diffusion model fo
sigma_list = [0.01, 0.1, 1, 10]
fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, sigma in enumerate(sigma_list):
    multiple_particle_diffusion = diffusion_model(num_particle=100, step_

    particle = {
        f"particle_{j}": [multiple_particle_diffusion[i][j] for i in rang
```

```
        for j in range(len(multiple_particle_diffusion[0]))
    }
    for key in particle.keys():
        axs[idx].plot(
            [particle[key][i][0] for i in range(len(particle[key]))],
            [particle[key][i][1] for i in range(len(particle[key]))]
        )
    axs[idx].grid()
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
    axs[idx].set_title(rf"$\sigma$ = {sigma}")

plt.tight_layout()
plt.suptitle(r"Diffusion Vector Field Trajectory for Various $\sigma$", f
plt.show()
```



2.3 More Example, Simulation, and Discussion

In [15]:
```
# import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from typing import Optional

import torch
import torch.nn as nn
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

1. Implementing Brownian Motion with various sigma

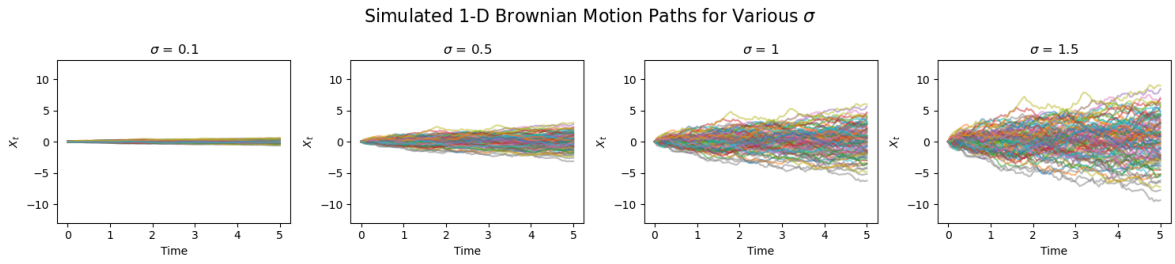
Given the initial condition $X_0 = 0$,

$$dX_t = \sigma dW_t$$

In [16]:
```
def brownian_motion_2(num_path: int, num_period: float, step_size: int, s

    """
    Simulate the 1-D Brownian motion.
    Args:
```

```
        num_path (int): Number of paths to simulate.
        num_period (float): Total time duration.
        step_size (int): Number of discrete time steps.
        sigma (float): Diffusion term for the Brownian motion.
        random_seed (int, optional): Random seed for reproducibility. Def
    Returns:
        np.ndarray: A 1D array of shape (num_path, step_size) representin
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    dt = num_period / step_size
    dW = sigma * np.sqrt(dt) * np.random.randn(num_path, step_size)
    X = np.cumsum(dW, axis=1)
    X = np.hstack((np.zeros((num_path, 1)), X)) # add initial position at

    return X
```

In [17]:
```
# simulate the 1-D Brownian motion with various sigma values
sigma_list = [0.1, 0.5, 1, 1.5]
num_path = 100
num_period = 5
step_size = 100
time_points = np.linspace(0, num_period, step_size + 1)

fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, sigma in enumerate(sigma_list):
    BM_2 = brownian_motion_2(num_path, num_period, step_size, sigma, rand
    for path in range(BM_2.shape[0]):
        axs[idx].plot(time_points, BM_2[path], alpha=0.5)
    axs[idx].set_ylim([-13, 13])
    axs[idx].set_xlabel("Time")
    axs[idx].set_ylabel("$X_t$")
    axs[idx].set_title(rf"$\sigma$ = {sigma}")

plt.tight_layout()
plt.suptitle(r"Simulated 1-D Brownian Motion Paths for Various $\sigma$",
plt.show()
```



---

2. Implementing an Ornstein-Uhlenbeck Process

For an Ornstein-Uhlenbeck process (OU process), the drift term is $u_t(X_t) = -\theta X_t$ and the stochastic term is $\sigma_t = \sigma$. Given the initial condition $X_0 = x_0$:

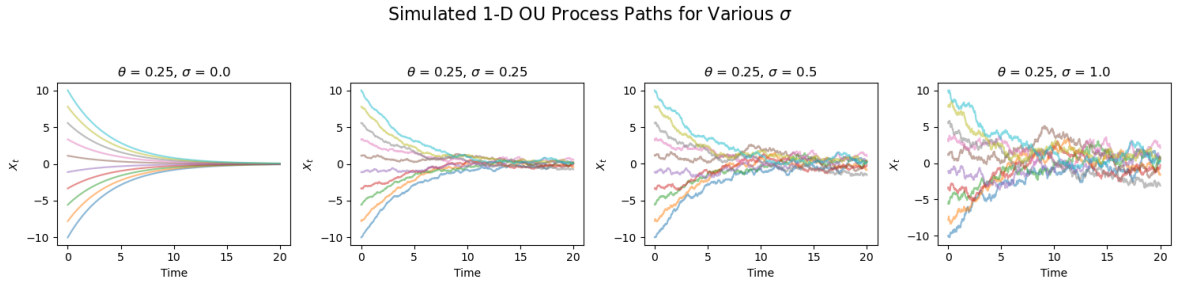
$$dX_t = -\theta X_t dt + \sigma dW_t.$$

Note: The OU process is a type of stochastic process with mean-reverting properties, where θ is the rate of reversion.

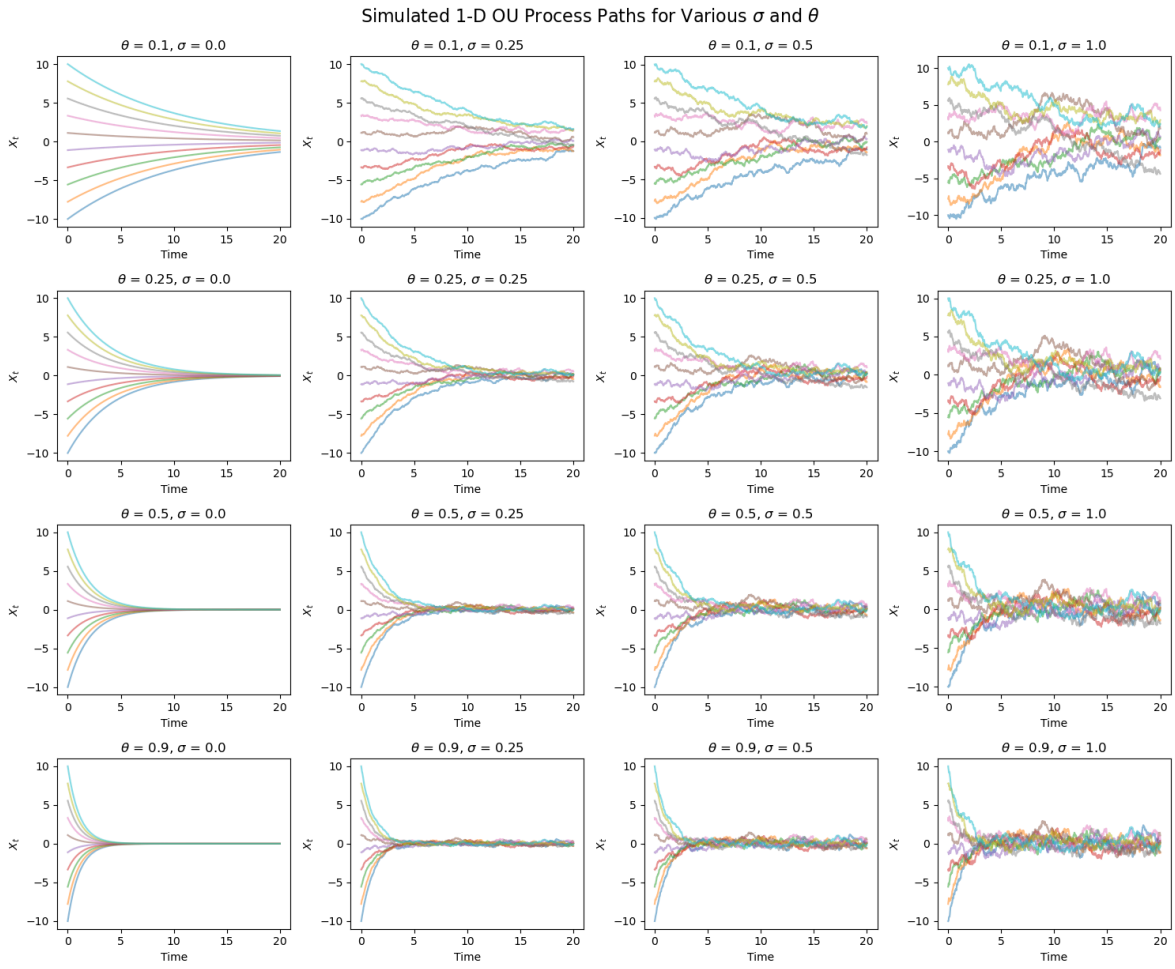
In [18]:
```
def OU_process(num_path: int, num_period: float, step_size: int,
               theta: float, sigma: float, random_seed: Optional[int]=Non

    """
    Simulate the 1-D Ornstein-Uhlenbeck (OU) process using Euler-Maruyama

    Args:
        num_path (int): Number of paths to simulate.
        num_period (float): Total time duration.
        step_size (int): Number of discrete time steps.
        theta (float): Mean-reversion rate.
        sigma (float): Volatility parameter.
        random_seed (int, optional): Random seed for reproducibility.

    Returns:
        np.ndarray: Array of shape (num_path, step_size + 1) with OU proc
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    dt = num_period / step_size

    X = np.zeros((num_path, step_size + 1))
    X[:, 0] = np.linspace(-10, 10, num_path)

    for i in range(1, step_size + 1):
        dW = np.sqrt(dt) * np.random.randn(num_path)
        X[:, i] = X[:, i - 1] - theta * X[:, i - 1] * dt + sigma * dW

    return X
```

In [19]:
```
# simulate the 1-D OU process with various sigma values
parameters = [
    (0.25, 0.0),
    (0.25, 0.25),
    (0.25, 0.5),
    (0.25, 1.0)
]
num_path = 10
num_period = 20
step_size = 1000
time_points = np.linspace(0, num_period, step_size + 1)

fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, (theta, sigma) in enumerate(parameters):
    OU = OU_process(num_path, num_period, step_size, theta, sigma, random
    for path in range(OU.shape[0]):
        axs[idx].plot(time_points, OU[path], alpha=0.5)
    axs[idx].set_xlabel("Time")
    axs[idx].set_ylabel("$X_t$")
    axs[idx].set_title(rf"$\theta$ = {theta}, $\sigma$ = {sigma}")
```

```
plt.tight_layout()
plt.suptitle(r"Simulated 1-D OU Process Paths for Various $\sigma$", font
plt.show()
```



In [20]:
```
# simulate the 1-D OU process with various sigma values and theta values
parameters = [(theta, sigma) for theta in [0.1, 0.25, 0.5, 0.9] for sigma
num_path = 10
num_period = 20
step_size = 1000
time_points = np.linspace(0, num_period, step_size + 1)

fig, axs = plt.subplots(4, 4, figsize=(15, 12))
axs = axs.flatten()

for idx, (theta, sigma) in enumerate(parameters):
    OU = OU_process(num_path, num_period, step_size, theta, sigma, random
    for path in range(OU.shape[0]):
        axs[idx].plot(time_points, OU[path], alpha=0.5)
    axs[idx].set_xlabel("Time")
    axs[idx].set_ylabel("$X_t$")
    axs[idx].set_title(rf"$\theta$ = {theta}, $\sigma$ = {sigma}")

plt.tight_layout()
plt.suptitle(r"Simulated 1-D OU Process Paths for Various $\sigma$ and $\
plt.show()
```



---

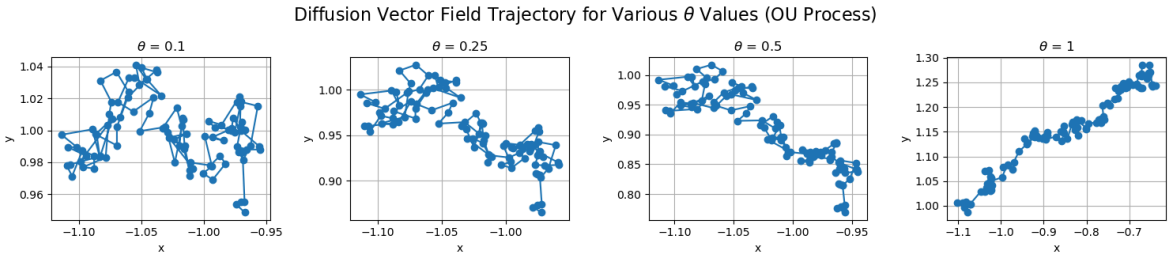
3. Simulating Diffusion Models with OU Process

The following simulation attempts to incorporate the OU process into diffusion models to demonstrate its effect on particles.

In [21]:
```python
def diffusion_model_OU(num_particle: int, step_size: int, theta: float, s

    """
    Simulate the diffusion field of particles based on OU process.
    Args:
        num_particle (int): Number of particles to simulate.
        step_size (int): Number of steps for the simulation.
        theta (float): Mean-reversion rate for the OU process.
        sigma (float): Standard deviation for the diffusion term.
        random_seed (int, optional): Random seed for initial positions. D
    Returns:
        list: A list of numpy arrays representing the history of particle
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    # generate initial positions of particles
    X = np.random.normal(0, 1, size=(num_particle, 2))
    X_input = torch.tensor(X, dtype=torch.float32, requires_grad=False).t
    X_history = [X]
```

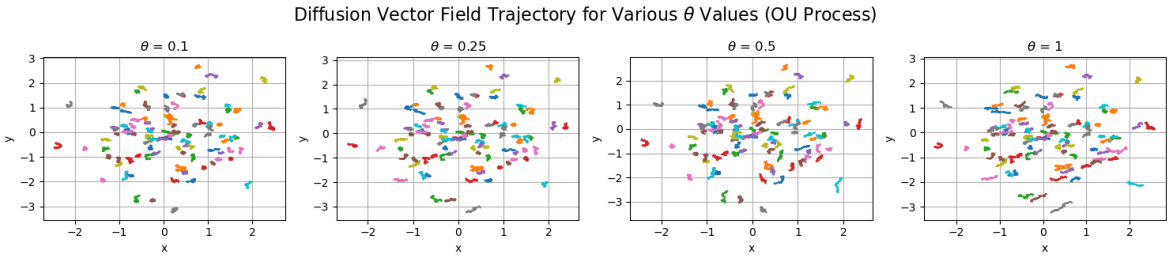
```
    model = VF_step_diffusion(input_size=2, hidden_size=10, output_size=2

    t = 0  # time step

    # main
    while t < 1:
        for _ in range(step_size):
            h = 1 / step_size
            with torch.no_grad():
                epsilon = np.random.normal(0, 1, size=(num_particle, 2))
                diffusion_term = torch.tensor(sigma * np.sqrt(h) * epsilo
                X_input = X_input - theta * h * model(X_input) + diffusio
            X_history.append(X_input.cpu().detach().numpy())
            t = t + h

    return X_history
```

In [22]:
```
# simulate the trajectory of a single particle in the diffusion model bas
theta_list = [0.1, 0.25, 0.5, 1]
fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, theta in enumerate(theta_list):
    single_particle_OU = diffusion_model_OU(num_particle=1, step_size=100

    x = [single_particle_OU[i][0][0] for i in range(len(single_particle_O
    y = [single_particle_OU[i][0][1] for i in range(len(single_particle_O

    axs[idx].plot(x, y, marker="o")
    axs[idx].grid()
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
    axs[idx].set_title(rf"$\theta$ = {theta}")

plt.tight_layout()
plt.suptitle(r"Diffusion Vector Field Trajectory for Various $\theta$ Val
plt.show()
```



In [23]:
```
# simulate the trajectory of multiple particles in the diffusion model ba
theta_list = [0.1, 0.25, 0.5, 1]
fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, theta in enumerate(theta_list):
    multiple_particle_OU = diffusion_model_OU(num_particle=100, step_size

    particle = {
        f"particle_{j}": [multiple_particle_OU[i][j] for i in range(len(m
        for j in range(len(multiple_particle_OU[0]))
    }
```

```
    for key in particle.keys():
        axs[idx].plot(
            [particle[key][i][0] for i in range(len(particle[key]))],
            [particle[key][i][1] for i in range(len(particle[key]))]
        )
    axs[idx].grid()
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
    axs[idx].set_title(rf"$\theta$ = {theta}")

plt.tight_layout()
plt.suptitle(r"Diffusion Vector Field Trajectory for Various $\theta$ Val
plt.show()
```



---

4. Transforming Distribution with SDEs

The above stochastic processes all observe how particles are transformed by SDEs; however, our true goal is to transform a distribution, that is, how to transform from p_{init} to p_{data}.

In the literature, there is a type of SDE that can accomplish this, namely **Langvein dynamic**, whose expression is as follows:

$$dX_t = \frac{1}{2}\sigma^2 \nabla \log p(X_t)dt + \sigma dWt$$

For this distribution, we have some constraints:

1. The distribution can be differentiated or the gradient can be computed.
2. We can sample from this distribution.

Here we also perform some simple simulations of Langevin dynamics.

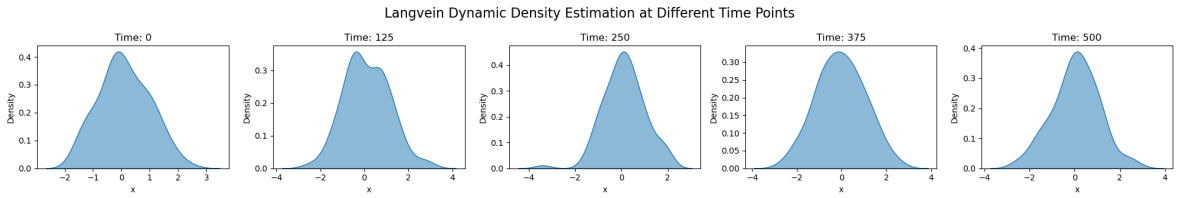
Case 1. Suppose $X \sim N(0,1)$, then $p(x) \propto e^{-x^2/x}$ and

$$\nabla \log p(x) = -x$$

In [24]:
```
def Langvein_dynamic(num_path: int, num_period: float, step_size: int, si
                     random_seed: Optional[int]=None) -> np.ndarray:

    """
    Simulate the Langvein dynamic process.
    Args:
        num_path (int): Number of paths to simulate.
        num_period (float): Total time duration.
```

```
        step_size (int): Number of discrete time steps.
        sigma (float): Standard deviation for the diffusion term.
        random_seed (int, optional): Random seed for reproducibility. Def
    Returns:
        np.ndarray: A 1D array of shape (num_path, step_size + 1) represe
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    dt = num_period / step_size

    X = np.zeros((num_path, step_size + 1))
    X[:, 0] = np.random.randn(num_path) # initial positions

    for i in range(1, step_size + 1):
        score = -X[:, i - 1]
        dW = np.sqrt(dt) * np.random.randn(num_path)
        X[:, i] = X[:, i - 1] + 0.5 * sigma ** 2 * score * dt + sigma * d

    return X
```

In [25]:
```
# simulate the Langvein dynamic density estimation at different time poin
num_path = 100
num_period = 20
step_size = 500
time_points = np.linspace(0, 500, 5)

Langvein = Langvein_dynamic(num_path, num_period, step_size, sigma=1)

particle = {f"particle_{j}": [Langvein[i][j] for i in range(len(Langvein)
            for j in range(len(Langvein[0]))}

fig, axs = plt.subplots(1, 5, figsize=(20, 3))
axs = axs.flatten()

for idx, key in enumerate(time_points):
    key = int(key)
    sns.kdeplot(particle[f"particle_{key}"], ax=axs[idx], fill=True, alph
    axs[idx].set_title(f"Time: {key}")
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("Density")
plt.tight_layout()
plt.suptitle("Langvein Dynamic Density Estimation at Different Time Point
plt.show()
```



Case 2. Suppose $\mathbf{X} \sim \mathcal{N}_2(\mu, \Sigma)$, then

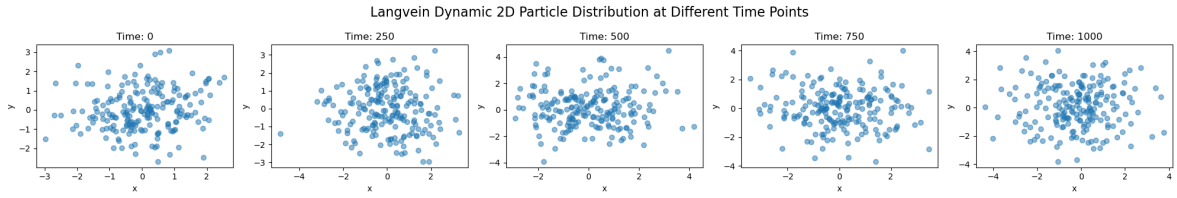
$$\nabla \log p(x) \propto -\Sigma^{-1}(\mathbf{x} - \mu)$$

In [26]:
```
def Langvein_dynamic_2D(num_path: int, num_period: float, step_size: int,
                        mu: np.ndarray, cov: np.ndarray, random_seed: Opt
```

```
    """
    Simulate the 2-D Langvein dynamic process.
    Args:
        num_path (int): Number of paths to simulate.
        num_period (float): Total time duration.
        step_size (int): Number of discrete time steps.
        sigma (float): Standard deviation for the diffusion term.
        mu (np.ndarray): Mean vector for the multivariate normal distribu
        cov (np.ndarray): Covariance matrix for the multivariate normal d
        random_seed (int, optional): Random seed for reproducibility. Def
    Returns:
        np.ndarray: A 2D array of shape (num_path, step_size + 1, 2) repr
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    dt = num_period / step_size
    dim = 2
    inv_cov = np.linalg.inv(cov)

    X = np.zeros((num_path, step_size + 1, dim))
    X[:, 0, :] = np.random.multivariate_normal(mean=mu, cov=cov, size=num

    for i in range(1, step_size + 1):
        score = -((X[:, i - 1, :] - mu) @ inv_cov.T)  # score function
        dW = np.sqrt(dt) * np.random.randn(num_path, dim)
        X[:, i, :] = X[:, i - 1, :] + 0.5 * sigma ** 2 * score * dt + sig

    return X
```

In [27]:

```
# simulate the Langvein dynamic distribution estimation at different time
num_path = 200
num_period = 20
step_size = 1000
time_points = np.linspace(0, step_size, 5)

Langvein_2D = Langvein_dynamic_2D(num_path, num_period, step_size, sigma=

particle_2D = {f"particle_{j}": [Langvein_2D[i][j] for i in range(len(Lan

fig, axs = plt.subplots(1, 5, figsize=(20, 3))
axs = axs.flatten()

for idx, key in enumerate(time_points):
    key = int(key)
    x = [particle_2D[f"particle_{key}"][i][0] for i in range(len(particle
    y = [particle_2D[f"particle_{key}"][i][1] for i in range(len(particle
    axs[idx].scatter(x, y, alpha=0.5)
    axs[idx].set_title(f"Time: {key}")
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
plt.tight_layout()
plt.suptitle("Langvein Dynamic 2D Particle Distribution at Different Time
plt.show()
```



In [28]:
```python
def diffusion_model_Langvein(num_particle: int, step_size: int, sigma: fl

    """
    Simulate the diffusion field of particles based on Langvein dynamic a
    Args:
        num_particle (int): Number of particles to simulate.
        step_size (int): Number of steps for the simulation.
        sigma (float): Standard deviation for the diffusion term.
        random_seed (int, optional): Random seed for initial positions. D
    Returns:
        list: A list of numpy arrays representing the history of particle
    """

    if random_seed is not None:
        np.random.seed(random_seed)

    # generate initial positions of particles
    X = np.random.normal(0, 1, size=(num_particle, 2))
    X_input = torch.tensor(X, dtype=torch.float32, requires_grad=False).t
    X_history = [X]

    model = VF_step_diffusion(input_size=2, hidden_size=10, output_size=2

    t = 0  # time step

    # main
    while t < 1:
        for _ in range(step_size):
            h = 1 / step_size
            with torch.no_grad():
                score = -X_input
                epsilon = np.random.normal(0, 1, size=(num_particle, 2))
                diffusion_term = torch.tensor(sigma * np.sqrt(h) * epsilo
                X_input = X_input + 0.5 * sigma ** 2 * score * h * model(
            X_history.append(X_input.cpu().detach().numpy())
            t = t + h

    return X_history
```

In [29]:
```python
# simulate the trajectory of multiple particles in the diffusion model fo
sigma_list = [0.01, 0.1, 1, 1.5]
fig, axs = plt.subplots(1, 4, figsize=(15, 3))
axs = axs.flatten()

for idx, sigma in enumerate(sigma_list):
    multiple_particle_diffusion = diffusion_model_Langvein(num_particle=1

    particle = {
        f"particle_{j}": [multiple_particle_diffusion[i][j] for i in rang
        for j in range(len(multiple_particle_diffusion[0]))
    }
    for key in particle.keys():
        axs[idx].plot(
```

```
            [particle[key][i][0] for i in range(len(particle[key]))],
            [particle[key][i][1] for i in range(len(particle[key]))]
        )
    axs[idx].grid()
    axs[idx].set_xlabel("x")
    axs[idx].set_ylabel("y")
    axs[idx].set_title(rf"$\sigma$ = {sigma}")

plt.tight_layout()
plt.suptitle(r"Diffusion Vector Field Trajectory for Various $\sigma$ (La
plt.show()
```