3. Neural ODEs and Flows

In the previous section, we interpreted discrete residual networks and Transformers as discretized approximations of ODEs. One significant limitation of such models is that the discretizations use fixed step sizes (i.e., $\Delta t = 1$), which prevents the models from adapting their computational effort to the complexity of the input instance. Such coarse discretizations are often ill-suited for problems requiring fine-grained modeling, where the underlying dynamics may evolve rapidly and require smaller time steps for accurate approximation. Furthermore, since the number of steps is determined in the model design phase, the network depth becomes a static hyperparameter rather than an adaptive property.

In this section, we begin by looking at **neural ODEs** [Chen et al., 2018], which model the evolution of system dynamics by combining neural networks and continuous-time ODEs. An important property of neural ODEs is that they treat the network depth as a continuous time variable. Instead of specifying a fixed sequence of discrete layers, the derivative of the hidden state is defined by a continuous-time ODE, and the final output is obtained by solving this ODE using a black-box ODE solver. Moreover, we describe the training of neural ODEs, in particular the adjoint sensitivity method which provides a memory-efficient way to optimize such models. Finally, we discuss the concept of **continuous normalizing flows**, which is a natural extension of neural ODEs for describing the continuous evolution of probability densities. This framework lays the foundation for further discussions on generative modeling (see Section 4).

3.1 Neural ODEs

We have seen that in residual networks, the hidden state is updated according to $\mathbf{x}_{l+1} = \mathbf{x}_l + \Delta t \cdot F(\mathbf{x}_l, \theta_l)$. This equation corresponds to the Euler discretization of a continuous-time ODE as $\Delta t \to 0$

$$\frac{d\mathbf{x}(t)}{dt} \quad = \quad f(\mathbf{x}(t), \theta(t)). \tag{1}$$

Here, we use two function notations F and f to distinguish between the discrete residual mapping and the instantaneous rate of change of the state, respectively.

A key distinction between the two models lies in the parameterization. In standard residual networks, each layer l has its own distinct set of parameters θ_l. In the continuous limit expressed in Eq. (1), the parameters $\theta(t)$ can vary continuously with time. However, in practice, standard neural ODEs typically use a shared set of parameters θ across the entire integration interval. To allow the dynamics to vary with depth (time), the current time t is explicitly appended as an input to the network. Thus, the dynamics can be formulated using a non-autonomous ODE

$$\frac{d\mathbf{x}(t)}{dt} \quad = \quad f(\mathbf{x}(t), \theta, t). \tag{2}$$

The introduction of the continuous time variable t as a replacement for the layer index l allows the model to evaluate the state at any time point rather than at fixed time steps.

Neural ODEs are called *neural* because the dynamical function $f(\cdot)$ is a neural network. In other words, we use a neural network to estimate the dynamics of the system. Given a neural ODE, there are two fundamental processes, as in supervised learning problems:

- **Inference** (Forward Pass). The inference process aims to compute the state at any time given the parameters.
- **Training** (Backward Pass). The training process aims to optimize the parameters θ to minimize a loss function.

For inference, the input to the model is the initial state $\mathbf{x}(t_0)$, and the output is the state $\mathbf{x}(t_1)$ at some final time t_1. That is, the forward pass is essentially the task of solving this IVP. The output state $\mathbf{x}(t_1)$ is the solution to the ODE, and is computed by integrating the dynamics

$$\mathbf{x}(t_1) \quad = \quad \mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(\tau), \tau; \theta) d\tau. \tag{3}$$

Since the integral of a complex neural network f cannot be evaluated analytically, the output is obtained using a numerical ODE solver (e.g., Euler, Runge-Kutta, or more advanced methods). This allows the evaluation

$$\mathbf{x}(t_1) = \mathrm{ODESolve}(\mathbf{x}(t_0), f, t_0, t_1, \theta). \tag{4}$$

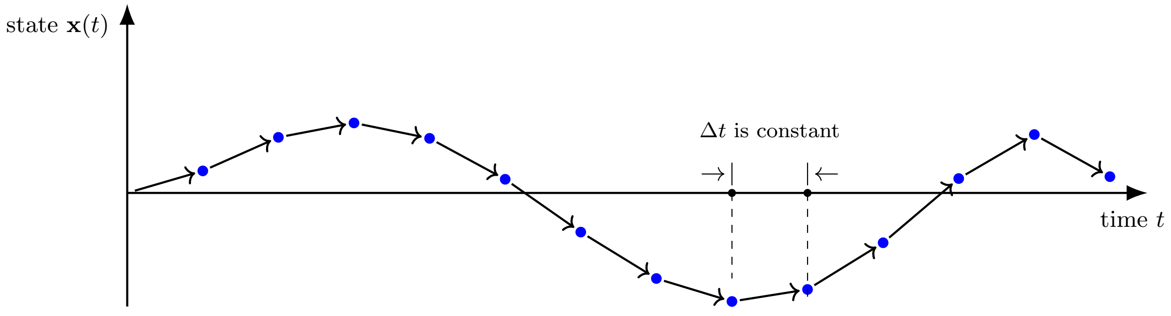
Here, ODESolve$(\cdot)$ represents a general ODE solver, which is typically implemented using off-the-shelf numerical integration tools. This formulation decouples the definition of the dynamics from the computation of the trajectory. So the solver can adapt the step size based on the complexity of the trajectory, which is a significant feature not present in fixed-depth residual networks. Figure 1 shows illustration of ODE solving with fixed and dynamic step sizes.

The training of neural ODEs can be seen as a standard task of training deep neural networks, where the parameters θ are updated iteratively by computing the gradient of the loss function with respect to these parameters. A straightforward approach is to perform standard backpropagation through the operations of the ODE solver. However, this often suffers from a high memory cost because it requires storing the intermediate states of the entire trajectory to apply the chain rule. The memory bottleneck may restrict the use of fine-grained solvers or long integration times. To address this, neural ODEs are typically trained using the **adjoint sensitivity method**, which computes gradients with a constant memory cost with respect to the number of integration steps. We will discuss the details of training neural ODEs further in Section 3.2.

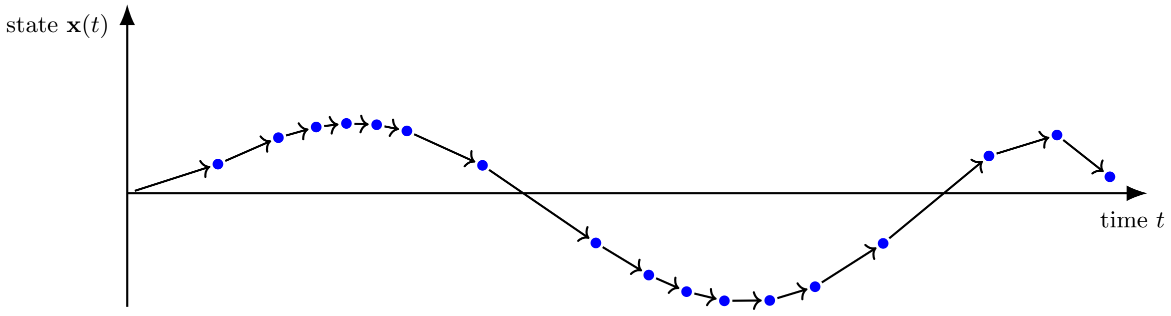
Simply put, neural ODEs use a neural network $f(\cdot)$ to approximate the dynamics of a system $\frac{d\mathbf{x}(t)}{dt}$. In the training stage, we optimize this network to model these dynamics. Then, in the testing stage, we can recover the state of the system at any given moment by simply solving the neural ODE.

3.2 The Adjoint Sensitivity Method

The adjoint sensitivity method (or the adjoint method) is a classical technique from the field of optimal control used to compute sensitivities for systems governed by differential equations.



(a) A system whose state evolves with a fixed time step (like residual networks)



(b) A system whose state evolves continuously or with a dynamic time step (like neural ODEs)

Figure 1: Illustrations of ODE solving with fixed and dynamic step sizes. Subfigure (a) shows a trajectory generated with a constant step size, similar to the discrete steps in a residual network (which can be seen as an Euler discretization of an ODE). Subfigure (b) shows a trajectory obtained with an dynamic step-size solver, which can be adopted in neural ODEs that model continuous state evolution.

In deep learning, it can be interpreted as continuous-time backpropagation. The core idea is to introduce a set of auxiliary variables, known as the **adjoint states**, which track how the gradient of the loss evolves backwards through time. It computes the gradients of the model parameters by solving a second differential equation, rather than explicitly applying the chain rule to the discrete operations of the forward pass.

This distinction should be emphasized when comparing it to standard backpropagation. In a naive implementation (often referred to as discretize-then-optimize), one would treat the numerical ODE solver as a computation graph with a finite number of layers and backpropagate directly through the internal operations of the solver. However, since an ODE solver may take thousands of small steps to achieve high precision, standard backpropagation requires storing the intermediate activations for every single step. This leads to prohibitive memory usage that scales linearly with the number of integration steps ($O(T)$). In other words, the discretization of ODE solving with a large number of steps makes the training memory-intensive, or sometimes even infeasible.

Formally, let us define the adjoint state $\mathbf{a}(t)$ as the gradient of the loss function $\mathcal{L}$ with respect to the hidden state $\mathbf{x}(t)$ at any given time t

$$\mathbf{a}(t) \quad = \quad \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t)}. \tag{5}$$

Intuitively, $\mathbf{a}(t)$ represents the sensitivity of the final loss to a small perturbation in the state at time t. Since the loss is computed based on the final state $\mathbf{x}(t_1)$, the boundary condition for the adjoint state is known

$$\mathbf{a}(t_1) \quad = \quad \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)}. \tag{6}$$

Note that while $\mathbf{x}(t)$ is solved forward from t_0 to t_1, the adjoint state $\mathbf{a}(t)$ must be solved backwards from t_1 to t_0, using $\mathbf{a}(t_1)$ as the initial value for the backward integration. Figure 2 shows a high-level illustration of the method.

3.2.1 Adjoint Dynamics

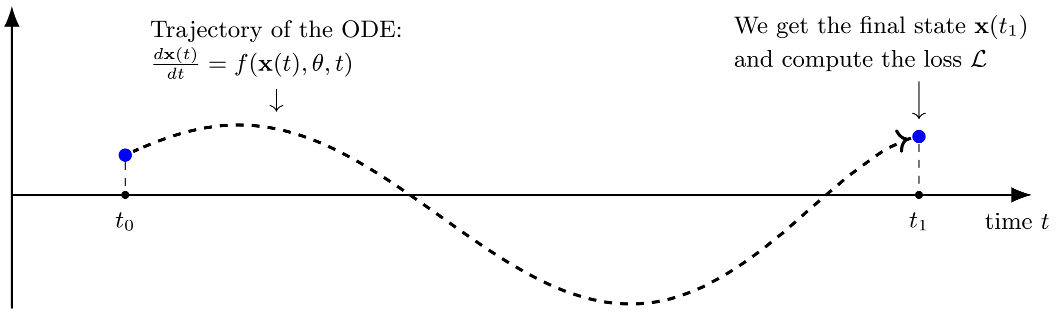
Instead of backpropagating through steps of the solver, we compute $\mathbf{a}(t)$ by solving a new ODE. To understand the dynamics of $\mathbf{a}(t)$, consider the discretization of the forward pass using a simple Euler method with a step size Δt. The state update from t to $t + \Delta t$ is

$$\mathbf{x}(t + \Delta t) \quad = \quad \mathbf{x}(t) + \Delta t \cdot f(\mathbf{x}(t), \theta, t). \tag{7}$$

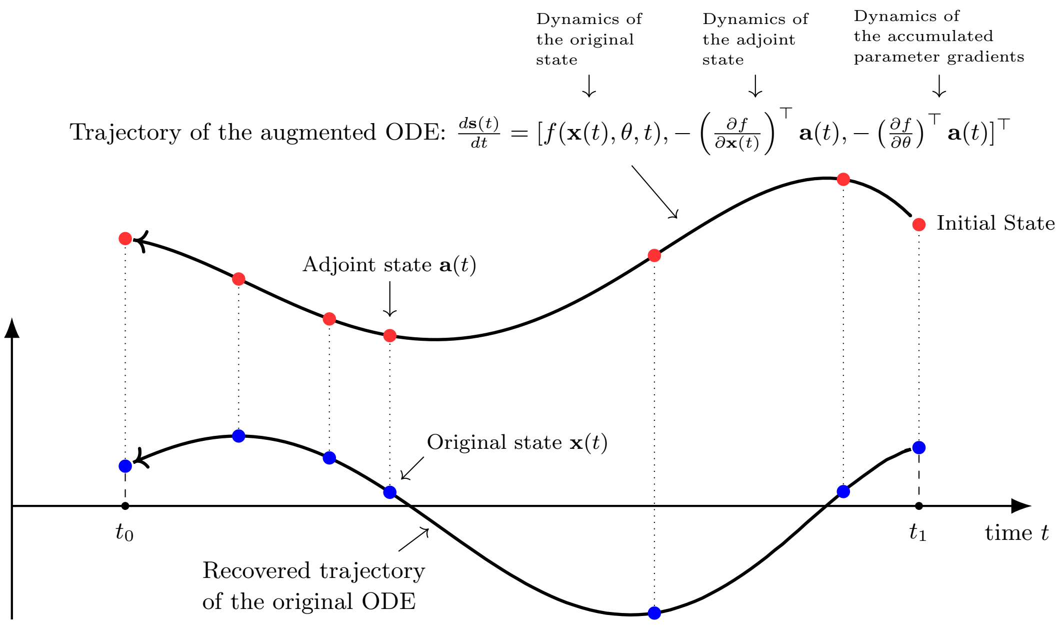
Applying the chain rule, the gradient of the loss with respect to the hidden state $\mathbf{x}(t)$ can be expressed as

$$\begin{aligned} \mathbf{a}(t) &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t)} \\ &= \left(\frac{\partial \mathbf{x}(t+\Delta t)}{\partial \mathbf{x}(t)}\right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t+\Delta t)} \\ &= \left(\frac{\partial \mathbf{x}(t+\Delta t)}{\partial \mathbf{x}(t)}\right)^{\top} \mathbf{a}(t+\Delta t), \end{aligned} \tag{8}$$

where $\frac{\partial \mathbf{x}(t+\Delta t)}{\partial \mathbf{x}(t)} \in \mathbb{R}^{d \times d}$ is the Jacobian matrix and $\mathbf{a}(t), \mathbf{a}(t+\Delta t) \in \mathbb{R}^d$ are column vectors.


Trajectory of the ODE:
$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta, t)$
We get the final state $\mathbf{x}(t_1)$
and compute the loss $\mathcal{L}$
t_0
t_1
time t

(a) Forward pass (integrating the dynamics of the ODE from t_0 to t_1)


Dynamics of the original state
Dynamics of the adjoint state
Dynamics of the accumulated parameter gradients
Trajectory of the augmented ODE: $\frac{d\mathbf{s}(t)}{dt} = [f(\mathbf{x}(t), \theta, t), -\left(\frac{\partial f}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t), -\left(\frac{\partial f}{\partial \theta}\right)^\top \mathbf{a}(t)]^\top$
Initial State
Adjoint state $\mathbf{a}(t)$
Original state $\mathbf{x}(t)$
t_0
t_1
time t
Recovered trajectory
of the original ODE

(b) Backward pass (integrating the dynamics of the augmented ODE from t_1 to t_0)

Figure 2: Illustration of the adjoint sensitivity method in neural ODEs [Chen et al., 2018]. There are two steps. In the forward pass (subfigure (a)), the state $\mathbf{x}(t)$ evolves from t_0 to t_1 by integrating the dynamics defined by the neural network $f(\mathbf{x}(t), \theta, t)$. The final state $\mathbf{x}(t_1)$ is used to compute the scalar loss $\mathcal{L}$. In the backward pass (subfigure (b)), instead of backpropagating through the solver's discrete steps, an augmented ODE system is solved backwards from t_1 to t_0. This augmented system simultaneously reconstructs the original state trajectory $\mathbf{x}(t)$, computes the adjoint state $\mathbf{a}(t)$ (the gradient of the loss with respect to the state), and accumulates the gradients with respect to the parameters θ. This approach allows for gradient computation with $O(1)$ memory cost as it avoids storing intermediate states during integration.

Substituting Eq. (7) into Eq. (8) yields

$$\begin{aligned}\mathbf{a}(t) &= \left(\frac{\partial\big(\mathbf{x}(t)+\Delta t\cdot f(\mathbf{x}(t),\theta,t)\big)}{\partial \mathbf{x}(t)}\right)^{\top}\mathbf{a}(t+\Delta t)\\ &= \left(\frac{\partial \mathbf{x}(t)+\partial\Delta t\cdot f(\mathbf{x}(t),\theta,t)}{\partial \mathbf{x}(t)}\right)^{\top}\mathbf{a}(t+\Delta t)\\ &= \left(\mathbf{I}+\Delta t\cdot\frac{\partial f(\mathbf{x}(t),\theta,t)}{\partial \mathbf{x}(t)}\right)^{\top}\mathbf{a}(t+\Delta t). \qquad (9)\end{aligned}$$

Rearranging this equation yields the difference quotient for the adjoint state

$$\frac{\mathbf{a}(t+\Delta t)-\mathbf{a}(t)}{\Delta t} = -\left(\frac{\partial f(\mathbf{x}(t),\theta,t)}{\partial \mathbf{x}(t)}\right)^{\top}\mathbf{a}(t+\Delta t). \qquad (10)$$

By taking the limit as $\Delta t \to 0$, we obtain the continuous-time dynamics of the adjoint state. This leads to the differential equation describing the instantaneous rate of change of $\mathbf{a}(t)$

$$\frac{d\mathbf{a}(t)}{dt} = -\left(\frac{\partial f(\mathbf{x}(t),\theta,t)}{\partial \mathbf{x}(t)}\right)^{\top}\mathbf{a}(t). \qquad (11)$$

Here, $\frac{\partial f}{\partial \mathbf{x}}$ is the Jacobian of the dynamics function. This equation tells us that the adjoint state is defined by a linear ODE that depends on the Jacobian of the original function $f(\cdot)$.

Similarly, we can derive the gradient for the parameters θ. At each time step t, the parameters θ make a small contribution to the change in state, which in turn affects the loss. The instantaneous gradient contribution at time t is the sensitivity of the loss to the state ($\mathbf{a}(t)$) multiplied by the sensitivity of the state change to the parameters ($\frac{\partial f}{\partial \theta}$). The total gradient is the accumulation (integral) of these instantaneous contributions over the entire trajectory

$$\frac{\partial \mathcal{L}}{\partial \theta} = -\int_{t_1}^{t_0}\left(\frac{\partial f(\mathbf{x}(t),\theta,t)}{\partial \theta}\right)^{\top}\mathbf{a}(t)dt. \qquad (12)$$

Note that the integral is performed backwards from t_1 to t_0, consistent with the backward pass of backpropagation.

3.2.2 The Augmented ODE and Memory Efficiency

Eqs. (11) and (12) give exact gradients but depend on the state trajectory $\mathbf{x}(t)$ at all times t. In standard backpropagation, this requires storing all intermediate states $\mathbf{x}(t)$ from the forward pass, resulting in $O(T)$ memory cost.

The key advantage of the adjoint sensitivity method is to avoid this storage by reconstructing the state trajectory backwards in time. Since the ODE describing the state evolution is deterministic and typically invertible, we can recover $\mathbf{x}(t)$ from the final state $\mathbf{x}(t_1)$ by running the dynamics backwards[1].

To compute the gradients efficiently, we construct an **augmented state** that contains the original state, the adjoint state, and the accumulated parameter gradients

$$\mathbf{s}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{a}(t) \\ \frac{\partial \mathcal{L}}{\partial \theta}(t) \end{bmatrix}, \tag{13}$$

where $\frac{\partial \mathcal{L}}{\partial \theta}(t)$ is the backwards integral from t_1 to t, i.e., $\frac{\partial \mathcal{L}}{\partial \theta}(t) = -\int_{t_1}^{t} \left(\frac{\partial f(\mathbf{x}(\tau),\theta,\tau)}{\partial \theta}\right)^\top \mathbf{a}(\tau)d\tau$. The dynamics of this combined system are solved jointly backwards from t_1 to t_0:

$$\frac{d\mathbf{s}(t)}{dt} = \begin{bmatrix} f(\mathbf{x}(t),\theta,t) \\ -\left(\frac{\partial f}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t) \\ -\left(\frac{\partial f}{\partial \theta}\right)^\top \mathbf{a}(t) \end{bmatrix}, \tag{14}$$

with the initial condition

$$\mathbf{s}(t_1) = \begin{bmatrix} \mathbf{x}(t_1) \\ \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)} \\ \mathbf{0} \end{bmatrix}. \tag{15}$$

The first component of the augmented dynamics is simply the original ODE. This means that as the solver integrates backwards, it reconstructs the trajectory $\mathbf{x}(t)$ on-the-fly. At any specific time t, the value of $\mathbf{x}(t)$ required to evaluate the Jacobians (in the second and third components) is readily available within the current augmented state $\mathbf{s}(t)$. As a result, we do not need to store the intermediate states of the forward pass, and the memory complexity is reduced from $O(T)$ to $O(1)$. In essence, the adjoint method trades the computational overhead of re-solving the state trajectory for memory savings. Note that this augmented ODE can be solved using off-the-shelf numerical solvers. One can also balance computational efficiency and numerical precision by simply adjusting the tolerance parameters based on specific requirements.

To wrap up, the training procedure of a neural ODE can be outlined by the following two steps:

1. **Forward Pass:** Solve the ODE from t_0 to t_1 to get $\mathbf{x}(t_1)$ and compute the loss $\mathcal{L}$.

2. **Backward Pass:** Construct the initial value for the augmented state at t_1: $\mathbf{s}(t_1) = [\mathbf{x}(t_1), \frac{\partial \mathcal{L}}{\partial \mathbf{x}(t_1)}, \mathbf{0}]^\top$. Use the ODE solver to integrate the augmented dynamics $\frac{d\mathbf{s}(t)}{dt} = [f(\mathbf{x}(t),\theta,t), -\left(\frac{\partial f}{\partial \mathbf{x}(t)}\right)^\top \mathbf{a}(t), -\left(\frac{\partial f}{\partial \theta}\right)^\top \mathbf{a}(t)]^\top$ backwards from t_1 to t_0.

1. An ODE is not inherently invertible in the simple function sense, but the concept applies to neural ODEs and flows (i.e., solutions over time) in machine learning. Informally, invertibility means reversing the transformation to find prior states or map between spaces.

3.3 Neural Dynamics and Flows

Neural ODEs provide a powerful tool for describing system dynamics. A key insight is that we can use a neural network $f(\cdot)$ to model a vector field. This defines a flow which represents as a continuous, invertible transformation that evolves the state space over time. In this subsection, we formalize these concepts to move from individual state trajectories to the evolution of entire probability distributions. This framework will serve as the foundation for generative models presented in the following sections.

3.3.1 Vector Fields

A vector field is a mathematical concept used to describe a space where every point has a specific magnitude and direction associated with it. To understand it intuitively, imagine the surface of a flowing river. At any given location and any specific moment, the water moves with a particular speed and direction. If we were to draw an arrow at every point in the river representing this local velocity, the resulting collection of arrows is a vector field. As another example that is more close to our previous discussion, imagine the state space is filled with a fluid. At every point in this space, the fluid flows with a specific velocity and direction. This assignment of a velocity vector to every point in space and time is known as a vector field.

Formally, a vector field on $\mathbb{R}^d$ is a map

$$f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d. \tag{16}$$

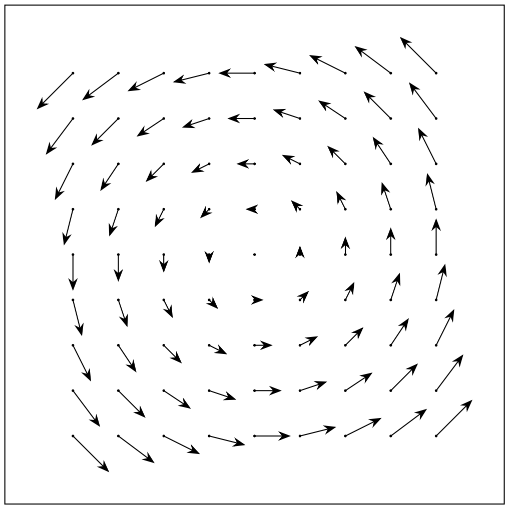
It assigns a velocity vector to each point $\mathbf{x}$ at time t. In the context of neural ODEs, we parameterize this field using a neural network $f(\mathbf{x}(t), \theta, t)$. Given a current state $\mathbf{x}(t)$ and time t, the network outputs the time derivative by using the ODE $\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), \theta, t)$, indicating "how fast and in what direction the data is moving right now". Note that the vector field here is time-dependent as the velocity at a point changes over time.

Figure 3 shows two example vector fields in a plane. The first is a vector field defined by $f(\mathbf{x}) = [-x_2, x_1]^\top$. A particle placed in this field will move perpendicular to its position vector, tracing out a circle. The second is a vector field defined by $f(\mathbf{x}) = [\sin(x_1) + \sin(x_2), \sin(x_1) - \sin(x_2)]^\top$. This creates a more complex flow pattern. In real-world applications, by using a deep neural network to parameterize $f(\cdot)$, we can model highly complex, non-linear, and time-varying dynamics.

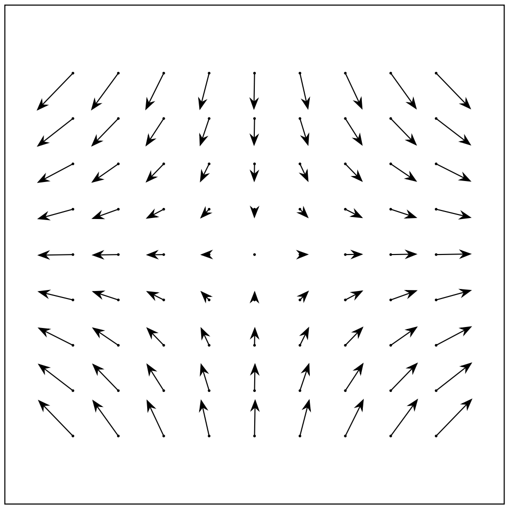
3.3.2 Flows and Diffeomorphisms

While a vector field describes the instantaneous velocity at any given point, a flow describes the long-term trajectory of points moving through that space. If we return to our river analogy, the vector field tells us how fast the water is moving at every specific coordinate right now. The flow, on the other hand, tells us where a leaf, dropped into the river at a certain position, will end up after a certain amount of time.

Mathematically, a flow map Φ is a function that tracks the evolution of all possible initial states over time. Formally, we define the flow as a mapping $\Phi : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$, which takes an initial point $\mathbf{x}(0)$ and a time parameter t to produce the state at time t. This is the solution to



(a) Vector field $f(\mathbf{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}$



(b) Vector field $f(\mathbf{x}) = \begin{bmatrix} \sin(x_1) + \sin(x_2) \\ \sin(x_1) - \sin(x_2) \end{bmatrix}$

Figure 3: Two examples of vector fields in the 2D plane. Each arrow represents the instantaneous velocity vector at that location. Subfigure (a) represents a linear system showing circular streamlines around the origin. Subfigure (b) represents a nonlinear system defined by sine functions, creating a periodic arrangement of fixed points and flow patterns.

the initial value problem

$$\frac{\partial \Phi(\mathbf{x}(0), t)}{\partial t} = f(\Phi(\mathbf{x}(0), t), \theta, t), \tag{17}$$

subject to the initial condition

$$\Phi(\mathbf{x}(0), 0) = \mathbf{x}(0). \tag{18}$$

To analyze how the entire space evolves, we often fix the time t and consider the time-t map, denoted as $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$, where $\Phi_t(\cdot) = \Phi(\cdot, t)$. While the vector field $f(\cdot)$ represents the velocity, the map Φ_t represents the resulting transformation of the entire space. As t increases, Φ_t pushes all points in $\mathbb{R}^d$ along the trajectories defined by the vector field $f(\cdot)$. It is important to note that the state $\mathbf{x}(t)$ is the evaluation of the flow at a specific initial point, i.e., $\mathbf{x}(t) = \Phi_t(\mathbf{x}(0))$. To further clarify the distinctions and relationships between these symbols, we summarize them in Table 1.

Importantly, for a flow to be well-defined and useful in generative modeling, it must be a **homeomorphism** and, more strictly, a **diffeomorphism**. A homeomorphism is a bijective (one-to-one and onto) map $g : \mathcal{X} \to \mathcal{Y}$ such that both $g(\cdot)$ and its inverse $g^{-1}(\cdot)$ are continuous. In intuitive terms, a homeomorphism is a rubber-sheet deformation. Imagine the state space is a sheet of flexible rubber. You can stretch it, shrink it, or twist it as much as you like, but you

Symbol	Name	Definition	Analogy	Interpretation
f	Vector Field	The derivative $\frac{d\mathbf{x}(t)}{dt}$	The instantaneous velocity of the water at a specific location and time.	The neural network itself. It defines how data evolves.
$\mathbf{x}(t)$	State/Trajectory	The value in $\mathbb{R}^d$ at time t.	The coordinates of a leaf at a particular moment t.	The hidden state at a certain time.
Φ	Flow Map	$\Phi : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$	A general function to find the end point given any start and any time.	The general solution of the ODE, i.e., the cumulative effect of the network over time.
Φ_t	Time-t Map	$\Phi_t(\cdot) = \Phi(\cdot, t)$ $(\mathbb{R}^d \to \mathbb{R}^d)$	A snapshot of the whole river: how the entire surface has shifted after t seconds.	The total transformation (diffeomorphism) from input to output.

Table 1: Concepts related to flows.

are not allowed to tear it (which would break continuity) or glue different parts together (which would break bijectivity). If one space can be transformed into another via a homeomorphism, the two spaces are considered topologically equivalent. From the perspective of neural ODEs, the flow $\Phi_t(\cdot)$ acts as a homeomorphism. This ensures that the global structure of the data is preserved: points that are close together in the initial distribution will remain relatively close in the transformed distribution, and no holes are created in the space.

In neural ODEs and flow-based models, we often need a stronger condition than just a homeomorphism. We need the transformation to be smooth so that we can perform calculus (e.g., compute gradients and use the change-of-variables formula). This leads us to the concept of a diffeomorphism.

A map $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$ is a diffeomorphism if:

- $\Phi_t(\cdot)$ is a homeomorphism (it is bijective, and both $\Phi_t(\cdot)$ and $\Phi_t^{-1}(\cdot)$ exist and are continuous).
- Both $\Phi_t(\cdot)$ and $\Phi_t^{-1}(\cdot)$ are continuously differentiable (C^1 or higher).

The diffeomorphic property of the flow is the foundation of many flow-based models. In a neural ODE, the vector field $f(\cdot)$ is typically parameterized by a neural network with smooth activation functions (e.g., Tanh), making $f(\cdot)$ continuously differentiable (C^1). While the Lipschitz continuity is sufficient to guarantee the uniqueness of trajectories (ensuring Φ_t is a homeomorphism) according to the Picard–Lindelöf theorem, the C^1 smoothness of the network further ensures that the flow $\Phi_t(\cdot)$ is a diffeomorphism[2].

2. This property is also called smooth dependence on initial conditions, which is a natural extension of the Picard–Lindelöf theorem to dynamical systems and manifolds [Coddington, 1955].

This property has advantages in many applications, especially in generative models. First, because $\Phi_t(\cdot)$ is a diffeomorphism, the transformation is fully reversible. If we know the state of the system at time t_1, we can integrate the same ODE backward in time to recover the exact initial state at t_0. This is the foundation of exact density estimation and memory-efficient training. Second, a flow-based transformation cannot change the fundamental topology of the input space. For example, a flow cannot transform a single connected component into two separate islands, nor can it tie a knot in a string that was not already knotted. This path-preserving nature ensures that the neural network learns a continuous deformation of the probability mass, rather than a chaotic reordering of the points. Further details on flows can be found in related papers, such as [Lipman et al., 2024].

3.3.3 Continuous Normalizing Flows

While a standard flow tracks the movement of individual points, continuous normalizing flows (CNFs) extend this framework to the evolution of entire probability distributions.

Imagine that, instead of a single leaf, we release a cloud of colored dye into the river. As the water flows, the dye cloud deforms, stretches, and compresses to follow the complex currents. While the shape of the cloud changes, the total mass of the dye remains constant. CNFs define an invertible mapping between a simple distribution (like a Gaussian) and the complex data distribution. In this context, the term *normalizing* refers to the process of simplifying (or normalizing) complex data back into the base distribution.

To formalize the evolution of probability distributions, we must define how the flow $\Phi_t(\cdot)$, which works on points in the state space, induces a transformation on probability measures. Let $p_t(\mathbf{x})$ denote the probability density function of the state at time t. The evolution of this density is governed by the conservation of probability mass: as the space expands or contracts under the flow, the local density must adjust inversely to preserve the total mass.

Consider an arbitrary region $\mathcal{S}_0$ in the initial space at time $t = 0$. Under the flow $\Phi_t(\cdot)$, this region evolves into a new region $\mathcal{S}_t = \Phi_t(\mathcal{S}_0)$ at time t. Since probability mass is neither created nor destroyed, the total probability contained within the region must remain invariant

$$\int_{\mathcal{S}_t} p_t(\mathbf{x}) d\mathbf{x} \quad = \quad \int_{\mathcal{S}_0} p_0(\mathbf{z}) d\mathbf{z}, \tag{19}$$

where $\mathbf{x}$ and $\mathbf{z}$ represent a sample in the spaces at time t and 0, respectively. To relate these integrals, we apply the multivariate change of variables theorem to the left-hand side. We substitute the integration variable $\mathbf{x}$ with its pre-image $\mathbf{z}$ using the mapping $\mathbf{x} = \Phi_t(\mathbf{z})$. This substitution introduces the absolute value of the Jacobian determinant to account for the change in volume element

$$\int_{\mathcal{S}_0} p_t(\Phi_t(\mathbf{z})) \left| \det \frac{\partial \Phi_t(\mathbf{z})}{\partial \mathbf{z}} \right| d\mathbf{z} \quad = \quad \int_{\mathcal{S}_0} p_0(\mathbf{z}) d\mathbf{z}. \tag{20}$$

Geometrically, the Jacobian determinant term $\left|\det \frac{\partial \Phi_t(\mathbf{z})}{\partial \mathbf{z}}\right|$ can be seen as a volume expansion factor[3]. It quantifies the ratio of the infinitesimal volume in the target space to the volume in the source space. The presence of this term ensures that the integration measure is correctly scaled to reflect the distortion of the coordinate system caused by the flow.

Since Eq. (20) holds for any arbitrary region $\mathcal{S}_0$, the integrands must be equal pointwise. By setting $\mathbf{z} = \mathbf{x}(0)$ and $\Phi_t(\mathbf{z}) = \mathbf{x}(t)$, we obtain the explicit relationship between the densities

$$p_t(\mathbf{x}(t)) \left|\det \frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)}\right| \quad = \quad p_0(\mathbf{x}(0)). \tag{21}$$

Rearranging this term yields the standard formula for the density transformation under a diffeomorphism

$$p_t(\mathbf{x}(t)) \quad = \quad p_0(\mathbf{x}(0)) \left|\det \frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)}\right|^{-1}. \tag{22}$$

Here, $\frac{\partial \mathbf{x}(t)}{\partial \mathbf{x}(0)}$ is the Jacobian matrix of the flow map, which represents the accumulation of local deformations from time 0 to t. See Figure 4 for an illustration of probability density transformation in a CNF.

Taking the logarithm of Eq. (22) gives the change in log-density

$$\log p_t(\mathbf{x}(t)) \quad = \quad \log p_0(\mathbf{x}(0)) - \log \left|\det \frac{\partial \Phi_t(\mathbf{x}(0))}{\partial \mathbf{x}(0)}\right|. \tag{23}$$

This is also called the push-forward density formula under the push-forward measure [Tao, 2011].

In practice, computing the determinant of the Jacobian for the entire flow is computationally expensive. We can simplify this by considering the infinitesimal change. Following Chen et al. [2018]'s work, by differentiating the log-density with respect to time, we arrive at the **instantaneous change of variables theorem**

$$\frac{d \log p_t(\mathbf{x}(t))}{dt} \quad = \quad -\mathrm{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right). \tag{24}$$

Here, $\mathrm{Tr}(\frac{\partial f}{\partial \mathbf{x}})$ is the trace of the Jacobian of the vector field, which is equivalent to the divergence, denoted as $\nabla \cdot f$. This quantity describes the instantaneous rate of expansion or contraction of an infinitesimal volume. If the divergence is positive, the volume expands, and the log-density decreases at a rate equal to the divergence. Conversely, if the divergence is negative, the volume contracts and the density increases.

This continuous formulation offers a computational advantage over discrete normalizing flows. In discrete flows, ensuring the tractability of the Jacobian determinant often requires restricting the neural network architecture (e.g., to triangular matrices) [Hoogeboom et al., 2019; Tran et al., 2019]. In contrast, CNFs require only the trace of the Jacobian. This allows $f(\cdot)$ to be parameterized by arbitrary deep neural networks, as the trace can be efficiently approximated

3. For instance, if the determinant is 2, an infinitesimal volume at $\mathbf{z}$ is stretched to become twice as large at $\mathbf{x}$.

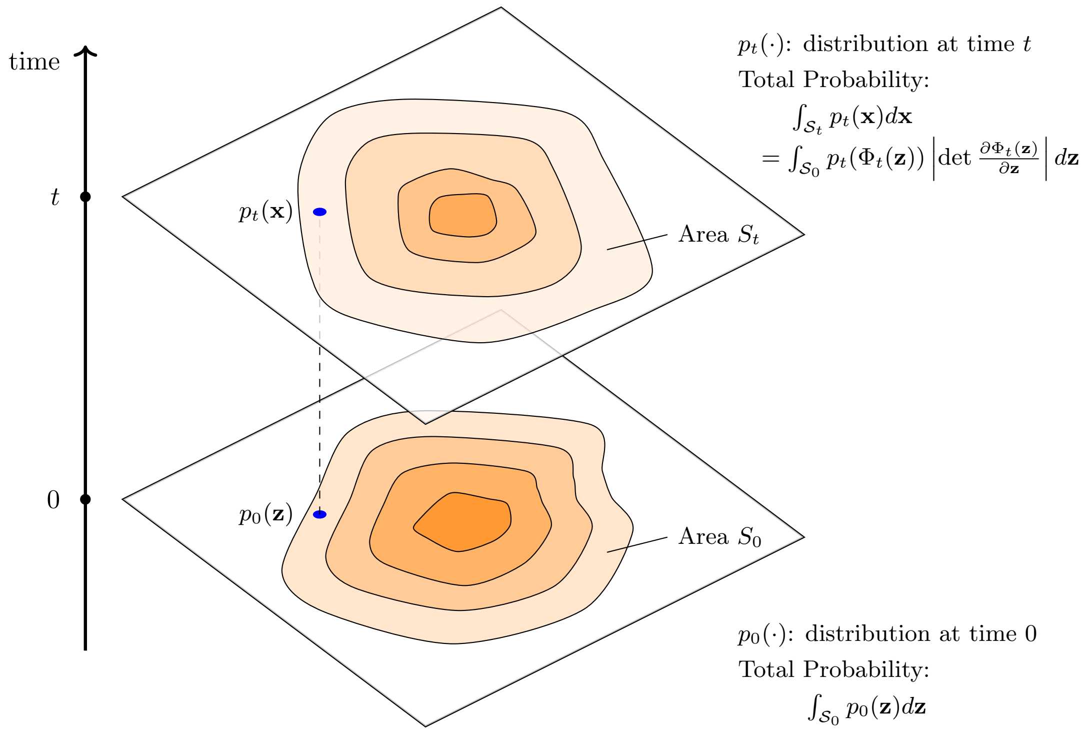

time
t
0
$p_t(\cdot)$: distribution at time t
Total Probability:
$\int_{\mathcal{S}_t} p_t(\mathbf{x}) d\mathbf{x}$
$= \int_{\mathcal{S}_0} p_t(\Phi_t(\mathbf{z})) \left|\det \frac{\partial \Phi_t(\mathbf{z})}{\partial \mathbf{z}}\right| d\mathbf{z}$
$p_t(\mathbf{x})$
Area S_t
$p_0(\mathbf{z})$
Area S_0
$p_0(\cdot)$: distribution at time 0
Total Probability:
$\int_{\mathcal{S}_0} p_0(\mathbf{z}) d\mathbf{z}$

Figure 4: Illustration of probability density transformation in a CNF. The diagram visualizes the continuous evolution of a probability distribution over time. A point $\mathbf{z}$ in the base distribution is mapped to $\mathbf{x} = \Phi_t(\mathbf{z})$ at time t. The transformation of the density function $p_t(\mathbf{x})$ is governed by the conservation of probability: although the geometry of the distribution (represented by the contour maps) stretches and compresses, the integral of the density over the corresponding areas remains constant.

using stochastic estimators (such as the Hutchinson trace estimator), compared to the $O(d^3)$ cost of a full determinant.

At inference time, we can integrate the joint dynamics of the state and the log-density. For example, in generative modeling, given a target data point $\mathbf{x}_{\text{data}}$ at time t_1, we can compute its log-likelihood by integrating backward to from t_1 to 0

$$\begin{aligned} \log p_{t_1}(\mathbf{x}_{\text{data}}) &= \log p_0(\mathbf{x}(0)) + \int_{t_1}^{0} \text{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right) dt \\ &= \log p_0(\mathbf{x}(0)) - \int_{0}^{t_1} \text{Tr}\left(\frac{\partial f(\mathbf{x}(t), \theta, t)}{\partial \mathbf{x}(t)}\right) dt. \end{aligned} \tag{25}$$

This provides a framework where the neural network learns a vector field that smoothly deforms a simple prior distribution into the complex data distribution. This continuous-time perspective on density evolution is at the core of advanced generative models like diffusion models. In the next section, we will explore how these models are applied to transform noise back into high-fidelity data.

3.3.4 Variants of Flows

While CNFs define the transformation via the integration of a vector field, the earlier and broader family of flow-based models focuses on constructing discrete sequences of invertible mappings. The primary challenge in these models is designing the neural networks such that the Jacobian determinant is computationally efficient (ideally linear in dimensionality, $O(d)$) while retaining high expressivity. Based on how they structure the transformation and the Jacobian matrix, these models can be categorized into several variants:

- **Coupling Flows**. Pioneered by NICE [Dinh et al., 2014] and RealNVP [Dinh et al., 2017], these models partition the input dimensions into two subsets. One subset remains unchanged, while the other undergoes an affine transformation parameterized by the first subset. This construction enforces a triangular Jacobian matrix, making the determinant calculation trivial (the product of diagonal terms) and computationally efficient.
- **Autoregressive Flows**. These models treat the input dimensions as a sequence, where the transformation of the i-th dimension depends only on the previous dimensions $\{1, \ldots, i-1\}$. This structure also yields a triangular Jacobian. Notable examples include masked autoregressive flow (MAF) [Papamakarios et al., 2017], which is efficient for density estimation, and inverse autoregressive flow (IAF) [Kingma et al., 2016], which is efficient for sampling.
- **Invertible Linear Transformations**. To ensure all dimensions can interact with each other, models like Glow [Kingma and Dhariwal, 2018] introduce learnable 1×1 convolutions. To maintain efficiency, the weight matrix is often parameterized via LU decomposition, reducing the complexity of the determinant calculation from cubic $O(d^3)$ to linear $O(d)$.
- **Residual Flows**. Inspired by residual networks, these flows define the mapping as $\mathbf{y} = \mathbf{x} + g(\mathbf{x})$. To guarantee invertibility, the Lipschitz constant of $g(\cdot)$ must be strictly bounded (typically < 1) [Behrmann et al., 2019]. Unlike coupling layers, residual flows do not have a triangular Jacobian. Instead, they often rely on stochastic estimators (such as the Russian Roulette estimator) to approximate the infinite series expansion of the log-determinant [Chen et al., 2019].

It should be noted that flow is a very general concept and has been extensively discussed in many fields, particularly in the study of dynamical systems and differential geometry. Mathematically, a flow can be formalized as a **one-parameter group of diffeomorphisms**. This definition implies that the mapping $\Phi_t(\cdot)$ satisfies the group law

$$\begin{aligned} \Phi_0(\mathbf{x}) &= \mathbf{x}, && (26) \\ \Phi_{t+s}(\mathbf{x}) &= \Phi_t(\Phi_s(\mathbf{x})). && (27) \end{aligned}$$

This algebraic structure provides the theoretical guarantee for invertibility: the inverse of a transformation over time t is simply the transformation over time $-t$, i.e., $\Phi_t^{-1}(\cdot) = \Phi_{-t}(\cdot)$.

Furthermore, from the perspective of fluid mechanics and statistical physics, the flow of a probability density is governed by the **continuity equation** [Batchelor, 2000], a PDE that describes the conservation of mass

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla \cdot (p_t(\mathbf{x}) f(\mathbf{x}, \theta, t)) \;=\; 0. \tag{28}$$

This PDE states that the rate of change of density at a point is exactly balanced by the divergence of the probability flux. The instantaneous change of variables formula used in CNFs (Eq. (24)) is, in fact, the solution to this continuity equation along the trajectories of the particles. Thus, flow-based models can be viewed not merely as neural networks with specific architectural constraints, but as computational implementations of classical transport phenomena. See Appendix B for a more detailed derivation of the instantaneous change of variables formula from the continuity equation.

Appendix B. Deriving CNFs from the Continuity Equation

Here we provide a derivation showing that the instantaneous change of variables formula used in CNFs is a direct consequence of the continuity equation governing probability conservation.

Let $p_t(\mathbf{x})$ denote the probability density function at time t, and let $f(\mathbf{x}, \theta, t)$ denote the continuous vector field generated by the neural network. Note that, for notational brevity, here we use $\mathbf{x}$ to denote the time-dependent trajectory $\mathbf{x}(t)$. The conservation of probability mass is described by the continuity equation (Eulerian view)

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla \cdot (p_t(\mathbf{x}) f(\mathbf{x}, \theta, t)) \;=\; 0, \tag{29}$$

where $\nabla \cdot$ denotes the divergence operator with respect to $\mathbf{x}$.

Using the vector calculus identity $\nabla \cdot (\phi \mathbf{A}) = (\nabla \phi) \cdot \mathbf{A} + \phi(\nabla \cdot \mathbf{A})$, we expand the divergence term, as follows

$$\frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot f(\mathbf{x}, \theta, t) + p_t(\mathbf{x}) \nabla \cdot f(\mathbf{x}, \theta, t) \;=\; 0. \tag{30}$$

While Eq. (30) describes the density evolution at a fixed point in space, CNFs operate by tracking the density along the trajectory of a moving particle (Lagrangian view). The trajectory is defined by the ODE $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}(t), \theta, t)$.

To bridge these two perspectives, we compute the total time derivative (also known as the material derivative) of the probability density along this trajectory. First, applying the multivariate chain rule to $p_t(\mathbf{x})$, we obtain the general form

$$\frac{dp_t(\mathbf{x})}{dt} \;=\; \frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot \frac{d\mathbf{x}}{dt}. \tag{31}$$

Here, from the perspective of fluid mechanics, the partial derivative $\frac{\partial p_t}{\partial t}$ can be seen as the local rate of change at a fixed spatial location, while the total derivative $\frac{dp_t}{dt}$ can be seen as the total change experienced by the particle as it moves along its trajectory.

Next, by substituting the specific ODE of the flow $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}, \theta, t)$ into Eq. (31), we obtain the material derivative in terms of the vector field f

$$\frac{dp_t(\mathbf{x})}{dt} = \frac{\partial p_t(\mathbf{x})}{\partial t} + \nabla p_t(\mathbf{x}) \cdot f(\mathbf{x}, \theta, t). \tag{32}$$

Now, by substituting the expression for the material derivative from Eq. (32) into the expanded continuity Eq. (30), we can identify and replace the first two terms

$$\underbrace{\frac{\partial p_t}{\partial t} + \nabla p_t \cdot f}_{\frac{dp_t}{dt}} + p_t(\nabla \cdot f) = 0, \tag{33}$$

where arguments for p_t and f are omitted for clarity. This simplifies the relation to

$$\frac{dp_t(\mathbf{x})}{dt} = -p_t(\mathbf{x})\nabla \cdot f(\mathbf{x}, \theta, t). \tag{34}$$

To derive the standard CNF formulation, we consider the log-density $\log p_t(\mathbf{x})$. Using the derivative rule $\frac{d}{dt}\log u(t) = \frac{1}{u(t)}\frac{du(t)}{dt}$, we divide both sides by $p_t(\mathbf{x})$, and obtain

$$\frac{d\log p_t(\mathbf{x})}{dt} = -\nabla \cdot f(\mathbf{x}, \theta, t). \tag{35}$$

Finally, recognizing that the divergence of the vector field is the trace of its Jacobian matrix, i.e., $\nabla \cdot f = \mathrm{Tr}\left(\frac{\partial f}{\partial \mathbf{x}}\right)$, we arrive at the instantaneous change of variables formula (i.e., Eq. (24))

$$\frac{d\log p_t(\mathbf{x})}{dt} = -\mathrm{Tr}\left(\frac{\partial f(\mathbf{x}, \theta, t)}{\partial \mathbf{x}}\right). \tag{36}$$

References

George Keith Batchelor. An introduction to fluid dynamics. Cambridge university press, 2000.

Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In International conference on machine learning, pages 573–582. PMLR, 2019.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. Advances in neural information processing systems, 31, 2018.

Ricky TQ Chen, Jens Behrmann, David K Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. Advances in neural information processing systems, 32, 2019.

EA Coddington. Theory of ordinary differential equations. McGraw-Hill Book Company, 1955.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. arXiv preprint arXiv:1410.8516, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. In International Conference on Learning Representations, 2017.

Emiel Hoogeboom, Jorn Peters, Rianne Van Den Berg, and Max Welling. Integer discrete flows and lossless compression. Advances in Neural Information Processing Systems, 32, 2019.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. Advances in neural information processing systems, 31, 2018.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. Advances in neural information processing systems, 29, 2016.

Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky TQ Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. Flow matching guide and code. arXiv preprint arXiv:2412.06264, 2024.

George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. Advances in neural information processing systems, 30, 2017.

Terence Tao. An introduction to measure theory, volume 126. American Mathematical Society, 2011.

Dustin Tran, Keyon Vafa, Kumar Agrawal, Laurent Dinh, and Ben Poole. Discrete flows: Invertible generative models of discrete data. Advances in Neural Information Processing Systems, 32, 2019.