Going Backwards through Chaotic Swarms and Biological Protocols

Matthew Shang

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Going Backwards through Chaotic Swarms and Biological Protocols

Matthew Shang

Solving an *inverse problem*, the problem of recovering underlying parameters or initial conditions from observed outcomes, enables computational models to be updated based on real-world data. Given a forward simulation that moves from known parameters to outcomes through a complex process, finding an efficient method for inverse reasoning is challenging. This thesis advances inverse methods specifically for computational models of swarming behavior and biological experiments.

Swarming models simulate synchronized flocking and schooling behaviors observed in nature. These simulations, widely used by graphics programmers for visual effects and by biologists to study emergent behaviors, commonly use the *boids* algorithm. Boids prescribe simple local rules for individual agents without global coordination. Solving the inverse problem for boids, which means recovering parameters from observed swarm behaviors, would enable appearance-driven control and data-driven modeling. Derivative-based methods encounter immediate challenges due to discontinuities in traditional implementations, which we smooth with an energy potential model. However, a more fundamental challenge is that boids exhibit chaotic behavior, magnifying small perturbations exponentially over time. This exponential growth causes derivatives computed via automatic differentation to become numerically unstable. We explore the theory behind derivatives in chaotic systems and discuss practical advancements toward inverse methods in the presence of chaos.

Biological experiments increasingly leverage formalized protocols, represented as programs, to enhance reproducibility by standardizing common procedures. However, existing

protocol languages inadequately represent error sources and error handling. When protocols inevitably fail, debugging relies on human expertise, a process that is challenging for unfamiliar protocols and constrains scalability in automated facilities. A prerequisite for inverse reasoning through an experimental protocol is comprehensive modeling of all potential outcomes, including failures. To address this, we introduce Oops, a novel programming language for formalizing biological protocols with their potential errors. Oops concisely models errors using probabilistic semantics, an extensible laboratory abstraction, and metaprogramming techniques. Given an observed experimental outcome, Oops enables probabilistic inference to identify errors by correlating lab observations with protocol conditions. We formalize a collection of molecular cloning protocols and present case studies demonstrating how Oops can explain errors and assess diagnostic capabilities.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thank you to Gilbert for the countless discussions that drove the research in this thesis, Craig for creating the joyful yet mathematically rich topic of boids, and Eric for patiently teaching computer scientists about biology. I would also like to thank the PLSE and GRAIL communities at UW for being wonderful environments for sharing and learning about research, and for hanging out in general. Finally, this thesis would not be possible without the unwavering support of my friends and family.

Part I

TOWARDS GRADIENT-BASED INVERSE DESIGN OF ARTIFICIAL SWARMING

Chapter 1
AN INTRODUCTION TO SWARMING AND BOIDS

One of the most awe-inspiring phenomena in nature is the self-organization of thousands of birds or fish into swarms, seemingly without global communication or central control[1]. To replicate such behavior for computer-generated animations, Craig Reynolds introduced the *boids*[2] algorithm in 1987 [29]. Boids follow three simple behaviors: *alignment*, *cohesion*, and *avoidance*. Even though each agent acts based only on these local rules, a remarkable global swarming behavior emerges.

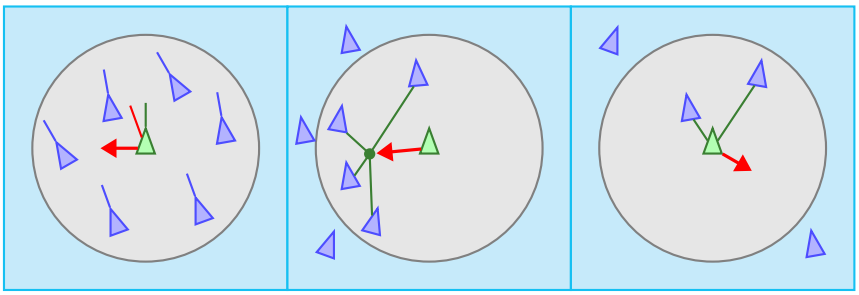

Figure 1.1: Boids rules visualized. (Left) Boids **align** themselves with their neighbors, (center) **cohere** by moving toward the center of their neighbors, and (right) **avoid** collisions[3].

Alignment encourages a boid to steer in the direction of its neighbors' average heading. Cohesion guides a boid towards the center of mass of neighboring boids. Avoidance ensures a boid steers clear of potential collisions with other boids or obstacles. Each behavior has an associated radius of interaction, defining a boid's local neighborhood. These behaviors

[1] As far as we know.

[2] A *boid* is an individual agent in the swarm.

[3] This figure is adapted from figures originally by Craig [30].

are biologically plausible. Alignment and cohesion reflect the drive to remain close together, while avoidance represents self-preservation. For a mathematically rigorous description of the model, see Chapter 4.

The basic boids model allows numerous nature-inspired extensions. For example, boids typically have a limited field of vision, reflecting real-world constraints on animal sight. Another common modification introduces a distance-based reduction in the influence of distant neighbors, since intuitively one focuses on closer neighbors [4].

The result is a powerful swarming model capable of expressing diverse collective visual appearances. Boids have been effectively applied in contexts ranging from bird animations in the video game *Half-Life*, drone swarms in robotics, and penguin armies in the movie *Batman Returns*. Examples of boid trajectories under various behavioral rules appear in Figure 1.2.

The expressive power of boids comes with a cost, which is a plethora of parameters. These include interaction radii, distance-based decay rates, rule weights, and field of view. Since parameters influence only the local interactions, their global effects on swarm appearance are often unclear. As a result, finding the right parameters to achieve a desired look is a high-dimensional trial-and-error process.

To avoid this cumbersome trial-and-error process, this thesis proposes an inverse approach to boid parameter optimization. Instead of matching exact boid positions and velocities, we aim to control simulation statistics—quantities computed from instantaneous snapshots and averaged over time. These quantities include properties such as the average separation between boids, which captures part of the visual appearance of a swarm we might want to adjust. With sufficiently expressive statistics, simulations can even match data from real-world swarms, such as the starling murmurations studied and quantified by the STARFLAG project [10].

In Chapter 2, we establish the mathematical framework for solving this inverse problem.

---

[4] One study [10] suggests that starlings, known for flocking in mumurations of thousands, only consider their seven closest neighbors.

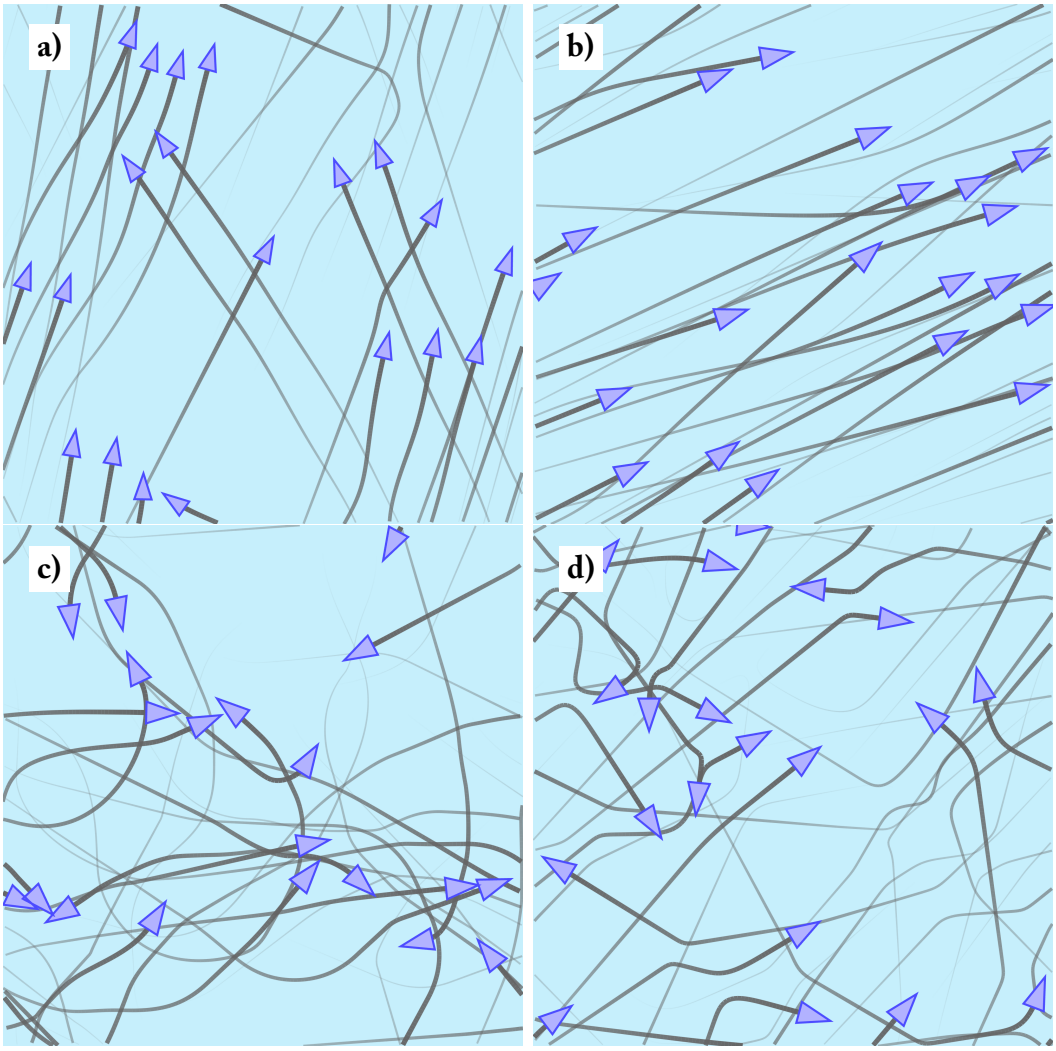

Figure 1.2: Visual examples of boid motion. **a)** A combination of all three behaviors creates subflocks moving in separate directions. **b)** Alignment only makes a globally aligned swarm. **c)** Cohesion only creates snaking, intertwining paths. **d)** Avoidance only results in angular paths with sudden turns.

We view boids as a *dynamical system*, characterized by a state evolving over time according to an ordinary differential equation (ODE). This reduces the inverse problem into finding the derivative of an objective function based on system trajectory. However, due to *chaos*, traditional automatic differentiation methods fail in practice. Despite the derivative existing in theory, in practice, small perturbations in chaotic systems expand exponentially, causing derivative estimates to explode.

A recent line of work, known as space-split sensitivity [34], proposes an algorithm for computing derivatives in chaotic systems. Chapter 3 provides an exposition of chaotic dynamics theory and this new approach. Along the way, we visualize and explain the remarkable geometric structures contained in chaotic attractors.

In Chapter 4, we recast the classic boids model as a smooth ODE. Motivated by the desire for an ODE as well as system smoothness, we develop and visualize a swarming model based on *energy functions*. Steering to minimize these energies naturally induces swarming motion. Chapter 5 applies our theoretical and modeling advances. We first compute the *unstable dimension*, a geometric quantity indicating the number of directions in which trajectories diverge. This is a critical input to the space-split algorithm. Finally, we summarize our progress towards practical inverse solutions.

As an additional insight, Chapter 6 discusses *cotangent dynamics*, describing the evolution of *covectors* in dynamical systems. We demonstrate that cotangent dynamics can accelerate standard algorithms in computational dynamics and propose that they may also connect deeply with split-space sensitivity analysis.

Chapter 2

CHAOS IN SWARMS

2.1 Mathematical Definitions

We begin by establishing a mathematical vocabulary for the rest of the chapters. The boids system can be abstracted as a *dynamical system*, which has:

- A state space $\mathcal{M}$. This is also called *phase space*. For example, for n boids living in two dimensions, the state space is $\mathbb{R}^{2n} \times \mathrm{SO}(2)^n$, where each boid has two components representing position and one component representing heading.

- A vector field F which drives the dynamics of the system according to the ordinary differential equation (ODE) $\frac{\mathrm{d}u}{\mathrm{d}t} = F(u;\theta)$. The dynamics are parameterized by p parameters, although we only consider $p = 1$ in this thesis. Usually, we will refer to this parameter as θ. Assume that F is sufficiently smooth when necessary. For the full definition of F in the boids system, see Chapter 4.

- An initial condition $u_0 \in \mathcal{M}$ and a trajectory of states $u_t \in \mathcal{M}$ for integers $t > 0$, defined by $u_t = f(u_{t-1};\theta)$. The discrete map f steps a state forward by integrating the vector field F for a fixed timestep h.

Other than the boids, we also have an instantaneous objective function S, which takes a single state in $\mathcal{M}$ and produces a real-valued score. For example, S could reward separation between boids by returning the minimum distance between all pairs of boids. The combination of a trajectory from the dynamical system and the instantaneous objective function leads to the full objective we wish to set θ to maximize.

Definition 2.1.1. Define the objective function to be the *infinite-time average*

$$\overline{S} = \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} S(u_t).$$

The optimization problem is to find $\arg\max_\theta \overline{S}$.

One approach to this optimization problem is to repeatedly move θ in the direction of $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$. The crucial subproblem is to find this derivative.

2.2 Ergodic Time Averages

First, let's step back and motivate this definition of the objective. Why involve infinite time? Intuitively, the value of the limit in Definition 2.1.1 should depend on the initial condition u_0. Surprisingly, this is not the case if the dynamical system is *ergodic*. An ergodic system is one that mixes thoroughly, so the state does not become trapped in a subset of phase space. Having an ergodic system enables a probabilistic view on states in which there is a distribution over the phase space of how likely the system is to be at every particular state. Let μ be a probability measure, meaning that $\mu(\mathcal{M}) = 1$. Formally:

Definition 2.2.1. Let B be a subset of $\mathcal{M}$[1]. Suppose that if $f^{-1}(B) = B$, then either $\mu(B) = 0$ or $\mu(B) = 1$. Then μ is *ergodic*. Equivalently, μ is ergodic if and only if for all B, if $\mu(B) > 0$ then

$$\mu\left(\bigcup_{n=1}^{\infty} f^{-n}(B)\right) = 1.$$

In words, a state that has been evolved for some amount of time could have originated from almost anywhere in phase space.

Definition 2.2.2. A measure μ is *f-invariant*, or just invariant, if $\mu(f^{-1}(B)) = \mu(B)$ for all B.

---

[1]Technically, it should be part of the σ-algebra.

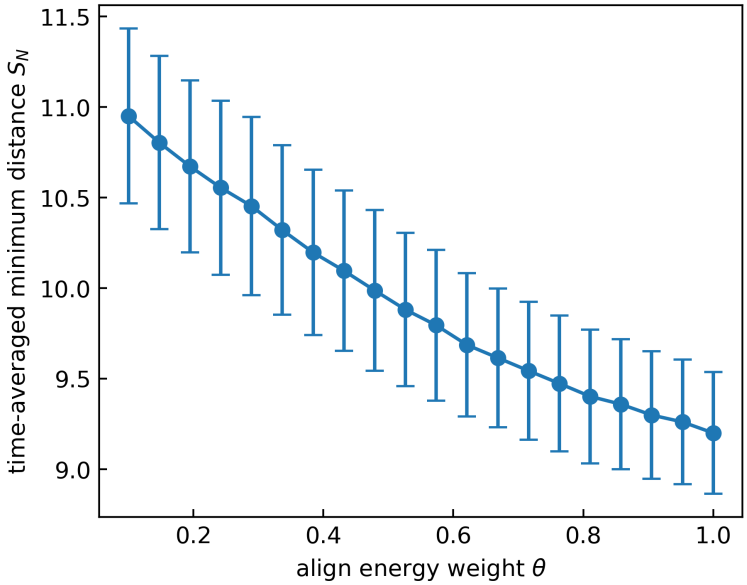

Figure 2.1: Time-averaged minimum distance responds to alignment weight. If the system is ergodic, then samples of $(1/N)\sum_{t=1}^{N} S(u_t)$ starting from random initial conditions for a sufficiently large N should converge to $\int S \, \mathrm{d}\mu$. For this plot, S measures the minimum distance separating pairs of boids and θ is alignment weight. The data comes from 2048 random initial conditions time-averaged over $N = 12000$ steps (100 seconds).

Theorem 1 (Birkhoff's Ergodic Theorem)**.** Let μ be an invariant ergodic measure. Suppose g is L^1-integrable. Then

$$\frac{1}{N}\sum_{t=1}^{N} g(u_t) \to \int_{\mathcal{M}} g \, \mathrm{d}\mu$$

for μ-almost every initial condition u_0.

Returning to our objective, the Ergodic Theorem says that if the system has an invariant ergodic measure μ, then $\overline{S}$ not only exists, but is also the same for almost all initial conditions. Thus, it characterizes the average behavior of boids according to S with a specific parameter θ. Figure 2.1 demonstrates empirically that $\overline{S}$ exists. Importantly, there appears to be smooth dependence of $\overline{S}$ on θ. The slope of the graph is our desired $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$.

2.3 Exponential Growth in Derivatives of Time Averages

Automatic differentiation appears well-suited for estimating $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$. Define the finite-time average S_N to be $\frac{1}{N}\sum_{t=1}^{N} S(u_t)$, the score over a trajectory of finite length. We assume that $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$ converges to $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$ as N goes to infinity, although there are technical conditions under which the limit and the derivative formally commute. By ergodicity, $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$ from a particular initial condition is an estimator for $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$ for sufficiently large N.

We test automatic differentiation on S_N for two different S: the average x coordinate of all boids and the minimum pairwise distance. Figure 2.2 shows the results. The main observation is that the variance of the estimates blows up rapidly with time. In fact, the growth is exponential. By the Central Limit Theorem, extending the integration time by a second would require an exponential increase in the number of samples, rendering this naive estimator intractable.

2.4 Tangent Dynamics

To understand why the norm of $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$ blows up exponentially, we first need to understand *tangent dynamics*. A tangent vector is an infinitesimal perturbation. The space of all such perturbations to $x \in \mathcal{M}$ is called the *tangent space* at x and is denoted by $T_x\mathcal{M}$. To see how tangent vectors evolve in a dynamical system, suppose the initial condition u_0 is perturbed by ϵv_0 for a miniscule ϵ, where v_0 is the initial perturbation direction (see Figure 2.3). Define v_t to be the resulting perturbation at step t, so $\epsilon v_t = f^t(u_0 + \epsilon v_0) - u_t$. By a first-order expansion,

$$u_{t+1} + \epsilon v_{t+1} = f(u_t + \epsilon v_t) \approx f(u_t) + \epsilon J_t v_t,$$

where J_t is the Jacobian of the system evaluated at u_t. By canceling $u_{t+1} = f(u_t)$ from both sides and dividing through by ϵ, we get that $v_{t+1} = J_t v_t$. This is known as the *homogeneous tangent equation*.

Now suppose that the parameter to f is also perturbed by ϵ. By a first-order expansion

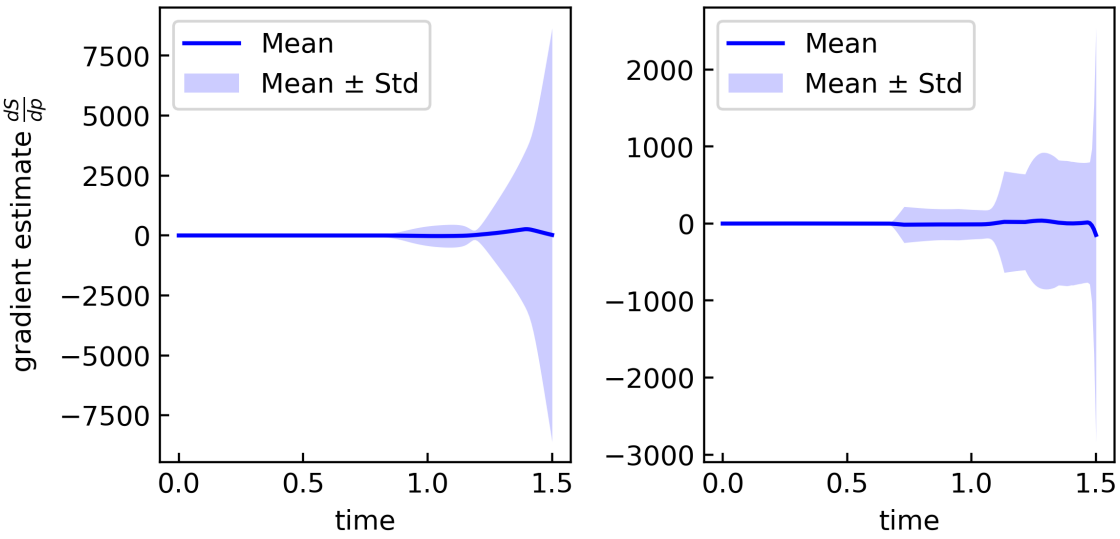

Figure 2.2: Naive gradient estimates blow up exponentially with time. On the left, S is the average x coordinate of all boids. On the right, S is the minimum distance between all pairs of boids. Both plots are obtained from 256 independent runs initialized uniformly at random and integrated for 180 steps with a time step of $1/120$.

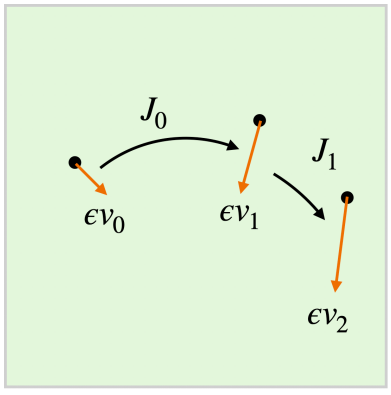

Figure 2.3: Infinitesimal perturbations.

again,

$$u_{t+1} + \epsilon v_{t+1} = f(u_t + \epsilon v_t; \theta + \epsilon) \approx f(u_t) + \epsilon J_t v_t + \epsilon \frac{\partial f}{\partial \theta}(u_t).$$

Letting $f_\theta = \frac{\partial f}{\partial \theta}$, we get that

$$v_{t+1} = J_t v_t + f_\theta(u_t). \tag{2.1}$$

This is called the *inhomogeneous tangent equation*. When $v_0 = 0$, the meaning of v_t is actually $\frac{\partial u_t}{\partial \theta}$, or the change in state at step t due to a change in the parameter θ. To see this, note that $\frac{\partial u_0}{\partial \theta} = 0$ since the initial condition is independent of the parameter, and then verify that $\frac{\partial u_t}{\partial \theta}$ satisfies Equation 2.1:

$$\frac{\partial u_{t+1}}{\partial \theta} = \frac{\partial}{\partial \theta} f(u_t) = J(u_t) \frac{\partial u_t}{\partial \theta} + f_\theta(u_t).$$

Finally, let's connect the inhomogeneous tangent equation back to the derivative of $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$. By carrying the derivative through the definition of S_N, we get

$$\frac{\mathrm{d}S_N}{\mathrm{d}\theta} = \frac{1}{N} \sum_{t=1}^{N} \frac{\partial}{\partial \theta} S(u_t) = \frac{1}{N} \sum_{t=1}^{N} \left\langle \nabla S(u_t), \frac{\partial u_t}{\partial \theta} \right\rangle. \tag{2.2}$$

Hence, the derivative is a sum of inner products between the gradient of S and the solution to the inhomogeneous tangent equation.

2.5 Chaotic Dynamics

Equation 2.2 tells us that the growth in magnitude of $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$ with N is roughly governed by the growth of $\left\| \frac{\partial u_t}{\partial \theta} \right\|$, assuming that ∇S is bounded. In a *chaotic system*, solutions to the inhomogeneous and homogeneous tangent equations grow exponentially in norm with time. This means intuitively that two states which start close together rapidly diverge. We formalize this idea in terms of tangent dynamics since a separation may be bounded in phase space while tangent vectors can grow unbounded.

The amount of chaos in a dynamical system can be quantified by measuring how fast the solution to the homogeneous tangent equation grows. If it grows according to $e^{\lambda t}$ for some $\lambda > 0$, then we say that the system is chaotic. The rate of growth λ is called the *Lyapunov exponent*.

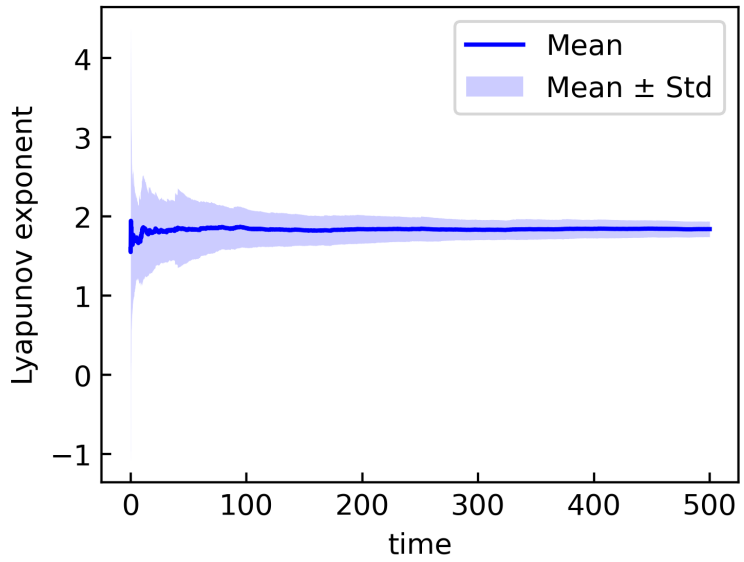

Figure 2.4: The maximal Lyapunov exponent of boids is positive. Boids are chaotic. The plot shows the running estimate over time; the final estimate is about 1.84. The distribution is over 64 runs.

Definition 2.5.1. The *maximal Lyapunov exponent* at u_0 in direction v_0 is given by

$$\lambda(u_0, v_0) = \lim_{N \to \infty} \frac{1}{N} \ln \frac{\|v_N\|}{\|v_0\|}$$

where $v_{t+1} = J(u_t)v_t$ for all $t > 0$.

In an ergodic system, $\lambda(u_0, v_0)$ is the same for almost every u_0 and v_0. We denote this constant by λ and say that λ is the maximal Lyapunov exponent of the system. Later, we will explore how varying v_0 in precise ways results in an entire spectrum of Lyapunov exponents.

Returning to swarming, Figure 2.4 computes λ for a boids system. It shows how λ converges to around 2. Boids are chaotic! Visually, a swarm of boids is greatly impacted by just a single boid veering into the crowd and causing it to change course. The diminishing standard deviation as time increases demonstrates convergence of the limit and the tight

spread demonstrates how the limit is the same for different initial conditions.

Estimates for a single run were obtained by implementing Definition 2.5.1 with a sufficiently large value for N. In a naive implementation, the components of v_t rapidly exceed floating point capacity. The trick is that

$$\ln \frac{\|v_N\|}{\|v_0\|} = \ln \frac{\|v_N\|}{\|v_{N-1}\|} \cdot \frac{\|v_{N-1}\|}{\|v_{N-2}\|} \cdots \frac{\|v_1\|}{\|v_0\|} = \sum_{t=1}^{N} \ln \frac{\|v_t\|}{\|v_{t-1}\|}.$$

Note that $\|v_t\| = \|J_{t-1}v_{t-1}\|$, implying that we can renormalize v_t to have unit norm after each step and accumulate the sum of $\ln\|J_t v_t\|$. This also shows that the Jacobian of a chaotic system expands vectors in one direction on average.

Chapter 3
THE THEORY OF CHAOTIC DERIVATIVES

This chapter is an exposition of results from a long line of work in chaotic systems. Instead of being a comprehensive resource for these ideas, this chapter is intended to guide intuition and serve as a starting point for understanding referenced works. Concepts include Oseledets' multiplicative ergodic theorem and the Lyapunov spectrum (Section 3.1); attractors, SRB measures, and the stable/unstable manifold theorem (Section 3.2); Ruelle's formula (Section 3.3); and space-split sensitivity (Sections 3.4–3.7).

3.1 Chaos, Revisited

3.1.1 Linear ODEs

To motivate Oseledets' theorem for general dynamical systems, we first turn to the simpler case of linear systems. Let $A : \mathbb{R}^n \to R^n$ be a linear transformation. Assume that A is diagonalizable, so there are matrices P and D such that the columns of P are the eigenvectors of A and the diagonal entries of D are the eigenvalues of A. Consider the linear ODE $\frac{\mathrm{d}x}{\mathrm{d}t} = Ax$. According to the tangent dynamics from the previous chapter, a tangent vector v evolves according to the homogeneous tangent equation

$$\frac{\mathrm{d}v}{\mathrm{d}t} = J(t)v \tag{3.1}$$

where $J(t)$ is the Jacobian of the system at time t. In a linear system, the Jacobian is always A (since $\frac{\mathrm{d}}{\mathrm{d}x}Ax$ is constant). Hence, Equation 3.1 becomes $\frac{\mathrm{d}v}{\mathrm{d}t} = Av = PDP^{-1}v$. By setting $w := P^{-1}v$, we get that $\frac{\mathrm{d}w}{\mathrm{d}t} = Dw$. The solution to this ODE is $w(t) = e^{Dt}w(0)$, where e^{Dt} is a matrix exponentiation. Finally, we have

$$v(t) = Pe^{Dt}P^{-1}v(0). \tag{3.2}$$

Now, we know exactly how tangent vectors evolve in the system. Let the eigenvalues of A be $\lambda_1 > \cdots > \lambda n$ and the corresponding eigenvectors be $v_1, \ldots, v_n$. If we set $v(0)$ to v_1, then

$$v(t) = Pe^{Dt}P^{-1}v_1 = Pe^{Dt}\begin{bmatrix}1 & \cdots & 0\end{bmatrix} = P\begin{bmatrix}e^{\lambda_1 t} & \cdots & 0\end{bmatrix} = e^{\lambda_1 t}v_1.$$

Generalizing this, we know that v_i becomes $e^{\lambda_i t}v_i$ after t time. Therefore, if $\lambda_i < 0$, then v_i shrinks to zero. If $\lambda_i = 0$, then v_i remains constant. And if $\lambda_i > 0$, then v_i grows in norm exponentially. From just local information (the Jacobian), we know how tangent solutions evolve, globally!

3.1.2 Oseledets' Theorem

In a general setting, we have nonlinear ODEs, so the exponential solution analysis is not applicable. The trouble is that the Jacobian varies over time. *Oseledets' multiplicative ergodic theorem* says that local information still characterizes the global behavior of the system, even with a continuously changing Jacobian. Just as tangent vectors started along eigenvectors had growth prescribed by eigenvalues in the linear case, tangent vectors started along *Lyapunov vectors* grow at an exponential rate determined by *Lyapunov exponents*. The "multiplicative" part of the theorem comes from the solution to a tangent equation of the form $\dot{v} = Jv$ using a continuous multiplication of J. When J is constant, a continuous multiplication becomes a natural exponential. Here is the theorem formally:

Theorem 2 (Oseledets' theorem [46])**.** For $x \in \mathcal{M}$ and $v \in T_x\mathcal{M}$, let

$$\lambda(x, v) = \lim_{N\to\infty} \frac{1}{N} \ln\|Df_x^n(v)\|.$$

Suppose μ is an invariant measure. Then there exist subspaces E_i such that $T_x\mathcal{M} = \oplus_{i=1}^n E_i(x)$ for almost all x. Furthermore, there exist measurable functions λ_i, called Lyapunov exponents, for which $\lambda_i(x) = \lambda(x, v)$ for all $v \in E_i(x)$. If μ is also ergodic, then $\lambda_i(x)$ is the same almost everywhere.

We call the direct sum of the subspaces $E_i(x)$ for which $\lambda_i > 0$ the *unstable subspace* at x. Similarly, the *stable subspace* at x is the direct sum of $E_i(x)$ where $\lambda_i < 0$ and the *center subspace* the direct sum of those where $\lambda_i = 0$. The unstable subspace contains all tangent vectors at x that grow exponentially forward in time and the stable subspace all vectors that shrink exponentially forward in time (or, equivalently, grow when traveling backwards—see Chapter 6 for more on this duality). When f is the flow map of an ODE with vector field F, the center subspace $E^c(x)$ exists and contains the flow direction $F(x)$ at x. In this direction, tangent vectors do not shrink nor expand since perturbing x forwards by $\epsilon F(x)$ is the same as starting the system ϵ time in the future.

3.2 The Global Geometry of Chaotic Systems

When visualized, chaotic systems have visually striking complex, fractal-like shapes in phase space. These shapes are *attractors*, subsets of phase space that all trajectories tend towards. More precisely, an attractor Λ is an invariant set with a neighborhood, called the attractor's basin, such that $f^n(x) \to \Lambda$ for every x in the basin. Once on the attractor, trajectories smear to cover the entire attractor. In this section, we seek an understanding of chaotic attractors from two viewpoints: a geometric viewpoint of attractors as manifolds, and a probabilistic viewpoint on measures that describe attractors through density.

3.2.1 Unstable and stable manifolds

The unstable and stable subspaces are closely related to the *unstable and stable manifolds*.

Theorem 3 (Stable/unstable manifold theorem [46])**.** Suppose μ is invariant and suppose there is a positive (resp. negative) Lyapunov exponent almost everywhere. Then unstable manifolds (resp. stable manifolds) are well defined almost everywhere. In particular, define

$$W^u(x) = \{y \in \mathcal{M} : \limsup_{n\to\infty} \frac{1}{n} \log d(f^{-n}(x), f^{-n}(y)) < 0\}.$$

Then $W^u(x)$ is the unstable manifold at x. Similarly, $W^s(x)$ is the stable manifold at x, defined like $W^u(x)$ is but with $d(f^{-n}(x), f^{-n}(y))$ replaced with $d(f^n(x), f^n(y))$. Fur-

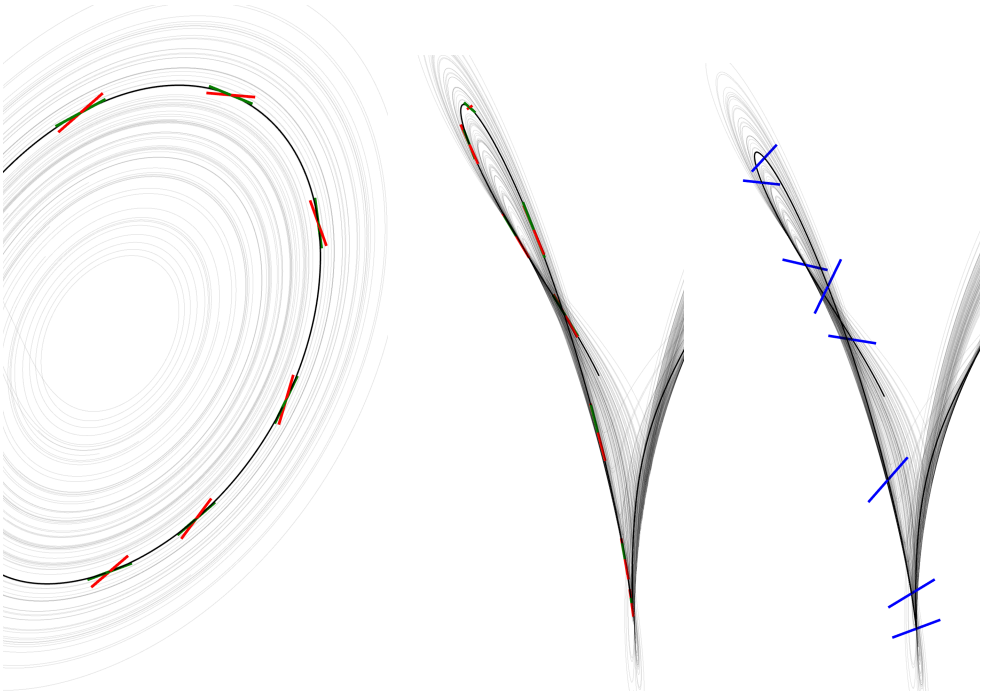

Figure 3.1: Unstable, stable, and center subspaces of the Lorenz attractor. The Lorenz system famously has a butterfly-shaped attractor [22]. (**Left**) The center subspaces (green) follow the flow direction (black). (**Center**) Both the center subspaces (green) and unstable subspaces (red) are tangent to the surface of the attractor. (**Right**) The stable subspaces (blue) stick out of the attractor, since all directions leaving the attractor are sucked back in.

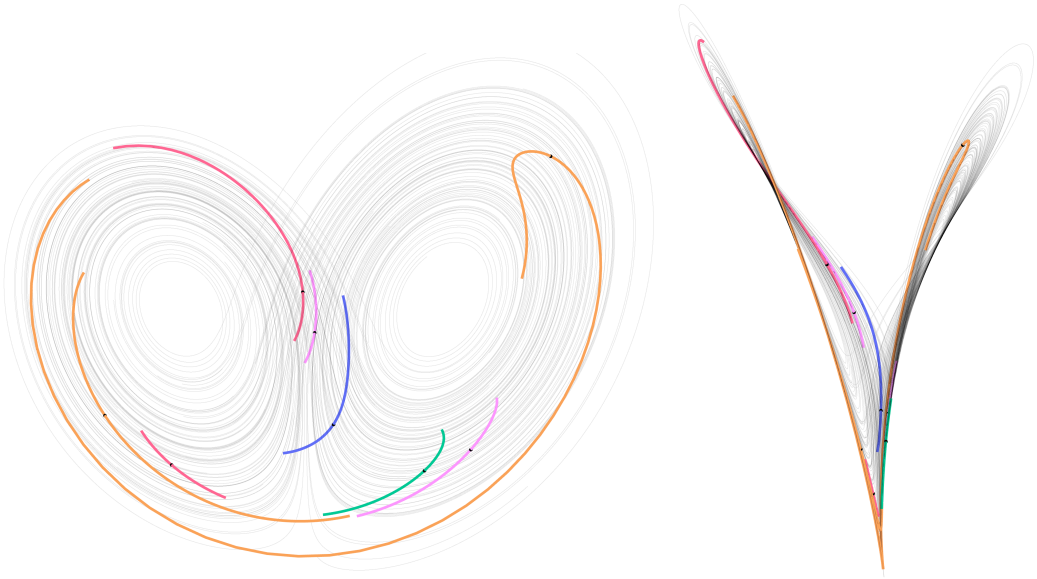

Figure 3.2: Unstable manifolds on the Lorenz attractor. Locally, the geometry of the attractor is described by a manifold, even though the global attractor is a fractal object.

thermore, $W^u(x)$ is tangent at x to the unstable subspace and $W^s(x)$ is tangent at x to the stable subspace.

In words, the unstable manifold at x is the set of points that were close to x in the past, while the stable manifold at x is the set of points that will be close to x in the future. An intriguing property of both Oseledets' theorem and the existence of stable and unstable manifolds is that they have very few assumptions. The crucial assumption is having an ergodic invariant measure.

3.2.2 SRB measures

So far, we know that unstable manifolds characterize an attractor's geometric shape and that there is an invariant measure which describes the distribution of states in phase space. We now link the shape to the distribution. Since all trajectories tend towards the attractor, the distribution must be concentrated also on the attractor. An *SRB measure* has an even

stronger link, which says that the invariant measure in fact has a density function when restricted to unstable manifolds.

Definition 3.2.1 ([46]). An f-invariant measure μ is called an SRB measure if f has a positive Lyapunov exponent almost everywhere and μ has absolutely continuous conditional measures on unstable manifolds.

In general, there are no theorems that determine if a concrete dynamical system has an SRB measure. One strict assumption that does guarantee the existence, and in fact, uniqueness, of an SRB measure is *uniform hyperbolicity*. Uniform hyperbolicity creates a strong structural guarantee on both the growth and contraction rates in the unstable and stable subspaces. In the specific case when f is a flow of a vector field, it is called an *Anosov diffeomorphism*.

Definition 3.2.2 ([43, 31, 47]). We say that f is an Anosov diffeomorphism there is a continuous invariant splitting of the tangent space into $E^u(x) \oplus E^s(x) \oplus E^c(x)$, and there are constants $c, \lambda > 0$ such that

- E^c is one-dimensional and $E^c(x) = \mathbb{R}F(x)$,
- E^u is uniformly expanding, meaning that $\|Df^{-t}v\| \leq ce^{-\lambda t}\|v\|$ if $v \in E^u(x)$,
- and E^s is uniformly contracting, meaning that $\|Df^{t}v\| \leq ce^{-\lambda t}\|v\|$ if $v \in E^s(x)$.

In an Anosov diffeomorphism, the attractor is also called an *Axiom A* attractor.

Uniform hyperbolicity also provides a strong condition on the separation between the unstable, stable, and center subspaces. The smallest angle between expanding, contracting, and center subspaces are globally bounded away from zero. This property also results in a numerical test for hyperbolicity by examining the distribution of angles between subspaces along a trajectory (see [19]).

3.2.3 Physical measures

In the definition of an attractor, there is a basin of attraction that tends towards the attractor. However, there is no guarantee that the basin has positive Lebesgue measure. This poses a computational challenge. To obtain states on the attractor, since it is unlikely that a fractal-like attractor has an analytic definition, we need to start from points in the basin and evolve for sufficiently long to get points near the attractor. A *physical measure* upgrades the basin of attraction to have positive Lebesgue measure.

Definition 3.2.3 ([46])**.** Suppose μ is invariant. We say that μ is a physical measure if there is a positive Lebesgue measure set $U \subset \mathcal{M}$ such that

$$\frac{1}{n}\sum_{t=0}^{n-1} S(x_t) \to \int S \, \mathrm{d}\mu$$

for every initial condition x_0 starting in U.

The definition of a physical measure is a variation on the statement of Birkhoff's ergodic theorem. Recall that theorem says that an expectation over an invariant ergodic measure μ can be approximated by computing a time average starting from a random point according to μ. Now, an expectation over a physical measure can be approximated by starting from a uniformly random point, or by choosing a random point "with one's eye" [17].

While physical measures do not distinguish between chaos and simple equilibria, we would like an SRB measure to also be physical for computational tractability. One connection between SRB measures and physical measures is that ergodic SRB measures with no zero Lyapunov exponents are physical measures [46].

3.3 Taming Tangent Growth with Ruelle's Formula

Recall that we want a derivative of an infinite-time average with respect to the parameter, or $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$. As a proxy, we study the derivative of the finite-time average, $\frac{\mathrm{d}S_N}{\mathrm{d}\theta}$. By carrying the

derivative through the definition of S_N, we get

$$\frac{\mathrm{d}S_N}{\mathrm{d}\theta} = \frac{1}{N}\sum_{t=1}^{N}\frac{\partial}{\partial\theta}S(u_t) = \frac{1}{N}\sum_{t=1}^{N}\left\langle \nabla S(u_t), \frac{\partial u_t}{\partial\theta}\right\rangle. \tag{3.3}$$

3.3.1 Two ways of understanding the perturbation $\frac{\partial u_t}{\partial\theta}$

The interesting term in Equation 3.3 is $\frac{\partial u_t}{\partial\theta}$, the change in the state at step t due to a change in the parameter. One way of understanding this term is a brute-force expansion. For example, for $t = 3$ we have

$$\begin{aligned}
\frac{\partial u_3}{\partial\theta} &= \frac{\partial}{\partial\theta}[f(u_2;\theta)] \\
&= J(u_2;\theta)\frac{\partial u_2}{\partial\theta} + f_\theta(u_2;\theta) \\
&= J(u_2;\theta)\left(J(u_1;\theta)\frac{\partial u_1}{\partial\theta} + f_\theta(u_1;\theta)\right) + f_\theta(u_2;\theta) \\
&= J(u_2;\theta)J(u_1;\theta)\left(J(u_0;\theta)\frac{\partial u_0}{\partial\theta} + f_\theta(u_0;\theta)\right) + J(u_2;\theta)f_\theta(u_1;\theta) + f_\theta(u_2;\theta).
\end{aligned}$$

Assuming that the initial state u_0 does not depend on θ, we have (dropping the parameter θ for clarity)

$$\frac{\partial u_3}{\partial\theta} = J(u_2)J(u_1)f_\theta(u_0) + J(u_2)f_\theta(u_1) + f_\theta(u_2).$$

In general,

$$\frac{\partial u_t}{\partial\theta} = \sum_{k=0}^{t-1}\left(\prod_{i=t-1}^{k+1} J(u_i)\right) f_\theta(u_k)$$

Another way of understanding $\frac{\partial u_t}{\partial\theta}$ is through the *inhomogeneous tangent equation*, which we first encountered back in Section 2.4. This equation describes how a tangent vector evolves due to a perturbation in the dynamics from varying the parameter.

Claim 3.3.1. Define $v_0 = 0$ and $v_{t+1} = J(u_t)v_t + f_\theta(u_t)$. Then $v_t = \frac{\partial u_t}{\partial\theta}$.

Proof (perturbation). Let $\epsilon > 0$. By perturbing both the state by ϵv and the parameter by ϵ we get

$$u_{t+1} + \epsilon v_{t+1} = f(u_t + \epsilon v_t; \theta + \epsilon) \approx f(u_t;\theta) + \epsilon J(u_t)v_t + \epsilon f_\theta(u_t)$$

through a first-order expansion of f. Cancelling out $u_{t+1} = f(u_t)$ and dividing by ϵ yields the desired result. □

Proof (substitution). By the chain rule,

$$\frac{\partial u_{t+1}}{\partial \theta} = \frac{\partial}{\partial \theta} f(u_t) = J(u_t)\frac{\partial u_t}{\partial \theta} + f_\theta(u_t).$$

□

3.3.2 Expanding the finite-time average

Now we substitute back into the sum in Equation 3.3.

$$\begin{aligned}
\frac{1}{N}\sum_{t=1}^{N}\left\langle \nabla S(u_t), \frac{\partial u_t}{\partial \theta}\right\rangle =& \frac{1}{N}\left\langle \nabla S(u_1), f_\theta(u_0)\right\rangle \\
&+\frac{1}{N}\left\langle \nabla S(u_2), J(u_1)f_\theta(u_0) + f_\theta(u_1)\right\rangle \\
&+\frac{1}{N}\left\langle \nabla S(u_3), J(u_2)J(u_1)f_\theta(u_0) + J(u_2)f_\theta(u_1) + f_\theta(u_2)\right\rangle \\
&\vdots \\
&+\frac{1}{N}\left\langle \nabla S(u_N), J(u_N)\cdots J(u_1)f_\theta(u_0) + \cdots + J(u_{N-1})f_\theta(u_{N-2}) + f_\theta(u_{N-1})\right\rangle
\end{aligned} \tag{3.4}$$

The terms on the right sides of the inner products form a triangle of terms, with three interesting viewpoints. Each row is a contribution to the overall perturbation from the perturbation to a single point in the trajectory at step t. Each column shows how a perturbation in the dynamics from $\frac{\partial f}{\partial \theta}$ propagates forward in time due to the Jacobian at every point in the trajectory.

To interpret the diagonal, here is Equation 3.4 with some terms colored:

$$\begin{aligned}\frac{1}{N}\sum_{t=1}^{N}\left\langle \nabla S(u_t), \frac{\partial u_t}{\partial \theta}\right\rangle =& \frac{1}{N}\left\langle \nabla S(u_1), f_\theta(u_0)\right\rangle \\ &+\frac{1}{N}\left\langle \nabla S(u_2), J(u_1)f_\theta(u_0) + f_\theta(u_1)\right\rangle \\ &+\frac{1}{N}\left\langle \nabla S(u_3), J(u_2)J(u_1)f_\theta(u_0) + J(u_2)f_\theta(u_1) + f_\theta(u_2)\right\rangle \\ &\vdots \\ &+\frac{1}{N}\left\langle \nabla S(u_N), J(u_N)\cdots J(u_1)f_\theta(u_0) + \cdots + J(u_{N-1})f_\theta(u_{N-2}) + f_\theta(u_{N-1})\right\rangle\end{aligned} \tag{3.5}$$

The green terms form the sum

$$\frac{1}{N}\sum_{t=1}^{N}\left\langle \nabla S(u_t), f_\theta(u_{t-1})\right\rangle. \tag{3.6}$$

By Birkhoff's Ergodic Theorem (see Section 2.2 for a refresher), as N goes to infinity, the sum in Equation 3.6 goes to

$$\int_{\mathcal{M}}\left\langle \nabla S(u), f_\theta(f^{-1}(u))\right\rangle \mathrm{d}\mu(u).$$

We can do the same to the red terms in Equation 3.5, which form the sum

$$\frac{1}{N}\sum_{t=1}^{N}\left\langle \nabla S(u_t), J(u_{t-1})f_\theta(u_{t-2})\right\rangle. \tag{3.7}$$

As N goes to infinity, this sum goes to

$$\int_{\mathcal{M}}\left\langle \nabla S(u), J(f^{-1}(u))f_\theta(f^{-2}(u))\right\rangle \mathrm{d}\mu(u). \tag{3.8}$$

Since μ is an invariant measure and so $f^*\mu = \mu$, Equation 3.8 can also be written as

$$\int_{\mathcal{M}}\left\langle \nabla S(f(u)), J(u)f_\theta(f^{-1}(u))\right\rangle \mathrm{d}\mu(u) = \int_{\mathcal{M}}\frac{\mathrm{d}(S\circ f)}{\mathrm{d}u}\cdot f_\theta(f^{-1}(u))\,\mathrm{d}\mu(u)$$

In general, the kth diagonal computes a gradient of the time-k lag. Adding up all the diagonals results in Ruelle's formula, a formula for $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$ based on space averages rather than time averages.

Theorem 4 (Ruelle's formula [31])**.**

$$\frac{\partial \overline{S}}{\partial \theta} = \sum_{k=0}^{\infty} \int \frac{\mathrm{d}(S \circ f^k)}{\mathrm{d}u} \cdot \chi \, \mathrm{d}\mu$$

where $\chi := f_\theta \circ f^{-1}$.

Ruelle's formula is nice because it no longer involves $\frac{\partial u_t}{\partial \theta}$, which grows in norm exponentially as we saw in Section 2.5. However, it is still impractical because the iterated f makes Monte-Carlo estimates of the inner integral have high variance. While Ruelle only formally proved Theorem 4 for uniformly hyperbolic systems, it has been conjectured to hold for systems which are not uniformly hyperbolic and we assume it to hold from here on. (The main difficulty is the formal conditions under which the limit from the derivative and the limit to infinite time can be swapped.) In the remainder of the chapter, we explore how to make this formula computationally realizable.

3.4 Taming Ruelle's Formula with Space Splitting

If we knew that χ always laid in the stable subspace, then evaluating Ruelle's formula directly would not be a problem. But that is not guaranteed, so the norm can grow exponentially as χ propagates through successive Jacobians of f. The idea of space splitting, first introduced by Chandramoorthy, Śliwiak, and Wang [11, 35], is to decompose χ into three components. The *unstable component* χ_u lies in the unstable subspace, the *center component* χ_c lies in the center subspace, and the *stable component* χ_s breaks the pattern and does not necessarily strictly lie in the stable subspace. Each of the three components can be evaluated separately using Monte-Carlo, leading to three contributions which sum to $\frac{\mathrm{d}\overline{S}}{\mathrm{d}\theta}$.

Assume that the system has $m \geq 1$ positive Lyapunov exponents and an SRB measure μ. Let $q^1, \ldots, q^m$ be vector fields such that $\{q^1(x), \ldots, q^m(x)\}$ is an orthonormal basis of the unstable subspace at every $x \in \mathcal{M}$. Additionally, recall that F is the underlying vector

field of the ODE. The splitting of χ is

$$\chi = \chi_u + \chi_c + \chi_s = \underbrace{\left(\sum_{i=1}^{m} c^i q^i\right)}_{\chi_u} + \underbrace{(c^0 F)}_{\chi_c} + \underbrace{\left(\chi - \sum_{i=1}^{m} c^i q^i - c^0 F\right)}_{\chi_s},$$

where the c^i are scalar fields to be defined. So, Ruelle's formula becomes

$$\sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_u \, \mathrm{d}\mu + \sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_c \, \mathrm{d}\mu + \sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_s \, \mathrm{d}\mu. \quad (3.9)$$

The first term in Equation 3.9 is the *unstable contribution*, the middle is the *center contribution*, and the last term is the *stable contribution*.

Geometrically, the decomposition splits the effect of a parameter perturbation into three pieces. The unstable contribution measures how the parameter affects the density on the attractor. The center contribution describes how the parameter affects the speed of the flow. Finally, the stable contribution captures how the parameter moves the location of the attractor, or the support of the invariant measure.

3.5 Space Splitting: the Stable Contribution

Evaluating the stable contribution is equivalent to solving the regularized tangent equation

$$v_{t+1} = J_t v_t + \chi_s(u_{t+1})$$

and evaluating

$$\lim_{N\to\infty} \frac{1}{N} \sum_{t=1}^{N} \langle \nabla S(u_t), v_t \rangle. \quad (3.10)$$

The reasoning is exactly the same as the reasoning we used to derive Ruelle's formula in Section 3.3. By expanding Equation 3.10 into a triangle of terms like we did in Equation 3.4 and converting sums along diagonals into space averages, we get exactly the formula we want,

$$\sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_s \, \mathrm{d}\mu.$$

Now we motivate the definition of the scalar fields $c^0, \dots, c^m$. We want the regularized tangent solutions v_t to be bounded in norm—otherwise, we are back to the chaotic problem. We can choose the scalars to make v_{t+1} orthogonal to the unstable and center subspaces at u_{t+1}. In particular, we want

$$\begin{aligned} v_{t+1} \cdot q^i_{t+1} &= 0 \quad \text{for } i = 1, \dots, m \\ v_{t+1} \cdot F_{t+1} &= 0. \end{aligned}$$

The subscript notation q^i_{t+1} and F_{t+1} is shorthand for evaluating the vector fields at u_{t+1}. Unfortunately, this system is not trivial to solve because while the basis of q^i are orthonormal, they are not necessarily orthogonal to the flow direction $\hat{F}$. One simplification that can be made is not projecting out the center direction, since the bulk of the growth in tangent space is due to the expanding directions of the unstable subspace spanned by the q^i. In that case, direct expressions for c^i are given by

$$c^i_{t+1} := q^i_{t+1} \cdot (J_t v_t + f_\theta(u_t)).$$

3.6 Space Splitting: the Center Contribution

The center component is treated separately from the stable component because a perturbation in the center direction can still grow over time, albeit just not exponentially. Recall that the center direction corresponds to directions where the Lyapunov exponent is zero. This means the average growth or contraction in norm is dominated by exponential growth, but that growth can still be polynomial in time, which would increase the variance of a gradient estimator with integration time.

Computing the center contribution relies on the covariance property, which says that $F(f(u)) = J(u)F(u)$, or that the flow direction evolved forward for the fixed timestep h is the same as the flow direction one timestep later, in the limit as $h \to 0$. Then since

$$D(S \circ f^k) \cdot \chi_c = \frac{\mathrm{d}S}{\mathrm{d}u} \cdot (J_{k-1} \cdots J) \cdot (c^0 F) = c^0 \left(\frac{\mathrm{d}S}{\mathrm{d}u} \circ f^k \right) (F \circ f^k),$$

we have that

$$\sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_c \, \mathrm{d}\mu = \sum_{k=0}^{\infty} \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} c_k^0 \frac{\mathrm{d}S}{\mathrm{d}u}(u_{t+k}) \cdot F(u_{t+k}).$$

3.7 Space Splitting: the Unstable Contribution

The final piece of the decomposition is the unstable contribution. Recall that the unstable component

$$\chi_u = \sum_{i=1}^{m} c^i q^i$$

is tangent to the unstable manifold. Hence, the strategies we used to evaluate the stable and center contributions no longer apply because tangent vectors in the unstable subspace grow exponentially in norm with time. The main idea to evaluate the unstable contribution is integration by parts on the unstable manifold.

We refer the reader to Sliwiak's thesis for a full derivation of a formula for the unstable contribution [35, Section 5.3]. The short story is that

$$\begin{aligned} \sum_{k=0}^{\infty} \int D(S \circ f^k) \cdot \chi_u \, \mathrm{d}\mu &= \sum_{k=0}^{\infty} \sum_{i=1}^{m} \int c^i \partial_{q^i} (S \circ f^k) \, \mathrm{d}\mu \\ &= -\sum_{k=0}^{\infty} \sum_{i=1}^{m} \int (S \circ f^k)(c^i g^i + b^{i,i}) \, \mathrm{d}\mu \end{aligned} \quad (3.11)$$

where

$$b^{i,j} := \partial_{q^j} c^i \text{ and } g^i := \frac{\partial_{q^i} \rho}{\rho}.$$

The meaning of the ∂_{q^i} operator is the directional derivative along the unstable basis vector q^i. The density function $\rho(x)$ is the density of μ conditioned on the unstable manifold at x. The critical transformation in the last expression of Equation 3.11 is that the gradient operator is no longer on $S \circ f^k$, so tangent vectors are not blown up by the iterated Jacobian of f.

The vector quantity g^i is a special term known as the *density gradient*. The density gradient at x describes the relative change in the density at x (in some direction), due to the

division by ρ. Another interpretation is that $\frac{\partial_{q_i}\rho}{\rho} = \partial_{q_i} \ln \rho$. The density gradient comes from μ being an invariant measure. In particular, the *measure preservation* property says that

$$\rho(f(x)) = \frac{\rho(x)}{|\det \frac{\mathrm{d}f}{\mathrm{d}x}(x)|}.$$

The density gradient arises naturally from an "importance sampling" rearrangement. Suppose for a moment that μ has full support over the entire space, so ρ is defined unconditionally on the whole space. By measure preservation, we have that

$$\int S(x)\,\mathrm{d}\mu(x) = \int S(f(x))\,\mathrm{d}\mu(x) = \int S(f(x))\rho(x)\,\mathrm{d}x,$$

since μ is an invariant measure. Then taking the derivative yields

$$\frac{\mathrm{d}}{\mathrm{d}\theta}\int S(f(x))\rho(x)\,\mathrm{d}x = \int \frac{\mathrm{d}S}{\mathrm{d}x}(f(x))\frac{\partial f}{\partial\theta}(x)\rho(x) + S(f(x))\frac{\mathrm{d}\rho}{\mathrm{d}\theta}\,\mathrm{d}x$$

Ignoring the second term, integration by parts on the first term yields

$$\int S(f(x))\frac{\mathrm{d}}{\mathrm{d}x}\left[\frac{\partial f}{\partial\theta}(x)\rho(x)\right]\mathrm{d}x$$

assuming that the boundary is periodic. Then

$$\int S(f(x))\left(\frac{\partial^2 f}{\partial x\,\partial\theta}\rho + \frac{\partial f}{\partial\theta}\frac{\mathrm{d}\rho}{\mathrm{d}x}\right)\mathrm{d}x = \int (S\circ f)\left(\frac{\partial^2 f}{\partial x\,\partial\theta} + \frac{\partial f}{\partial\theta}\cdot\frac{1}{\rho}\frac{\mathrm{d}\rho}{\mathrm{d}x}\right)\mathrm{d}\mu$$

where we divided by ρ to convert back to integrating with respect to μ. This "importance sampling" led to the appearance of a density gradient $\frac{1}{\rho}\frac{\mathrm{d}\rho}{\mathrm{d}x}$. We see that the $b^{i,i}$ term in Equation 3.11 corresponds to a derivative which is the change of the perturbation $\frac{\mathrm{d}f}{\mathrm{d}\theta}$.

Chapter 4

A SMOOTH PROPER ODE FOR SWARMING BASED ON ENERGY POTENTIALS

In the theoretical framework developed by the last two chapters, we assumed that the dynamics were driven by an ODE $\frac{\mathrm{d}u}{\mathrm{d}t} = F(u)$. Furthermore, we assumed that this ODE was C^k, so that F had k continuous derivatives. In the classical definition of boids, is not an autonomous ODE nor is it continuous. This chapter defines and motivates a C^k autonomous ODE which results in a system that swarms.

4.1 State and Dynamics

Consider a swarm of n boids living in a d-dimensional space. The state consists of positions $\vec{x} \in \mathbb{R}^{n \times d}$ and headings $\vec{\varphi} \in \mathrm{SO}(d)^n$.

The boids ODE is

$$\begin{aligned} \frac{\mathrm{d}\vec{x}}{\mathrm{d}t} &= s \cdot N(\vec{\varphi}) \\ \frac{\mathrm{d}\vec{\varphi}}{\mathrm{d}t} &= -\nabla_{\varphi} E(\vec{x}, \vec{\varphi}) \end{aligned}$$

where s is a constant speed, N converts headings into unit vectors, and E is an energy to be defined. The idea is that E is small if the boids are in a swarming formation and large if not; boids steer to minimize E, resulting in swarming motion.

4.2 Energies

The total energy consists of three separate energies representing alignment, avoidance, and cohesion.

$$E(\vec{x}, \vec{\varphi}) = E_{\text{align}}(\vec{x}, \vec{\varphi}) + E_{\text{avoid}}(\vec{x}, \vec{\varphi}) + E_{\text{cohere}}(\vec{x}, \vec{\varphi})$$

These energies take inspiration from a sample notebook in a molecular dynamics library [33].

All energies involve a *visibility* term which weights the contribution of one boid on another based on the first boid's location in the second boid's field of vision. Intuitively, a boid pays more attention to other boids it can directly see in front of it. Formally, define

$$V(o, \varphi) := \frac{(1 + \hat{o} \cdot N(\varphi))^k}{2^k}$$

where o is the vector offset from the boid of interest to another boid and φ is the heading of the boid of interest. In the remainder of the chapter, we use the hat to denote a normalized vector. We also reference an integer k, which is a parameter of the model controlling the number of continuous derivatives. The purpose of the division by 2^k is to normalize V to be in $[0, 1]$.

4.2.1 Align Energy

Alignment is the principle that boids should travel in the same directions as their neighbors. We define align energy to be a sum of pairwise align energy weighted by the visibility term:

$$E_{\text{align}}(\vec{x}, \vec{\varphi}) := \sum_{i \neq j} E_{\text{align}}^{\text{pairwise}}(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i, \vec{\varphi}_j) V(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i) \tag{4.1}$$

Pairwise align energy consists of a weighting by a parameter J_{align}, a smooth distance thresholding up to a parameter D_{align}, and an alignment term $1 - N(\varphi_1) \cdot N(\varphi_2)$:

$$E_{\text{align}}^{\text{pairwise}}(o, \varphi_1, \varphi_2) := \begin{cases} J_{\text{align}} \left(1 - \frac{\|o\|}{D_{\text{align}}}\right)^k (1 - N(\varphi_1) \cdot N(\varphi_2))^2 & \|o\| \leq D_{\text{align}} \\ 0 & \|o\| > D_{\text{align}} \end{cases} \tag{4.2}$$

At a fixed distance, the pairwise energy is maximized when the two boids point in opposite directions and minimized when they point in the same direction. Figure 4.1 visualizes the energy landscape for a single pair of boids.

4.2.2 Avoid Energy

Avoidance is the principle that boids avoid collisions with other boids and the environment. Similarly to align energy, avoid energy consists of a sum of pairwise avoid energy and wall

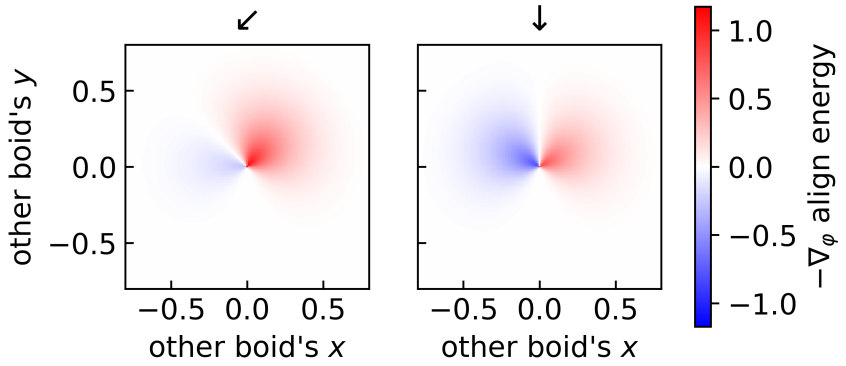

Figure 4.1: The pairwise align energy landscape. One boid, pointing up, is at $(0, 0)$. The direction of the arrow indicates the direction the other boid is pointing in.

avoid energy:

$$E_{\text{avoid}}(\vec{x}, \vec{\varphi}) := \sum_{w \in \mathcal{W}} \sum_{i=1}^{n} E_{\text{avoid}}^{\text{wall}}(\text{proj}_w(\vec{x}_i) - \vec{x}_i, \vec{\varphi}_i, w_\varphi) + \sum_{i \neq j} E_{\text{avoid}}^{\text{pairwise}}(\vec{x}_j - \vec{x}_i) V(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i) \quad (4.3)$$

Pairwise avoid energy only depends on the offset vector. It has a weighting by J_{avoid} and a smooth distance thresholding up to D_{avoid}:

$$E_{\text{avoid}}^{\text{pairwise}}(o) := \begin{cases} J_{\text{avoid}} \left(1 - \frac{\|o\|}{D_{\text{avoid}}}\right)^k & \|o\| \leq D_{\text{avoid}} \\ 0 & \|o\| > D_{\text{avoid}} \end{cases}$$

The visibility term in avoid energy is crucial. Without visibility, the sum of pairwise avoid energy would have no dependence on the heading, and so the gradient of pairwise avoid energy with respect to heading would be zero. Put simply, the boids would not avoid collisions with each other. Wall avoid energy has a smooth weight depending on the distance to the wall and a weight based on the alignment between the heading and the normal of the

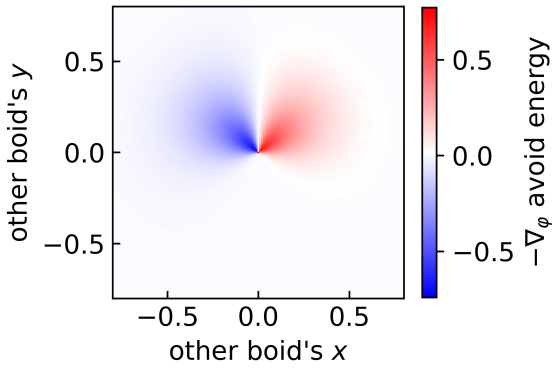

Figure 4.2: The pairwise avoid energy landscape. One boid, pointing up, is at $(0, 0)$.

wall, encouraging boids to steer away:

$$E^{\text{wall}}_{\text{avoid}}(o, \varphi, w_\varphi) := \begin{cases} J_{\text{avoid}} \left(1 - \frac{\|o\|}{D_{\text{avoid}}}\right)^k \frac{(1 - N(\varphi) \cdot N(w_\varphi))^k}{2^k} & \|o\| \leq D_{\text{avoid}} \\ 0 & \|o\| > D_{\text{avoid}}. \end{cases}$$

In the definition of wall avoid energy, $\mathcal{W}$ is a set of all walls, $\text{proj}_w(\cdot)$ projects a point onto the wall, and w_φ is the normal direction of wall w. Figure 4.2 visualizes the energy landscape for a single pair of boids. In practice, the weight of wall avoidance energy should be higher than the weight of pairwise avoidance so that boids always prioritize avoiding the environment over avoiding each other.

4.2.3 Cohere Energy

Cohesion is the principle that boids stick close to their neighbors. This is distinct from alignment, which puts nearby boids on parallel paths[1]. Specifically, cohesion works by steering towards the center of mass of the neighborhood. Computing a center of mass involves a weighted average over a variable number of neighbors, which is challenging to express in a

[1]Thanks to Craig for this distinction.

differentiable manner. First, the total cohere energy is a sum of per-boid cohere energy:

$$E_{\text{cohere}}(\vec{x}, \vec{\varphi}) := \sum_{i=1}^{n} E^{(i)}_{\text{cohere}}(\vec{x}, \vec{\varphi}).$$

Per-boid cohere energy involves the center of mass and a smooth weighting term:

$$E^{(i)}_{\text{cohere}}(\vec{x}, \vec{\varphi}) := J_{\text{cohere}} \frac{(1 - \widehat{COM}_i(\vec{x}, \vec{\varphi}) \cdot N(\vec{\varphi}_i))^k}{2^k} \cdot \sum_{j \neq i} w(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i).$$

This says that energy is lowest when the heading of the boid closely matches the heading towards the center of mass. The term COM_i computes that heading by taking a weighted average of the offsets $x_j - x_i$ weighted by a function w:

$$COM_i(\vec{x}, \vec{\varphi}) := \frac{\sum_{j \neq i} w(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i)(\vec{x}_j - \vec{x}_i)}{\epsilon + \sum_{j \neq i} w(\vec{x}_j - \vec{x}_i, \vec{\varphi}_i)}$$

The purpose of the ϵ, set to a small value such as 10^{-6}, is to keep the center of mass function differentiable when there are no nearby neighbors. The w function smoothly ramps down the influence of a boid on the weighted averagee depending on its distance and visibility:

$$w(o, \varphi) := \begin{cases} V(o, \varphi) \left(1 - \frac{\|o\|}{D_{\text{cohere}}}\right)^k & \|o\| \leq D_{\text{cohere}} \\ 0 & \|o\| > D_{\text{cohere}} \end{cases}$$

Figure 4.3 visualizes the cohere energy landscape in an example swarm.

4.3 From Classical Boids to an ODE

Here we discuss challenges when converting boids into an ODE that motivate the energy potential formulation.

4.3.1 Velocity constraints

Boids are typically defined with a velocity vector to represent both speed and heading. In this definition, position is incremented with velocity and velocity is incremented with an acceleration. However, it can be undesirable for boids to move too quickly or too slowly. A

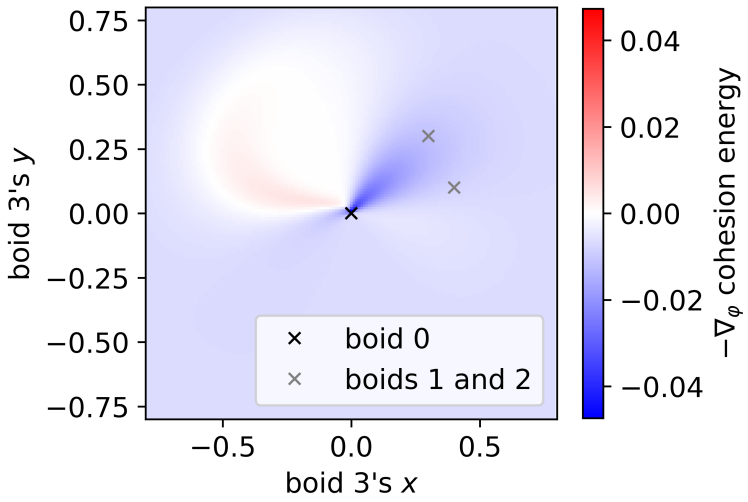

Figure 4.3: The cohesion energy landscape. This setting has four boids. Boid 0, on which cohesion energy is computed, is at the origin. Boids 1 and 2 are up to the right. When boid 3 is in the same direction as boids 1 and 2, cohesion energy strongly steers boid 0 towards them. If boid 3 is closer to boid 0 than boids 1 and 2 are, it can influence boid 0 to steer to the left.

natural approach to enforce a minimum and maximum speed is to clamp the norm of the velocity vector after applying the acceleration. In pseudocode, this might look like

```
def update_boid(old_pos, old_vel, accel, dt):
  new_pos = old_pos + old_vel * dt
  new_vel = old_vel + accel * dt
  speed = norm(new_vel)
  if speed < MIN_SPEED:
    new_vel = new_vel / speed * MIN_SPEED
  else if speed > MAX_SPEED:
    new_vel = new_vel / speed * MAX_SPEED
  return new_pos, new_vel
```

Modeling this update as a vector field, we want a function F such that

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v \text{ and } \frac{\mathrm{d}v}{\mathrm{d}t} = F(x, v).$$

This forces F to be aware of the discretized timestep `dt` to maintain the speed constraint. It is possible to add a drag term to penalize high speed and a "boost" term to encourage a minimum speed. However, we found these terms difficult to balance with other swarming rules. Our approach is to make boid speed constant and represent headings directionally. It could be possible to reintroduce variable speed into the state and use energies that penalize speed leaving a desired range.

4.3.2 Hard distance thresholds

Each of the rules in boids comes with a maximum distance of interaction D. Fix a boid B and consider the influence of another boid B' on B. In order for the effect of a rule determining how B should steer in response to B' to be differentiable with respect to the position of B', that effect should vary smoothly, from zero when B' is further than D from B, to nonzero when B' is closer than D to B. While a distance-based falloff is typically used within a distance of D so that boids that are further away have less influence, the cutoff for the influence outside of D is usually a hard conditional. We not only need the influence to be continuous but also a certain number of its derivatives to be continuous. Our general approach is to use a polynomial that attenuates the weight down to zero at a distance of

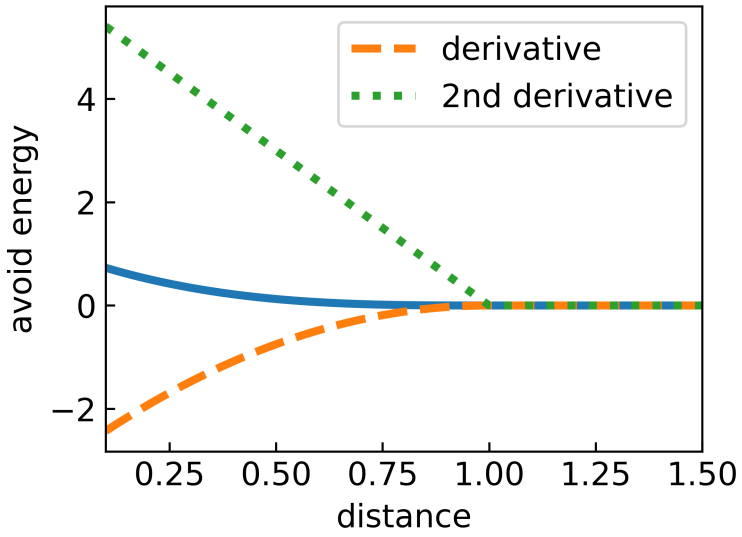

Figure 4.4: A C^2 distance falloff. In this example, the radius of interaction is 1. The energy smoothly drops to zero outside of this radius. The energy is defined as a piecewise function of the distance d. If $d < 1$ then the energy is $(1 - d)^3$; otherwise the energy is 0.

D. The degree of the polynomial controls the number of continuous derivatives. Figure 4.4 shows an example.

4.3.3 Combining steering directions

The main challenge that motivates the energy potentials is figuring out which direction to steer to satisfy all behaviors. Usually, the steering function is decomposed into a separate steering vector from each behavior. How should these vectors be combined? It is difficult to combine them in a way that is differentiable and prioritizes the right behavior at the right time. By introducing a scalar potential function, combining potentials from separate behaviors is as easy as adding them and we can appeal to the gradient to find the steering direction that balances them.

Chapter 5
TOWARDS DERIVATIVES OF CHAOTIC SWARMS

In this chapter, we apply the theory from Chapter 3 to the boids swarming ODE from Chapter 4. Throughout the chapter, we fix the dimension that the boids live in to two. To recap, the system has a state which is a pair of n 2-dimensional vectors representing position and n angles representing direction. The dynamics are driven by a vector field $F(x, \varphi) = (s \cdot N(\varphi), -\nabla_\varphi E(x, \varphi))$ where s is a constant speed, $N(\varphi) = (\cos(\varphi), \sin(\varphi))$, and E is a swarming energy function. Finally, f is the time-h flow map which discretizes the flow.

5.1 Computing the Unstable Dimension

Before splitting space, we need to compute the dimension of unstable manifolds. This means computing the number of positive Lyapunov exponents.

5.1.1 Benettin's Algorithm

The standard method for computing the m largest Lyapunov exponents is known as *Benettin's algorithm*. It works through repeated orthonormalization using QR decomposition. Recall that in Section 2.5, we computed the maximum Lyapunov exponent by starting with a random unit vector, evolving it forward by the Jacobian, recording the relative change in norm, and renormalizing the vector. Then, the maximum Lyapunov exponent was an average of the per-step relative changes. This process is visualized in Figure 5.1. By Oseledets' theorem, almost all directions have a component in the most expanding subspace, and that component eventually dominates the entire vector as successive renormalizations shift the weight towards it.

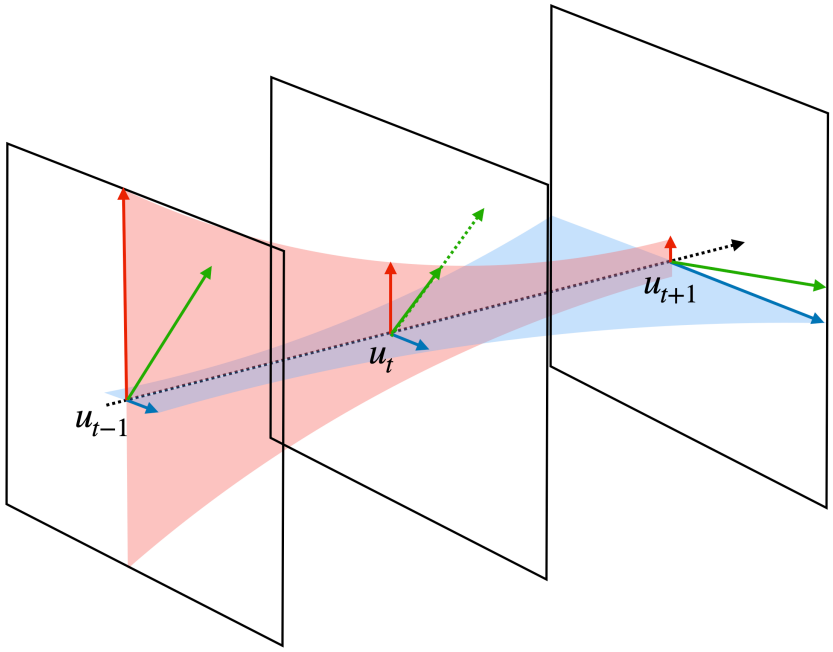

Figure 5.1: Repeated orthonormalization for the maximum Lyapunov exponent. The black dashed arrow is the flow in phase space, the black squares represent the tangent space at different points, and the red and blue regions represent contraction and expansion under tangent dynamics. At step $t-1$, the process starts with randomly initialized unit vector in green. When the vector evolves through Df_{t-1}, its component in the expanding direction (blue) grows and its component in the contracting direction (red) shrinks. After it is renormalized, the relative weight in the expanding direction increases. Eventually, the vector converges to the most expanding direction.

To find the second largest Lyapunov exponent, we do the same thing, except for tracking two unit vectors q^1 and q^2 that are always orthogonal (the superscript is an index, not an exponent). After sending both vectors through the Jacobian and renormalizing q^1, projecting q^1 out of q^2 before renormalizing q^2 ensures that q^2 has no component in the most expanding direction. Hence, the evolution of q^2 is controlled by the second Lyapunov exponent. A generic method for keeping an orthonormal basis of m vectors and obtaining the

relative growth of each direction is QR decomposition, visualized in Figure 5.2.

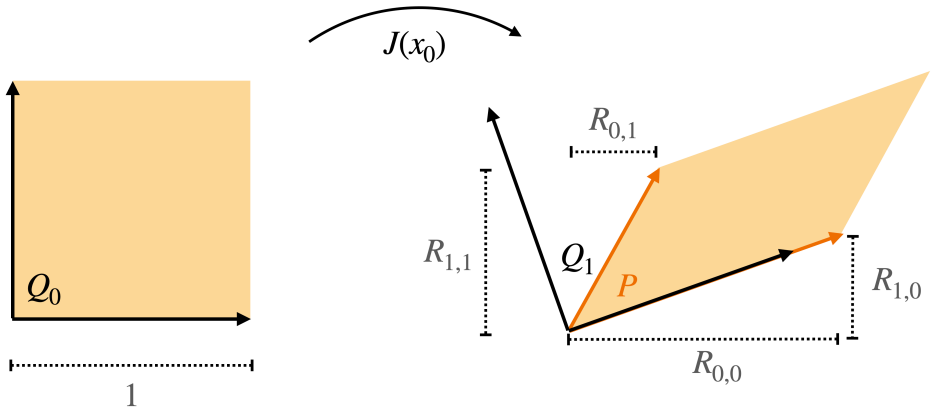

Figure 5.2: QR decomposition. The orthonormal basis Q_0 sent through the Jacobian becomes P. Taking the QR decomposition of P results in Q_1, a new orthonormal basis, and R. The diagonal entries of R encode the local growth rates—in this case, an expansion along the first column of Q_0 given by $R_{0,0}$ and a contracting along the second column of Q_0 given by $R_{1,1}$. This figure is inspired by Figure 3 in [20].

Algorithm 1 defines the algorithm in full detail. It assumes a background trajectory $u_0, \ldots, u_N$ of states. Note that the basis of tangent vectors Q is sent through the Jacobian of the discretized flow map f and not that of the flow itself F (line 4). It is also important to normalize the estimates by the time step h to estimate the growth rate independent of the discretization (line 7).

5.1.2 Implementation Details

We implemented Algorithm 1 in JAX, a Python library for high-performance numerical computing [8]. We observed that 64-bit floating point precision was necessary to make Lyapunov computations converge to correct values. We also used a small timestep of $h = 1/120$ and the RK4 method for integrating.

Algorithm 1 m largest Lyapunov exponents [7]

Require: N (number of iterations), m (number of exponents)

Ensure: m largest Lyapunov exponents $\lambda_1, \dots, \lambda_m$

$Q_0 \leftarrow \mathbf{I}^{n \times m}$ ▷ *initialize basis of m orthonormal tangent vectors*

$\lambda \leftarrow \mathbf{0} \in \mathbb{R}^m$ ▷ *running estimate of Lyapunov exponents*

for $t \leftarrow 0$ to N **do**

  $P \leftarrow Df_t Q_t$ ▷ *send tangent basis forward through Jacobian*

  $Q_{t+1}, R \leftarrow \mathrm{QR}(P)$ ▷ *orthonormalize P*

  $\lambda \leftarrow \lambda + \ln|\mathrm{diag}(R)|$ ▷ *accumulate local growth rates*

return $\lambda/(N \cdot h)$

Algorithm 1 is easily parallelizable by estimating λ starting from different initial conditions in parallel. By ergodicity, all estimates converge to the same exponents, so the exponents from different runs can be combined by just averaging. Sample code can be found in Appendix B.1.

5.1.3 Results

Running our implementation of Benettin's algorithm on a boids system with $n = 12$ boids results in Figure 5.3. The full list of parameters can be found in Appendix A. We conclude that for 12 boids, there are 11 unstable dimensions since there are 11 positive Lyapunov exponents. As expected, there is on zero Lyapunov exponent corresponding to the center subspace. The unstable dimensions for boids systems with $n \in \{6, 12, 18, 24, 30, 36\}$ are plotted in Figure 5.4.

5.2 Progress Towards Computing the Derivative

We implemented the full version of space splitting for chaotic flows with arbitrary unstable dimension in JAX, but have only tested the full version on systems with one-dimensional unstable manifolds. For boids, we have only tested a computation of the stable contribution.

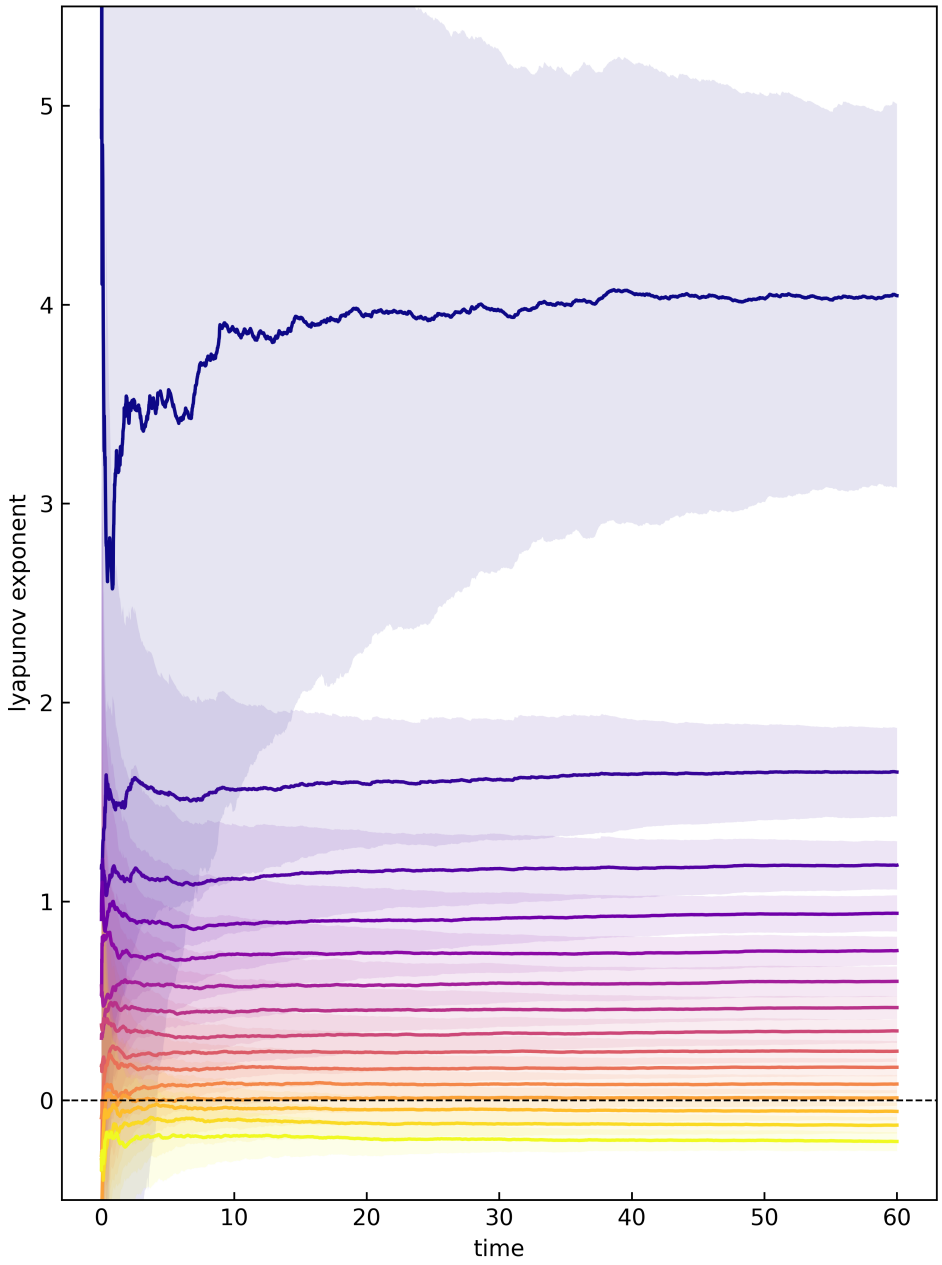

Figure 5.3: Time evolution of the 15 largest Lyapunov exponents. The system is sampled with 128 random initial conditions. All parameters are listed in Appendix A. Each solid line indicates the mean exponent; shaded regions show one standard deviation. The horizontal dashed line at zero is included for reference.

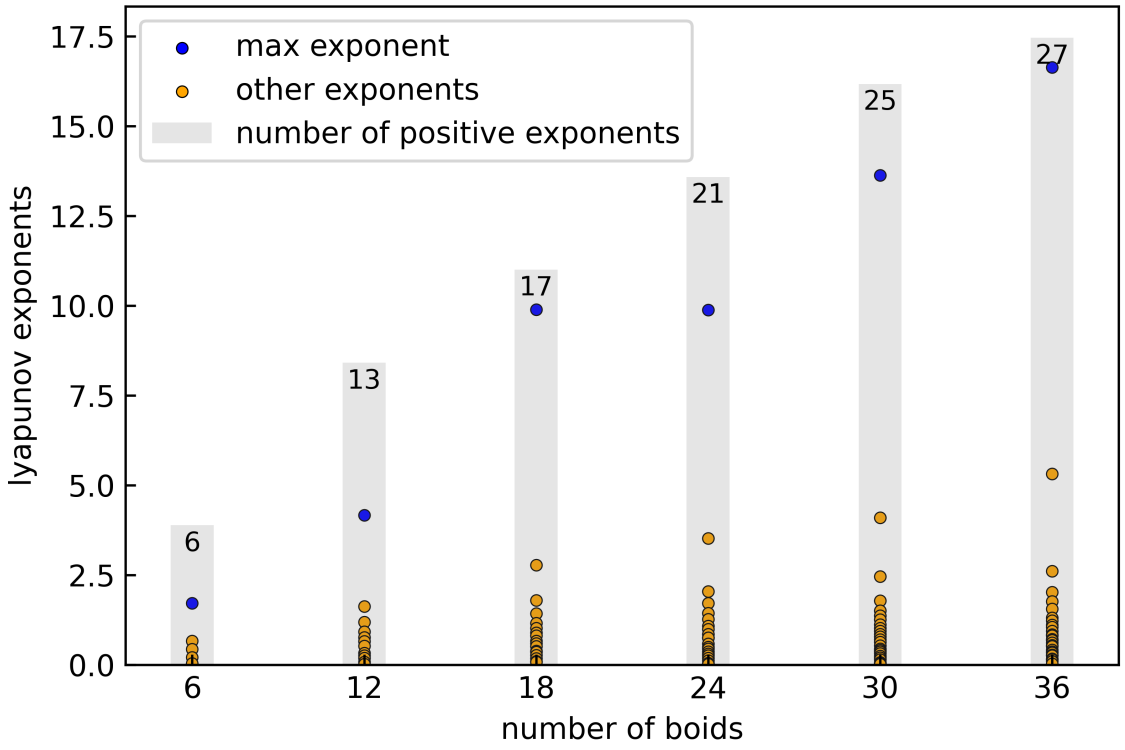

Figure 5.4: Number of positive exponents vs. number of boids. The number of positive exponents appears to scale sublinearly with the number of boids. The largest exponent grows roughly linearly, while most of the positive exponents remain under two.

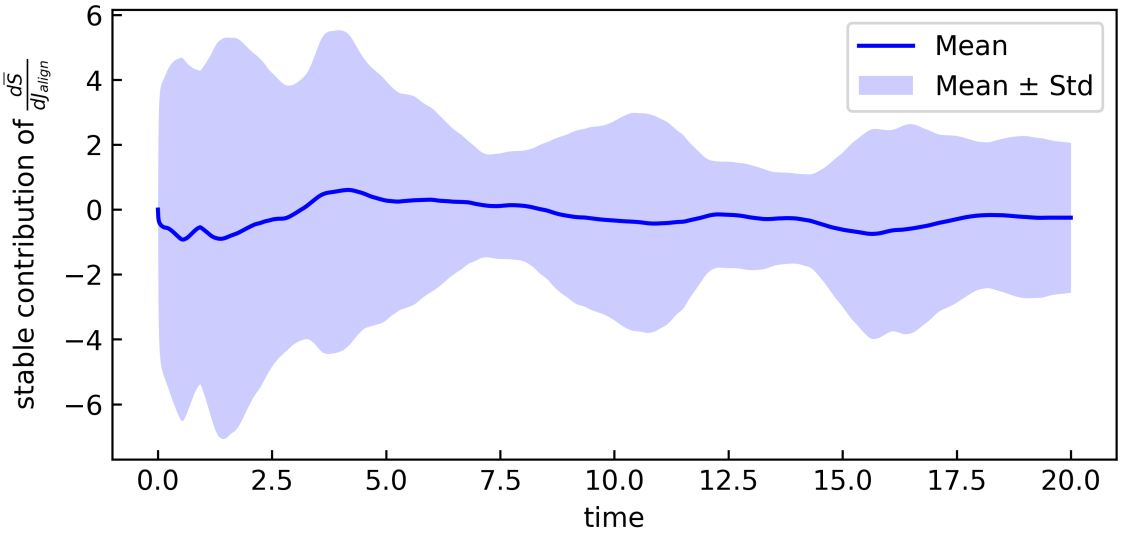

Figure 5.5: Stable contribution for the derivative of average x with respect to align weight.

Sample code is in Appendix B.2. Even getting the stable contribution to converge has been a challenge, with every layer of the stack from numerical stability, parameter and system choices, and theoretical considerations all potentially affecting the results. Figures 5.5 and 5.6 show the latest stable contribution results as of writing time. We compute the stable contribution for two different objectives S, the average x coordinate of all boids and the minimum pairwise distance, with respect to the parameter, align energy weight. To give a sense of the derivative values that we expect, the derivative of the average x coordinate with respect to any parameter in our model is zero due to symmetry. There is no reason for the center of all boids to be biased towards one side of the box. In Figure 2.1, we observed that the slope of the time-average of minimum distance plotted against align weight was negative. So, we should expect the derivative of minimum distance to be negative.

5.3 Discussion

The bright side of the current results is that the stable contributions remain bounded in time, unlike the original naive derivatives. This suggests that the unstable subspace computation

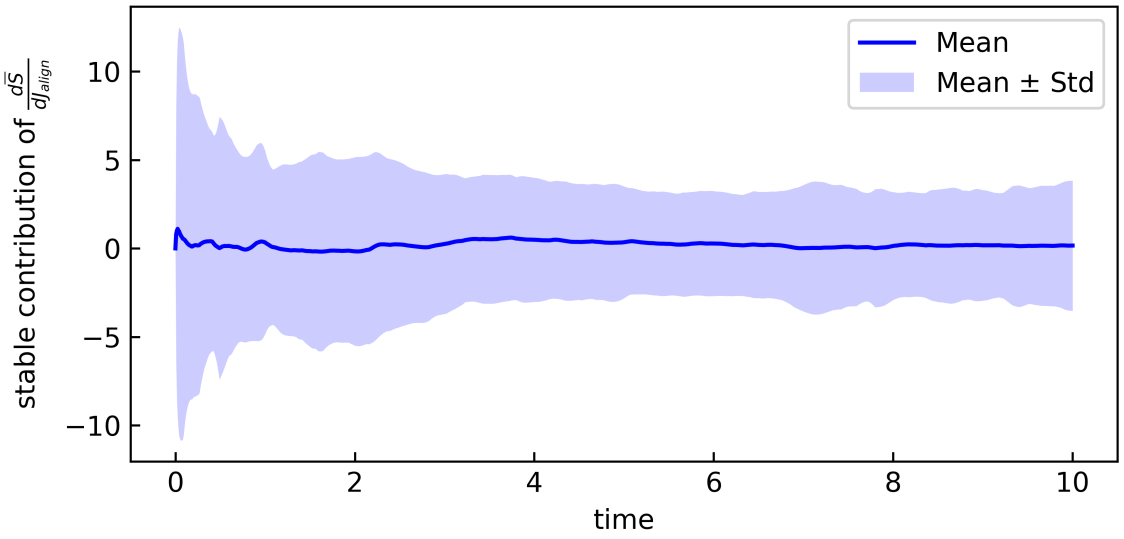

Figure 5.6: Stable contribution for the derivative of minimum distance with respect to align weight.

is working as intended, and so all unstable directions are indeed being projected out. We are hopeful that with enough engineering effort, the center and unstable contributions can also be run on boids.

One negative result is the lack of convergence over time for estimates of both average x and minimum distance. This may be because we currently do not project the center direction out. Center projection could be implemented by solving a system of $m+1$ equations (where m is the unstable dimension) to solve for the center component, or by finding a full spectrum to project directly onto the stable subspace. Other explanations for a lack of convergence are simply not integrating for long enough, and discontinuities in derivatives of the vector field, which we debugged and resolved many of and usually appear as enormous spikes in local growth rate. It is difficult to tell apart an insufficient length of integration from there being multiple ergodic components in the system.

The most glaring negative result is that the stable contribution of minimum distance appears to be zero, when we know the final derivative should be negative. This suggests that

the majority of the derivative contribution comes from the unstable contribution, which is unfortunately expensive with each step taking $\mathcal{O}(m^3)$ time. Of course, there could always be a structural issue, such as a severe lack of hyperbolicity.

Chapter 6
COTANGENT DYNAMICS FOR THE SMALLEST EXPONENT

Let V be a vector space. There is a *dual space*, denoted by V^*, containing *dual vectors*, which are linear functionals from V to $\mathbb{R}$. In this chapter, we explore how *cotangent vectors*, which are dual to tangent vectors, evolve according to a dynamical system.

6.1 Going There and Back Again

The main observation is that expanding subspaces become contracting subspaces and contracting subspaces become expanding subspaces in the time-reversed flow [13, Lemma 1.2]. By the time-reversed flow, we mean that $\frac{\mathrm{d}x}{\mathrm{d}t} = F(x)$ becomes $\frac{\mathrm{d}x}{\mathrm{d}t} = -F(x)$, so trajectories move backwards in time relative to the original system.

Let f be the discretized flow map which integrates forward by some timestep h. The "types" on variations of f determine the cotangent dynamics. The inverse of f is easy to obtain, by replacing all occurrences of F in f with $-F$. The derivative of the inverse maps from $T_{f(x)}\mathcal{M}$, the tangent space at $f(x)$, to $T_x\mathcal{M}$, the tangent space at x. We will call the derivative of the inverse J^{-1}. The adjoint of the derivative of the inverse is a linear map from T_x^*, the cotangent space at x, to $T_{f(x)}^*$, the cotangent space at $f(x)$. We will call this $J^{-\top}$. Figure 6.1 shows the relationships between J, J^{-1}, $J^\top$, and $J^{-\top}$.

The last map $J^{-\top}$ propagates cotangent vectors along a *forward* primal trajectory but according to the time-*reversed* map. Thus, the most expanding cotangent direction corresponds to the most contracting tangent direction. This suggests that we can compute the smallest Lyapunov exponents by using a variation of Algorithm 1 that propagates covectors through $J^{-\top}$ instead of vectors through J. In $\mathbb{R}^n$, every cotangent vector φ corresponds with a tangent vector in the sense that there is a tangent vector u for which $\varphi(v) = u^\top v$ for

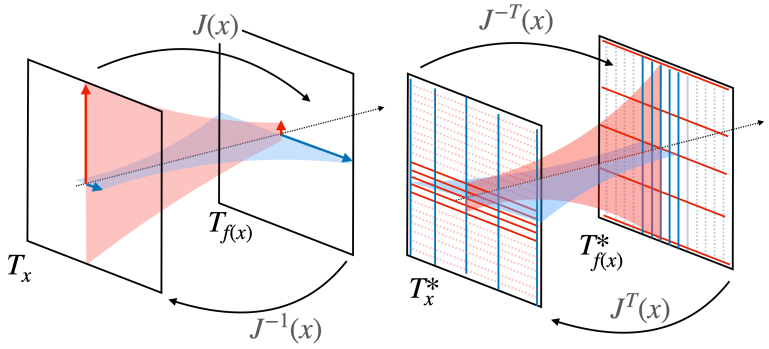

Figure 6.1: Duality between tangent and cotangent dynamics. Tangent vectors that expand and contract become cotangent vectors that contract and expand instead. The intuition is that to make vectors appear smaller, make the measuring devices bigger.

all vectors v[1]. Algorithm 2 describes the smallest Lyapunov exponent algorithm.

6.2 Discussion

With just the classical Benettin's algorithm, computing the kth Lyapunov exponent and subspace also means computing the first $k-1$ exponents. This is wasteful when we are only interested in contracting exponents. Algorithm 2 computes exponents k through n without computing the first $k-1$ exponents. When combined with Benettin's algorithm, this results in a new knob which is ignoring some chunk of exponents in the middle of the spectrum. It may be possible to trade computation time for some small exponential growth from middling exponents.

Combining tangent dynamics and cotangent dynamics can also be beneficial for computing the full spectrum of exponents. Figure 6.2 visualizes the idea. Tangent dynamics compute half the spectrum and cotangent dynamics compute the other half. The theoretical complexity of QR decomposition for an $m \times n$ matrix (where $m \geq n$) is $\mathcal{O}(mn^2)$. Thus, decomposing a single $n \times n$ matrix costs $\mathcal{O}(n^3)$, while decomposing two $n \times n/2$

[1]This is often stated as tangent vectors being "column vectors" and cotangent vectors being "row vectors".

Algorithm 2 m smallest Lyapunov exponents

Require: N (number of iterations), m (number of exponents)

Ensure: m smallest Lyapunov exponents $\lambda_{n-m+1}, \dots, \lambda_n$

$Q_0 \leftarrow \mathbf{I}^{m \times n}$ ▷ *initialize basis of m orthonormal cotangent vectors*

$\lambda \leftarrow \mathbf{0} \in \mathbb{R}^m$ ▷ *running estimate of Lyapunov exponents*

for $t \leftarrow 0$ to N **do**

  $P \leftarrow Q_t J_t^{-\top}$ ▷ *send tangent basis forward through inverse transpose Jacobian*

  $Q_{t+1}^{\top}, R \leftarrow \mathrm{QR}(P^{\top})$ ▷ *orthonormalize P*

  $\lambda \leftarrow \lambda - \ln|\mathrm{diag}(R)|$ ▷ *negate local growth to account for time-reversal*

return $\lambda/(N \cdot h)$

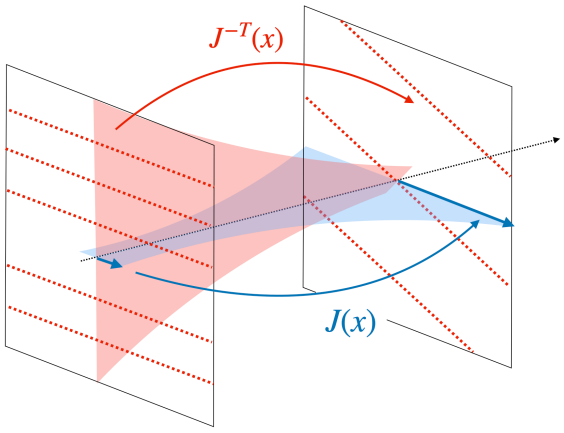

Figure 6.2: Tracking some directions with tangents and others with cotangents. In particular, the bottom half of contracting directions are tracked with cotangent vectors.

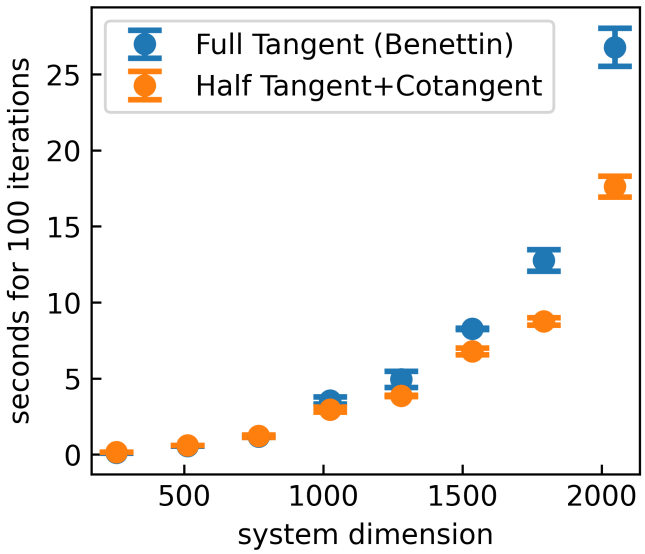

Figure 6.3: Combined tangent and cotangent performance improvement. We collect the time to estimate 100 local growth rates of the Lorenz 96 system [23] with varying system dimension over 7 runs. For large enough dimension, the cubic complexity QR decomposition begins to dominate the runtime.

matrices costs $\mathcal{O}(n^3/2)$. Since QR decomposition is the main bottleneck of both Lyapunov exponent computations and subspace computations, which are a component necessary for space splitting, we expect some performance improvement in certain scenarios. Namely, when computing the entire spectrum, just the contracting directions, or just some of the most expanding and some of the most contracting directions. Figure 6.3 shows a promising performance improvement when computing a full spectrum of the Lorenz 96 system [23].

One application of computing an entire Lyapunov spectrum of a chaotic system is to compute its Kaplan-Yorke dimension, which estimates the fractal dimension of the system's attractor [16]. The Kaplan-Yorke conjecture says that if j is the largest index for which

$$\sum_{i=1}^{j} \lambda_i \geq 0,$$

then the dimension of the attractor is

$$D := j + \frac{\sum_{i=1}^{j} \lambda_i}{|\lambda_{j+1}|}.$$

An intriguing connection which remains to be studied is the connection, if any, between cotangent dynamics and the density gradient used in the unstable contribution. Recall that for computing the stable contribution, we project the unstable directions out of the solution to a tangent equation. There is a tempting symmetry that the stable directions, which are unstable in the time-reversed dynamics, ought to describe something about the density gradient.

Part II

OOPS: A LANGUAGE FOR FORMALIZING FALLIBLE BIOLOGICAL PROTOCOLS

Chapter 7

AN INTRODUCTION TO PROTOCOL LANGUAGES AND Oops

Irreproducible biomedical research has been estimated to cost $28 billion a year [15]. Although biological experiments typically report their procedures in *protocols*, it remains extremely difficult to replicate results because inadequate descriptions cause ambiguity during execution. A study in cancer biology was only able to reproduce 50 out of 193 influential experiments even after receiving clarifications from the original experiments' authors [14]. Many of the experiments failed to replicate because they failed to match expected results over years of reproduction attempts. Lab technicians spend much of their time debugging experimental protocols because they often go wrong.

Protocols fail for a variety of reasons. The original procedure could have been confounded by an unrecognized factor, meaning that any replication attempt would be all but doomed without knowledge of that key factor. It is also possible that the technician accidentally left out a step or contaminated their workspace, or even that a random biochemical error occurred by chance. When protocols fail, it is important to determine what the cause was. A random genetic mutation is resolved by retrying the experiment, but systematic contamination requires all other experiments to be paused and lab equipment to be sanitized. To mitigate errors, practitioners have proposed protocol languages [9, 24, 5, 26, 25, 6, 3, 41, 32] that standardize biological protocols in code. Standardizing how protocols are documented reduces misinterpretation opportunities and supports robotic execution for large-scale experimentation and manufacturing.

Errors are still inevitable where biochemistry and humans are involved, despite formalized protocols and automation. However, existing protocol languages have limited if any support for specifying how protocols can go wrong. As shown by the cancer biology re-

producibility study [14], figuring out what error happened is challenging for unfamiliar protocols. On the biofoundry scale, human technicians must contend with errors made in unfamiliar protocols and with outcomes confounded by uncontrolled factors, all at high volume. The troubleshooting bottleneck limits the scalability of such services all while the number of novel protocols and demand for biomanufacturing and research grows [12]. Standardizing what the errors are that could affect a protocol outcome, how they could affect it, and how to determine which error may have occurred would both assist manual and enable automated troubleshooting efforts.

The goal of existing languages is to standardize what *should* happen when executing a protocol. As a result, protocols are fixed in three ways: the outcomes of individual steps, the outcomes of the entire protocol, and the steps taken in the protocol. But when formalizing what *could* happen, all of these need to deviate. On the scale of individual steps, biophysical processes and liquid transfers have inherent continuous variance. The outcome of a protocol varies depending on both variation within steps and mistakes made that deviate from the ideal protocol. In the larger experiment, the steps taken in a protocol are purposefully modified into *controls* to check for confounding errors. Writing down protocol variants that express all these combinations of errors and controls in deterministic protocol languages is repetitive at best and intractable at worst. Moreover, even if one did describe all these variants, writing down an outcome for every possible input would be infeasible.

Our insight is that a probabilistic protocol language with transformative metaprogramming makes specifying fallible protocols with their controls extremely concise. The probabilistic framework also enables reasoning powered by traditional probabilistic inference techniques.

We design a new language called Oops that represents biological protocols with probabilistic semantics. These semantics introduce errors on two different scales: Protocols can go wrong by mistakenly running steps drawn from a discrete distribution, and steps can go wrong by drawing their quantities, such as volumes and concentrations, from continuous distributions. These erroneous steps are mapped to erroneous outcomes through models

specified in a domain-specific language that abstracts the biology lab.

Probabilistic semantics alone can express fallible protocols, but the original intent becomes obfuscated. Furthermore, controls that purposefully vary a protocol still have to be written out separately. We observe that both a fallible protocol and controls are variants of the ideal protocol that can be automatically derived through transformative metaprogramming. We build a protocol metaprogramming system that separates error specification from the ideal specification and transforms protocols into control variants with user guidance.

Oops programs define distributions over experimental outcomes. Naturally, we would like to employ probabilistic inference to compute posterior distributions over errors conditioned on observed outcomes. Fortunately, observations in the lab are much like observations in existing probabilistic programming languages (PPLs). Observations in experiments can be made concurrently in control protocols running at the same time as the main protocol. Observations can also short circuit an experiment if it does not proceed as expected. Finally, observations may only happen at certain points in a protocol due to physical constraints. We develop a formalization of lab observations that casts Oops protocols as probabilistic models. We then describe probabilistic queries on these models and show how to compute them.

The main contributions of this work are:

- We design a probabilistic domain-specific language that specifies both what *should* happen and what *could* happen in a biological protocol (Chapter 9).

- We build a protocol metaprogramming system for concisely transforming a protocol into fallible and controlled variants. (Chapter 10). Controls help determine what *did* happen.

- We show how to map lab observations to conditioned Oops protocols and answer probabilistic queries on protocols (Chapter 11).

- We formalize a standard molecular cloning workflow and perform case studies of how

Oops could improve practitioners' understanding of errors in this workflow. (Chapter 12). Oops can not only guide an investigation into what went wrong but also help explore how a protocol can be designed to fail faster and provide more diagnostic information about failures.

Chapter 8
EXAMPLE: PCR IN Oops

The polymerase chain reaction (PCR) is a workhorse protocol of modern biology [39]. To motivate and introduce Oops, we'll show how to formalize a prototypical PCR protocol.

8.1 An Ideal PCR Amplification Protocol

The goal of a PCR protocol is to rapidly amplify a DNA segment of interest known as the *template*. The copying is done by DNA polymerase, an enzyme that synthesizes new DNA strands. This enzyme requires nucleotides (building blocks of DNA) and a buffer solution to operate, which are all combined into a *master mix*. Specific to the template are *forward* and *reverse primers*, which define the region to be amplified. The template, master mix, and primers are combined in a PCR machine, which repeatedly heats and cools the mixture to amplify the template.

Consider an amplification protocol in Oops:

```
@protocol
def amplify(F, R, T):
  M = Retrieve("master_mix")
  P = Create("pcr_result", volume=0 * uL)

  Transfer(M, P, 23.0 * uL)
  Transfer(F, P,  1.0 * uL)
  Transfer(R, P,  1.0 * uL)
  Transfer(T, P,  1.0 * uL)

  React(P, pcr)
  Store(M, P)
```

In Oops, which is embedded in Python, the function decorator `@protocol` marks a protocol. This `amplify` protocol is parameterized, which means it can be invoked from other protocols

with arguments for the forward primer (`F`), reverse primer (`R`), and template (`T`).

Variables such as `F` and `M` refer to *containers* [1]. At a given instant in time, the state of the lab can be viewed as the collective states of all containers, which we call the *inventory*. The body of a protocol is a list of *actions*, which are methods that transform an inventory. The `Retrieve` action in this protocol models a lab technician taking the `master_mix` container from a freezer and calling it `M`. Later, `M` (and `P`) is returned to the freezer using the `Store` action. Containers have an identifier and a temporary short name when on the lab bench.

After creating an empty container as `P` on the bench, liquids are transferred into `P`. Physical quantities in Oops are specified with units using the Pint [1] package. Now that `P` has all the ingredients necessary for the PCR reaction, the `React` action transforms `P` using an Oops *reaction* called `pcr`. This is a model of PCR specified by a function transforming a container that applies a basic limiting reaction model of PCR to the container's contents:

```
@reaction
def pcr(c):
  mm = c.get_conc("pcr_mm")
  tp = c.get_conc("pUC19")
  fw = c.get_conc("pUC19_Fw")
  rv = c.get_conc("pUC19_Rv")
  num_cycles = 30
  L = min(tp * 2**num_cycles, fw, rv, mm * 100)
  c.set_conc("pUC19", tp + limit)
  # Updating mix and primers omitted
```

The decorator `@reaction` registers this function as an Oops model that can be used by protocols. The model reads the concentration (`get_conc`) of master mix (`mm`), template (`tp`), and forward (`fw`) and reverse (`rv`) primers for the pUC19 plasmid to compute a final output concentration (`L`), which is written back to the concentration of the template (`set_conc`).

While this model is simplistic, it is easy in Oops to make a model more complex when the need arises. For example, we could modify `pcr` to explicitly model each PCR heating and cooling cycle, or to work on more species than just the pUC19 plasmid by adding containers to the starting inventory.

---

[1] The terse variable naming is intended to reflect lab naming conventions.

With the generic amplification protocol and model specified, we can define a full experimental protocol that amplifies the pUC19 template:

```
@protocol
def experiment():
  Fw = Retrieve("pUC19_Fw")
  Rv = Retrieve("pUC19_Rv")
  T = Retrieve("template")
  amplify(Fw, Rv, T)

  P = Retrieve("pcr_result")
  Observe(P.pUC19 > 0.01 * mg / uL)
  Store(F, R, T, P)
```

To invoke the `amplify` protocol we defined earlier, we call it like an ordinary Python function with container bench names as arguments. The `Observe` action both specifies the expected result of this experiment and conditions the protocol on the outcome.

8.2 Making Mistakes

One common source of error has nothing to do with molecular biology. It is retrieving the wrong sample from the freezer. A lab technician might retrieve a forward primer for a different plasmid with a similar label in the same box in the freezer. Oops expresses errors on which step was taken with probabilistically taken branches recorded by the `Either` action:

```
Either(
  Retrieve("pUC19_Fw", F),
  Retrieve("pBR322_Fw", F),
  0.99)
```

The correct primer is retrieved with probability 0.99 and the incorrect primer retrieved with probability 0.01. These probabilities can be calibrated by historical data.

If every line in `experiment` was augmented with a potential mistake, the intent of the original protocol would be hard to decipher. Oops provides a metaprogramming system to separate the intended protocol from the error-prone version and apply the error rewrites automatically.

```
experiment = edit(experiment,
  either({"name": "pBR322_Fw"},
```

```
        with_prob=0.01, at="Retrieve"))
```

The `edit` keyword begins a transformation. This transformation only has one operation, `either`, which replaces an action with an `Either` action that sometimes runs a variation of that action. In this case, the variation is that the container name being retrieved is replaced with `pBR322_Fw`. The target location is specified with the selector syntax `at="Retrieve"`, which selects the first `Retrieve` action. The result is an `Either` action identical to what we wrote by hand.

8.3 Creating Controls

A *control* is a variant of a protocol that controls for confounding factors. In our experiment, we want a *negative* control on the template to ensure that any output DNA comes from the template we put in. Negative means that we run a PCR reaction without the template. If such a negative control still amplifies the template, then we say it fails and consequently suspect contamination. Below we write out a negatively controlled version of `amplify`, with green text and red strikethroughs showing the difference from the original.

```
@protocol
def n_ctrl(F, R~~, T~~):
  M = Retrieve("master_mix")
  P = Create("pcr_result", volume=0 * uL)

  Transfer(M, P, 23.0 * uL)
  Transfer(F, P,  1.0 * uL)
  Transfer(R, P,  1.0 * uL)
  ~~Transfer(T, P, 1.0 * uL)~~

  React(P, pcr)
  Observe(P.pUC19 == 0)
  Store(M, F, R, P)
```

This looks similar to the original `amplify`. The template parameter is missing, along with actions that only take the template as an input. Protocol metaprogramming lets us derive the same negative control from the original protocol.

```
n_ctrl = edit(amplify,
```

```
    rename("n_ctrl"),
    remove_param("T"),
    insert("Observe(P.pUC19 == 0)", at="Store(M)") )
```

Here, we renamed the protocol to `"n_ctrl"` and inserted an expected observation of zero output. The `remove_param` transformation removes `T` from the list of parameters and also runs a dataflow analysis to remove actions dependent on `T`. Also note the selector syntax `"Store(M)"`, which selects a location inside the protocol to perform the edit. In this case, `"Store(M)"` looks for the first `Store` action with an `M`. See Section 10 for more details, including on how we also derive positive control variants.

Finally, we use more metaprogramming to insert this control into the experimental protocol right before the invocation of `amplify`:

```
experiment = edit(experiment,
  run(n_ctrl, at="amplify", args=("Fw", "Rv")) )
```

Again, we used a selector `"amplify"` that targets the invocation of the `amplify` subprotocol.

This amplification procedure is just the first part of the molecular cloning workflow we formalize in the evaluation (Section 12). There, we also demonstrate how Oops can determine how helpful this negative control is for figuring out the wrong primer error.

Chapter 9
THE Oops LANGUAGE

In this chapter, we define Oops formally, with the goal of using it to specify what *should* happen in a biological protocol and, just as importantly, what *could* happen. We will abstract the lab as a computing platform by distilling representations of data—containers (9.1) and inventories (9.2)—and of code that manipulates this data—actions (9.3), protocols (9.4), and reactions (9.5).

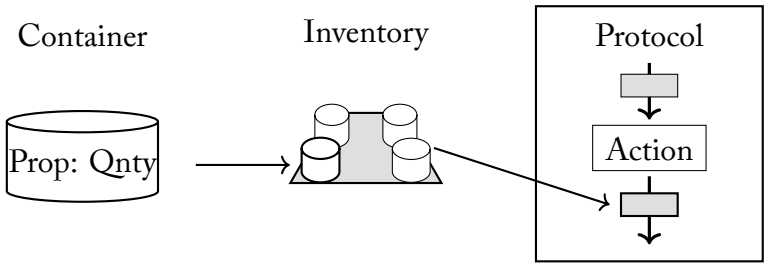

Figure 9.1: Protocols, actions, inventories, and containers.

9.1 Containers

Containers generalize test tubes, cuvettes, flasks, Petri dishes, and other vessels designed to hold, manipulate, and analyze biological materials. On the lowest level, Oops models containers as dictionaries keyed by *properties*, which are strings. Examples of properties include volume, DNA concentration, and number of cell colonies.

Oops makes a distinction between *mixtures*, which represent liquid mixtures with a volume and concentrations of species, and non-mixtures. Liquid transfer is not allowed from non-mixtures to mixtures. In the implementation, containers expose a basic API allowing

Inventory management	**Composition**
`Create`(name, Prop $\to$ Qnty)	`Run`(Prot, arguments)
`Store`(b)	`Steps`($A_1, \dots$)
`Retrieve`(name, b)	`Either`($(A_i, p_i), \dots$)
Container operations	**Observation**
`React`(b, Model)	`Observe`(b, Prop, Qnty $\to$ Bool)
`Transfer`(b_{src}, b_{dest}, Qnty, σ)	`Measure`(b, Prop, name)

Figure 9.2: Actions in Oops. b's are bench names, A's are actions, p's are probabilities, and σ's are standard deviations.

`get` and `set` operations while mixtures extend this interface to track concentration-valued properties and volume.

9.2 Inventories

An *inventory* represents a snapshot of the lab and consists of containers, observations that have been made, and other ambient properties, such as room temperature. Containers in an inventory can be stored in a freezer, or out on the workbench. Each container is identified by a full name and a short bench name that is temporarily assigned when placed on the bench during a protocol. For example, a test tube with a plasmid might be stored in a freezer with full name "pUC19-GFP-20240929", retrieved and referenced as simply "pG" in a bacterial transformation protocol, and then stored back in the freezer. Both full names and bench names exist in a global scope. Containers can be retrieved multiple times and even when on the bench, but may only be stored when on the bench.

9.3 Actions

An *action* is a method that represents a lab operation and describes how to transform an inventory with that operation. Actions are instantiated by parameters that dictate how the action behaves when called on an inventory, such as which bench name it should use or how much volume should be transferred. Actions are focused on producing results that can be interacted with, so Oops is not concerned with intermediate stages that are unobservable or that do not matter for the modeled set of errors and outcomes.

Figure 9.2 lists a core set of actions included in Oops. To give a flavor of how actions describe lab operations and how they can be composed into protocols, we describe them in more detail next. All actions must implement an interface that takes an inventory and returns an inventory.

Inventory management A surprisingly common error made in a biology lab is mislabeling or misreading containers, which is less surprising considering that storage tubes can be tiny, labels be hand written, and hundreds of tubes packed into a single box, hundreds of boxes packed into a single freezer, and dozens of freezers supporting a large lab. The `Create`, `Store`, and `Retrieve` actions express the management and mismanagement of containers.

Specifically, `Create` models the use of a fresh container that has been given a full identifier and placed on the bench. The `Retrieve` action represents retrieving a container from the lab store and placing it on the bench, while `Store` represents storing a container currently on the bench back into the store. Containers can be retrieved again once already on the bench, but can only be stored when on the bench and cannot be stored again without being retrieved first.

Composition The `Run` action runs an entire *protocol*, which we discuss in Section 9.4. A key action is `Either`, which is a nondeterministic branch. It is instantiated with a list of actions A_i and a list of probabilities p_i that sum to one. When called on an inventory, the `Either` action chooses one of its A_i to execute with probability p_i. The `Steps` action allows

for a sequence of actions to happen as a unit in `Either`.

The main purpose of `Either` is to express errors that can be made when executing a step, such as a mistaken retrieval. The probabilities p_i encode a prior on these mistakes.

Container operations Both `React` and `Transfer` model transformations of containers. The `React` action runs a model of a reaction on a container. We defer discussion of how Oops expresses models to Section 9.5.

The `Transfer` action transfers a volume of liquid from one container to another. When pipetting in the lab, the actual volume transferred depends on the user's skill and can be modeled as locally Gaussian centered at the desired volume with standard deviation σ corresponding to skill. When transferring small volumes, fluctuations can matter. What *should* happen is recovered by setting $\sigma = 0$.

To model the transfer of t units of liquid from container A to container B, we first set $u = t + \text{normal}(0, \sigma))$ and clamp u to be between 0 and the volume of A. Let $v_{\text{old}}(X)$ be the volume of X before the transfer and $v_{\text{new}}(X)$ be the volume after. The new volume of B is simply $v_{\text{new}}(B) = v_{\text{old}}(B) + u$, while the new volume of A is $v_{\text{new}}(A) = v_{\text{old}}(A) - u$. For any concentration property P in either A or B, let $c_{\text{old}}(X)$ be the concentration of P in X before the transfer and $c_{\text{new}}(X)$ be the concentration after. Then the new concentrations of P are

$$c_{\text{new}}(B) = \frac{c_{\text{old}}(B) \cdot v_{\text{old}}(B) + c_{\text{old}}(A) \cdot u}{v_{\text{old}}(B) + u} \tag{9.1}$$

and $c_{\text{new}}(A) = c_{\text{old}}(A)$.

Since Oops distinguishes between mixture and non-mixture containers, transfers must have a mixture as a source container. However, non-mixture containers can be destinations to model operations like transferring contents into a machine for analysis. For non-mixture destinations, we assume the existing volume to be zero.

Observation An inventory includes measurements in its state, modeling a lab technician's lab notebook recording a trace of their experiment. The `Measure` action records the quantity

associated with a property with a label so that it can be retrieved when analyzing a protocol trajectory.

`Observe` formalizes an expectation of what a property should be. It abstracts measurements that serve as verification steps that may short-circuit and halt the entire experiment. It also represents measurements taken in controls. In addition, `Observe` allows for conditioning, which we discuss further in Section 11. Its interface takes a container, a property P of that container, and an expected predicate f for that property. When the Oops framework executes an `Observe` (that is not in a control protocol), the program halts if the value of P does not satisfy f.

Both `Measure` and `Observe` abstract various measurements that can be taken in a lab, whether by eye, a machine, or an instrument. As an analogy, `Observe` actions are assertions, while `Measure` actions are print statements.

9.4 Protocols

A *protocol* is a list of actions that are executed sequentially. Oops enables composition and reuse of protocols through parameterized protocols, which means that protocols can be defined with container-valued parameters that are substituted for arguments at runtime. The `Run` action keeps track of arguments. Protocols have names and can be marked as controls, which distinguishes control protocols running concurrently from the main protocol.

We implement the Oops language as an embedded DSL in Python. The `@protocol` decorator parses a Python function AST into an Oops protocol AST, which is really a tree of actions. To connect action definitions to action instances created in the DSL, Oops provides an `@action` decorator that registers an action as a keyword and generates parsing code.

9.5 Reactions

To specify what outcome results from each path of actions taken through a protocol, Oops needs a model mapping inputs to an outcome. Otherwise, the user would have to specify

the outcomes of combinatorially many possible paths, each representing a different subset of mistakes. So far, the protocol language as described implements part of this model. Containers and actions like `Transfer` model the evolution of mixtures of species by tracking total volume and concentrations inside the mixture. However, nothing in this framework encodes how those species interact, which is the reason why anything interesting at all happens in the lab.

Oops allows the user to define the most interesting phenomena of the model, such as chemical reactions and cell growth, with *reactions*, which are functions that transform a single container to its state after a biological or chemical process runs to completion. Reactions are registered using the `@reaction` decorator and invoked using the `React` action. Since many processes run to completion before the next interaction with the container, usually, modeling the results of a reaction is sufficient. We purposefully do not aim to model interactions in full biophysical detail, since many biological processes do not have generic mathematical theories and modeling on first principles leads to infinite regress.

Exposing parts of the model definition to the user enables them to encode their own inductive reasoning about complex biological processes. Once the Oops model no longer reflects lab observations, the reactions can be updated to encompass new findings. This incorporates the *phenomenological* theories found in empirical lab work that combine with theories of underlying mechanisms to explain how outcomes come from inputs.

Chapter 10
PROTOCOL METAPROGRAMMING

The goal of this chapter is to derive protocols that specify what could happen and determine what did happen from a protocol that describes what should happen. To that end, we show how program transformation enables transforming protocols into positive and negative controls (Section 10.1) and separating the specification of what ideally happens from erroneous deviations through transformations (Section 10.2). We then build a metaprogramming language to express these transformations concisely (Section 10.3).

10.1 Metaprogramming Controls

There are two main types of controls. *Positive* controls confirm that experimental conditions are capable of producing results. *Negative* controls support hypotheses that results are due to a given experimental variable. Together, they separate the experimental outcome from confounding factors and verify that the setup works as intended. Both provide valuable information for determining what happened when a protocol goes wrong.

Given a protocol P and an experimental variable X in the protocol, running a negative control on X means executing P with a null version of X. Examples include omitting X in a reaction or using an inert substance in place of X. Running a positive control on X means executing P with other experimental variables fixed to values that are known to produce a positive result. In a PCR protocol, a negative control on the template means running a PCR without any template DNA to check for contamination, while a positive control on the primers fixes the template to a sequence the primers are known to amplify to check that the primers work.

We discovered that these methodological patterns correspond to systematic transforma-

```
@protocol
def neg_ctrl_A(A, B, C):
  Transfer(A, C)
  Transfer(B, C)
  React(C, model)
  Observe(C.prop == 0)
```

```
@protocol
def pos_ctrl_A(A, B, C):
  Retrieve(B, "good_B")
  Retrieve(C, "good_C")
  Transfer(A, C)
  Transfer(B, C)
  React(C, model)
  Observe(C.prop > 0)
```

Figure 10.1: Transforming a protocol into negative and positive controls. (**L**) Removing parameter `A` also removes the `Transfer` action using `A` as a source. (**R**) Specializing parameters `B` and `C` to fixed, known containers.

tions of code. Given an Oops protocol with some parameters, a negative control can be derived from the original by removing a parameter representing an experimental variable, eliminating actions dependent on that parameter along with actions dependent on those, and appending an observation of a negative result. Similarly, a positive control can be derived by replacing parameters outside of parameters to be controlled with constant values and appending an observation of a positive result. Figure 10.1 shows an example of these transformations on a simple protocol.

10.2 Encoding Errors with Metaprogramming

Protocol transformations also enable a workflow that allows users to separate the ideal protocol from that protocol with its controls and errors. This preserves the intent of the original protocol and makes it easy to iterate on both the protocol and its errors. Since controls are derived from the original protocol, changes to the original are automatically reflected in the controls. This is in the same spirit as user-schedulable languages [28, 18] except that transformations that add errors do not preserve semantics.

10.3 The Oops *Metaprogramming Language*

To express and compose code transformations, Oops exposes metaprogramming primitives that operate on protocols. Table 10.1 describes some of the primitives available. Sequences of transforms are applied to protocols by using the `edit` function, which has the interface `edit(protocol, ops...)`. This makes it easy to group related transforms and inspect intermediate programs.

To specify which action a transformation should target, we build a selector syntax. A selector with just the name of an action, such as `"Store"`, finds the first top-level `Store` action in a protocol. If there are multiple matching actions, a selector can be further specified with arguments in the action (such as a container name) and the names of any number of containing protocols. For example, a `Store(A)` action located in a subprotocol named `assemble` can be referenced with `"assemble(Store(A))"`.

Transform	Description
`remove_param(x)`	Remove `x` from the parameters and perform a forward dataflow analysis [4] to remove dependent actions.
`specialize(x, v)`	Remove `x` from the parameters of `p` and replace all `x`'s with `v`.
`insert(a, at)`	Insert action `a` before the first action matching `at`.
`replace(a, at)`	Replace the first action matching `at` with `a`.
`either(a, at, prob)`	Replace the first action a' matching `at` with `Either(a', a, prob)`.
`run(p, at, args)`	Insert protocol `p` invoked with `args` before the first action matching `at`.
`rename(n)`	Rename to `n`.

Table 10.1: Some of the metaprogramming operations in Oops.

Chapter 11
WHAT HAPPENED?

Recall that `Observe` actions function as assertions in protocols. We view these assertions as generalizations of two types of observations made in the lab while running protocols (Section 11.1). The joint distribution over errors and assertions leads to an interpretation of Oops protocols as probabilistic models (Section 11.2). We then define queries on these models motivated by questions about reproducibility (Section 11.3) and show how to compute these queries (Section 11.4).

11.1 Unifying Observations

In a software debugger, the entire state of the program is visible after every operation. In the lab, it is not possible or practical to exactly measure every physical property at every point. We focus on two main kinds of observations one might make while running a protocol:

Sanity checks Between some steps in a protocol, it is sensible to measure a quantity that confirms if the protocol is going as expected. For example, after amplifying template DNA using a qPCR machine, gel electrophoresis measures the concentrations of DNA fragments at different lengths. If the concentration at the length of the target DNA is nonexistent or low relative to other lengths, then something has gone wrong and it is not worth continuing forward.

Controls Other than the main protocol, a protocol might have subprotocols that are run concurrently (in the case of controls) or later (when troubleshooting). These protocols also result in observations that give meaningful information about the original protocol.

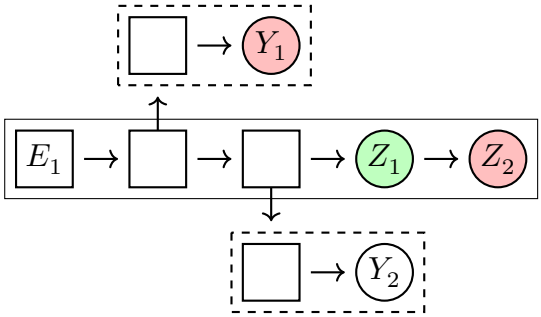

Figure 11.1: An Oops protocol viewed as a probabilistic model. The large solid box is the protocol and dashed boxes indicate controls. There is one potential error at E_1. Knowing that the main protocol failed at Z_2 determines that observation Z_1 succeeded. The control observation at Y_1 is known to have failed, but the control result Y_2 is unknown.

Both kinds of observations have pass/fail outcomes. What counts as a success or failure is determined by a property and a predicate that defines acceptable measurements of that property. In addition, both kinds of observations serve as breakpoints. A failed sanity check halts execution of the protocol, while a failed control observation provides evidence for a confounding factor. As a corollary, the knowledge that a sanity check occurred provides extra information that all previous sanity checks must have succeeded. Thus, the `Observe` action can represent both kinds of observations.

11.2 Oops *Protocols as Probabilistic Models*

An Oops protocol is a probabilistic model; a distribution over possible experiment outcomes. The random variables are discrete errors E_i, which come from `Either` actions, control assertions Y_i, which come from `Observe` actions in controls, and sanity check assertions Z_i, which come from `Observe` actions in the protocol. Figure 11.1 visualizes an example. An Oops protocol defines a joint distribution over all possible experiment outcomes

$$p(E_1, \ldots, E_n, Y_1, \ldots, Y_m, Z_1, \ldots, Z_k). \tag{11.1}$$

The domain of error E_i depends on the possible mistake actions and includes the possibility of not making a mistake. The domains of the Y_i are $\{\mathsf{T}, \mathsf{F}, \perp\}$, where $\perp$ represents an unobserved value. Since Z_i failing implies that every previous sanity check Z_j with $j < i$ must have passed, we represent the outcome of the main protocol with a single random variable Z whose domain is over the sanity checks along with a value T that indicates the entire protocol succeeded. In this way, every trace of an OOPS protocol has values defined for every variable.

11.3 Motivating Probabilistic Queries

Given the viewpoint of protocols as probabilistic models, we now pose probabilistic queries on those models that are motivated by questions about a protocol's reproducibility.

Error explanation Given a protocol, we would like to explain what went wrong. Knowing that a protocol went wrong means that some sanity check failed, so our query is for a conditional distribution over each error given the first failed check. We may also know the results of some controls depending on which were run either during or after the experiment. This leads to the posterior distribution

$$p(E_i \mid Y_1, \ldots, Y_m, Z), \tag{11.2}$$

where known control results Y_i are set to T or F and unknown results are set to $\perp$.

Mutual information Given an error, we would like to know if a control helps to explain the error. We can quantify the helpfulness with the *mutual information* between E_i, the error, and Y_j, the control outcome. From here, we omit the subscripts for clarity. The mutual information $I(E; Y)$ is defined as the difference in *entropy* between the prior of E and the posterior of E conditioned on Y; that is, $I(E; Y) = H(E) - H(E \mid Y)$. In turn, the entropy of a discrete distribution X is defined by $H(X) = \sum_{x \in \mathcal{X}} p_X(x) \log p_X(x)$. Entropy quantifies the amount of information needed to describe the state of X. If conditioning an

error on a control result leads to a distribution with less entropy than the original posterior, then the control provides information about that error.

To quantify the helpfulness of a control over all possible outcomes of a protocol, we consider the *conditioned* mutual information $I(E; Y \mid Z)$ where Z is the distribution over outcomes [45]. Recall that the domain of outcomes is all sanity check observations, where a failed observation halts the entire experiment. It can be shown that for jointly discrete random variables E, Y, and Z, Equation 11.3 holds.

$$I(E; Y \mid Z) = \sum_{z \in \mathcal{Z}} p_Z(z) \sum_{y \in \mathcal{Y}} \sum_{e \in \mathcal{E}} p_{E,Y|Z}(e, y \mid z) \log \frac{p_{E,Y|Z}(e, y \mid z)}{p_{E|Z}(e \mid z) p_{Y|Z}(y \mid z)}. \tag{11.3}$$

11.4 Computing Probabilistic Queries

All of the distributions of interest are manipulations of the joint distribution. In particular, we need to support marginalization and conditioning [40]. To compute these distributions, we use a sampling-based approach. This algorithm does not scale efficiently in the number of errors introduced to the protocol but is sufficient for our case studies.

In a preprocessing step, we collect an empirical distribution $\hat{p}$ by sampling the protocol P. Mistakes can be rare and multiple rare mistakes might have to occur for a failure outcome. As a result, we enumerate all choices of error possibilities and sample a separate empirical distribution for each. Let $\mathcal{C} = \mathcal{E}_1 \times \cdots \times \mathcal{E}_n$, where $\mathcal{E}_i$ are the possibilities made in error E_i. For every choice of errors and non-errors $c \in \mathcal{C}$, we fix a version of the protocol P_c that always and only makes the mistakes in c. Then, we collect an empirical distribution $\hat{p}(Y_1, \ldots, Y_m, Z \mid E = c)$ by collecting S samples $\{y_1^{(s)}, \ldots, y_m^{(s)}, z^{(s)}\}_{s=1}^{S}$. For clarity, we use $E = c$ as shorthand for the full assignment of each E_i to the choice c_i.

To answer queries, we provide a generic recipe for marginalization and conditioning.

Marginalization Suppose that we want to marginalize out all variables other than Z. First, we sum over all enumerated choices:

$$\widehat{p}(Z) = \sum_{c \in \mathcal{C}} \widehat{p}(Z \mid E = c) \cdot p(c). \tag{11.4}$$

The weight $p(c)$ can be calculated from the prior on the E_i defined by `Either` actions in the protocol. The marginalized distribution for a given choice is

$$\widehat{p}(Z = z \mid E = c) = \frac{1}{S} \sum_{i=1}^{S} [z = z^{(i)}]. \tag{11.5}$$

The process is similar for other marginalizations; we just keep the samples that agree in the remaining variables.

Conditioning Suppose that we want to condition on Z and we're interested in the posterior of one error E_i. By definition of the conditional distribution, we have

$$\widehat{p}(E_i \mid Z = z) = \frac{\widehat{p}(E_i, Z = z)}{\widehat{p}(Z = z)}. \tag{11.6}$$

Now, we just marginalize the joint distribution.

Chapter 12
FORMALIZING MOLECULAR CLONING IN Oops

We formalize an end-to-end molecular cloning workflow. Using Oops, we answer three questions about the cloning protocol motivated by our original problems:

A. What went wrong?

B. Does a control help find an error?

C. How quickly can an error condition be discovered?

12.1 Formalized Molecular Cloning

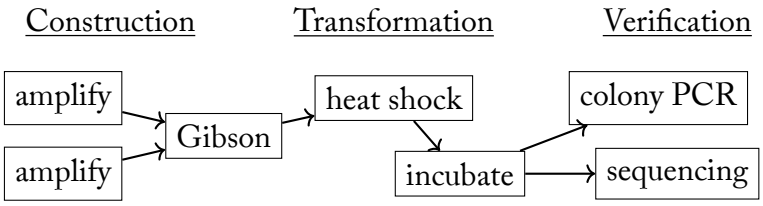

Figure 12.1: A flowchart of the molecular cloning protocols.

The goal of molecular cloning is to assemble DNA *molecules* and *clone* them into a population of cells [38]. First, in the construction phase, the DNA to be cloned is amplified by PCR and combined with vector DNA to form recombinant DNA molecules. Next, in the transformation phase, specially treated bacteria are induced to take up these molecules and replicate them. These bacteria replicate exponentially to generate a large number of copies of the original molecule. Finally, in the verification phase, the molecule is extracted from the cells and verified to have the DNA of interest.

Error	Description
Primer	Wrong or badly designed primer
Reagents	Degraded PCR master mix and pUC19 forward primer are used in an amplification
Assembly	Misassembly during Gibson assembly, including flipped inserts
Antibiotic	Wrong or missing antibiotic plated
Mixing	Harsh mixing breaks cells
Contam.	Competent cells contaminated with plasmid

Table 12.1: Errors introduced to our protocol.

Figure 12.1 shows a high level view of the workflow we use to evaluate Oops in this paper. The construction phase consists of two protocols. An amplification protocol like the one in Chapter 8 amplifies DNA coding for the green fluorescent protein (GFP), which is the DNA to be cloned, and a plasmid vector known as pUC19. A *plasmid* is a small, circular DNA molecule that replicates independently of a cell's chromosomal DNA. Then, a Gibson assembly protocol assembles the plasmid with GFP to produce recombinant molecules.

The transformation phase consists of a heat shock protocol and an incubation protocol. During heat shock, specially treated *E. coli* are suddenly heated and cooled to take up the recombinant plasmids. The cells are plated onto agar plates and incubated overnight to multiply. Since not every cell takes up a plasmid, transformed cells are selected for by adding an antibiotic to the plates that the pUC19 plasmid codes resistance for.

Finally, a technician selects a colony on the plate and scrapes it off for verification that the cells replicated the correct recombinant molecule. This involves two protocols: a colony PCR protocol, which is a quick verification that the lengths of replicated DNA match what is expected, and a sequencing protocol, which is expensive but checks that the DNA exactly matches what is expected.

Control	Description
+GFP	Positive control on GFP amplification
+Vector	Positive control on pUC19 amplification
-Plasmid	Transform cells without adding plasmid
+Cells	Positive control on cell transformation with known plasmid

Table 12.2: Controls available in our protocol.

We also introduce the errors shown in Table 12.1 and control protocols shown in Table 12.2.

12.2 Explaining What Happened

An early sign something has gone wrong in a molecular cloning pipeline is that the number of colonies on the plate incubated overnight is lower than expected. We use Oops to build an interactive "debugger" for the molecular cloning model that could guide a technician towards where to look.

The debugger can be queried with a failed observation:

```
>>> debugger.explain("xform(Observe(Plate))")

Explanations for 25 <= colonies <= 150 in Plate:
  Error 1:
    Degrade("Fw", "pUC19_Fw", ...): 0.75
    Skip(): 0.25
  ... (remaining errors omitted)
```

Explaining a failed observation means conditioning the protocol at that observation and inspecting the posterior distributions on errors. Here, we've conditioned on an `Observe` measuring the number of colonies in a container labeled `Plate`. In return, we get the marginal likelihood of the choices in every error, reported in decreasing order. According to the model, we might want to first check for degraded reagents that would cause low efficiency

amplification and subsequently low efficiency transformation. The "control unknown" chart in Figure 12.2 visualizes initial conditional likelihoods for three errors.

Now suppose that during the protocol, a positive control on the competent cells during transformation was run (this is called "+Cells" in Table 12.2). This control attempts to transform cells with a plasmid known to work well, checking that the cells can be transformed in the first place. If the positive control passes but the transformation did not work well, then there must be an issue with the actual plasmid being cloned. On the other hand, if the positive control fails, then something must have happened to the cells.

Suppose the positive control passed. We can ask the Oops debugger to condition on the control succeeding:

```
>>> debugger.condition(
      "xform_pos_p(Observe(Plate))", True)
>>> debugger.explain("xform(Observe(Plate))")

Explanations for 25 <= colonies <= 150 in Plate:
  Error 1
    Degrade("Fw", "pUC19_Fw", ...): 1.00
  ...
```

The model suggests that the reagents must have been degraded, leading to low transformation efficiency even though the control passed. Similarly, we can condition on a failed positive control. Figure 12.2 visualizes how conditioning on the positive control result affects the likelihoods of different errors. By adding more known observations and ruling out errors, the Oops debugger can guide a technician in their investigation of what went wrong. If all errors in the model are accounted for, then the actual cause of failure must be outside the model.

12.3 Information Gain from Controls

A control provides a baseline to compare the primary experimental results against. Ideally, a control should eliminate alternate explanations so that it is certain that observed effects are due to a manipulated variable. Since controls are usually executed concurrently with the

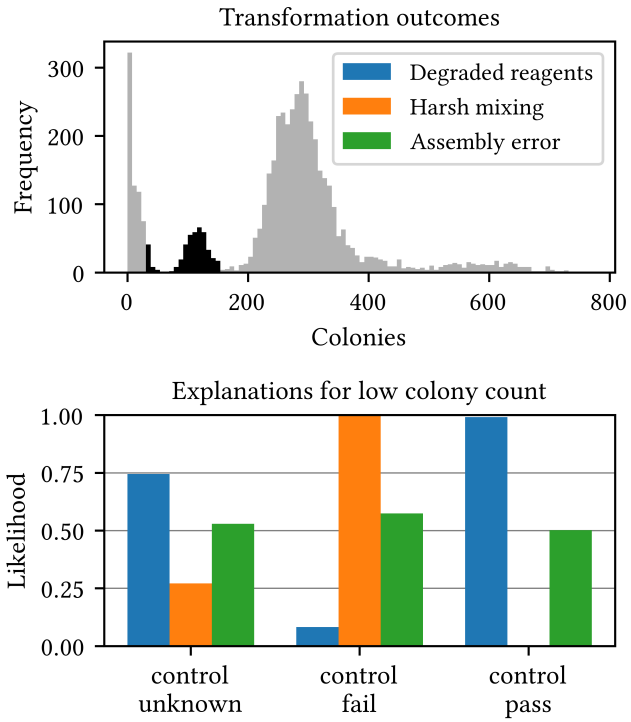

Figure 12.2: Explaning what went wrong. Top: Colony count outcomes of the bacterial transformation stage. Bottom: Explaining three types of error conditioned on observing a low number of colonies (highlighted region in the top) and the outcome of a control.

main protocol, one design question asks if it is worth running a control. Given an error of interest, we can quantify how helpful a control is for figuring out if the error happened by studying the *mutual information* between the random variable that the error happened and the random variable that the control passed or failed. We define mutual information formally in Section 11.3. Informally, mutual information quantifies the information gained about the error given the control result.

For example, focusing on posteriors for degraded reagents in Figure 12.2, conditioning on either control result concentrates more mass towards either the reagents being degraded or not being degraded, corresponding to information gain.

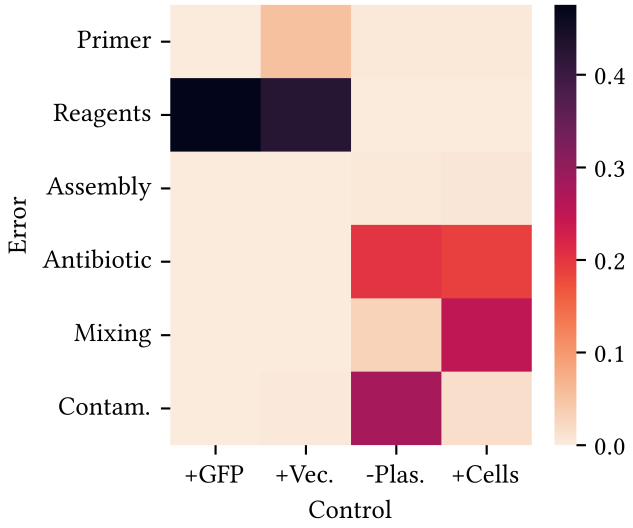

Figure 12.3: Conditional mutual information visualized for every pair of error and control. A dark color for a given error and control indicates high expected information gain in the error when the control outcome is known. A bright color means that the error and control are largely independent.

We use Equation 11.3 to compute conditional mutual information between every error and control implemented in our molecular cloning protocol (in tables 12.1 and 12.2). Figure 12.3 visualizes the results. It suggests that every control contributes bits of information towards every error other than misassembly.

12.4 Failing Fast

In large scale experimental workflows executed in industrial biofoundries, massively parallel cloning processes may take more than a week to complete, at significant cost. It is beneficial when executing this protocol to know that something has gone wrong before completing the entire process, and the earlier the better. If the protocol is going to fail, it better fail fast. One way to fail faster is to add verification steps, such as controls, at the cost of making the

entire protocol longer on successful runs. We analyze the impact of adding two different verification steps to our molecular cloning model to see how it could fail faster.

One step is to verify that the lengths of amplified DNA match the lengths of what is expected when amplifying the vector and insert. Each piece of DNA has a length measured in the number of base pairs making up the strand. In a protocol called gel electrophoresis, the output from a PCR is placed in a gel and molecules of different lengths are separated by an electric current. This checks that the amplification reaction worked as expected.

Another verification step we explore is a set of controls on the bacterial transformation. These are the "-Plasmid" and "+Cells" controls described in Table 12.2. This set of controls checks that the bacterial transformation works as expected.

Using Oops metaprogramming, we create two variants of the original molecular cloning protocol with the verification steps inserted. Then, we simulate 10,000 runs of each conditioned on a failed end result to collect distributions over the first failed observation (shown in Figure 12.4). The progress at an observation O is defined by

$$\frac{\#\text{ actions up to } O}{\#\text{ actions in the protocol}}. \tag{12.1}$$

Combinator actions such as `Either` and actions run in control protocols are not included in these counts.

Introducing the gel step fails a small number of executions early in the protocol, as shown by the orange line entering the plot (Figure 12.4) at a nonzero height. However, the gel introduces extra work before the rest of the protocol can be finished. So, the next step that can fail occurs later in the protocol. According to the Oops model, a more effective method of failing fast are the controls on transformation, represented by the green line. Since transformation controls can be run concurrently with the original transformation, this protocol fails faster.

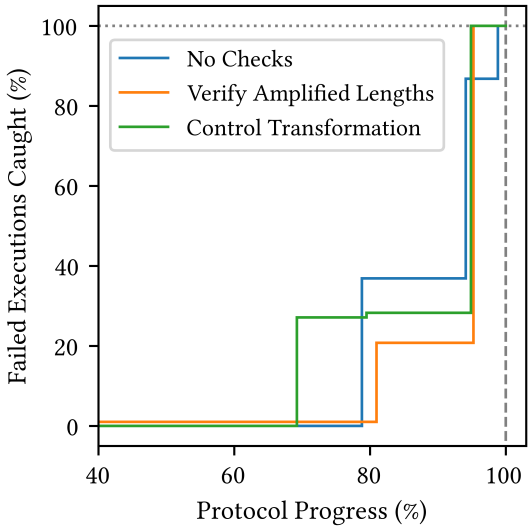

Figure 12.4: Failing fast. Where and how likely three molecular cloning protocol designs are to fail first, given that they ultimately fail to clone the insert.

Chapter 13
RELATED WORK

Protocol Languages Previous protocol languages focus mainly on standardizing what should happen in the protocol. On one end, languages compile to natural language descriptions meant for humans to execute, such as in protocols.io [37] and BioCoder [5]. On the other end, protocols are formalized to a level of detail that allows robots and microfluidic devices to execute them, such as Autoprotocol [24], Opentrons [25], and Puddle [44]. Other languages such as BioScript [26] and LabOP [6] have representations for both humans and machinery to execute. The key contribution of Oops is to introduce another axis of formalizing what else could happen when running a protocol.

Abate et al. [3] and followup work by Cardelli et al. [9] design a language with probabilistic semantics to characterize uncertainties. While they propagate stochasticity through a deterministic sequence of protocol operations, our language makes the operations executed stochastic as well. However, their Gaussian semantics enable an efficient estimator for gradients through simulated protocols.

Automation Systems Lab automation systems such as Aquarium [41], Benchling [2], and Synthace's Antha [32] combine interfaces for standardizing protocols with electronic notebooks for recording data. They also schedule operations to be executed by humans or robots, forming an "operating system" for the lab. To the best of our knowledge, existing systems focus on standardizing what should happen in protocols. While historical execution data recorded in these systems can be mined for historical failure reasons, their standardizations do not encode a priori assumptions that could inform a lab manager or be used for inference. The probabilistic error semantics from Oops could be integrated into these

systems' protocol interfaces, which are often no-code.

Although Aquarium and Benchling are intended more for organization than prediction, Antha is capable of executing protocols *in silico*, meaning through digital simulation. With a model of underlying processes, integrating Oops into Antha could enable both error inference and fitting error parameters such as probabilities and uncertainties to real execution traces.

Puddle [44] can autonomously recover from errors made when executing protocols on digital microfluidic devices. By detecting when droplet positioning diverges from what is expected, Puddle synthesizes instructions to reset to an error-free state. Microfluidic automation systems like Puddle could execute subprotocols in Oops and eliminate sources of error for Oops to worry about.

Metaprogramming and Domain-Specific Languages Oops builds on previous metaprogramming languages and domain-specific languages to achieve concise and flexible specification of biological protocols. Much like prior generative metaprogramming systems [36, 27, 18, 21], Oops programs include metaprograms that transform ASTs to reduce clutter in the original programs and automate repeated rewrites. The Oops language is designed around the lab as an unconventional computing platform and allows exploration of protocol design. Recent work has explored languages for exploring the design space of other unconventional computing methods such as analog circuits in Ark [42].

Chapter 14
CONCLUSION

Summary

In Part I, we addressed the inverse problem for swarming models, specifically the boids algorithm. We identified that naively differentiating through boids simulations fails due to the fundamental chaotic nature of swarm dynamics. We then motivated results from ergodic dynamics and chaos theory to understand a recent algorithm for estimating chaotic derivatives. After developing a smooth swarming model expressible as an ODE, we applied the theory to the model for early results towards derivatives through boids.

In Part II, we developed Oops, a programming language for formalizing biological protocols with errors. Integrating a probabilistic abstraction of the biology lab with a protocol metaprogramming system enabled concise expression of fallible protocols. Combined with probabilistic conditioning and queries, we showed how Oops answers diagnostic questions about a real-world molecular cloning protocol, including what went wrong, does a control help, and how quickly can an error be discovered.

Both parts of this thesis tackled inverse problems in complex models simulating the real world. In Part I, although we had to reformulate the model to be mathematically rigorous, the main challenge was in the mechanism for going backwards through the model. We took a different angle in Part II, where the mechanism for going backwards was already known (probabilistic inference), but the model itself needed augmentation to express inverse possibilities. The common threads are domain-specific specialization of general-purpose programming language paradigms and utilizing probabilistic methods. In both parts, general-purpose programming languages with automatic differentiation and probabilistic inference already exist, but they had to be augmented with domain-specific chaotic system handling

and biological constructs for efficient use. A probablistic viewpoint also showed up in both problems, since probability is the natural language for dealing with distributions over uncertainty. Despite the fact that boids are a deterministic system, chaos means that long-term behavior is effectively random.

Limitations and Future Work

While we made partial progress towards solving the inverse boids problem, the derivative and optimization both remain unsolved. Immediate future work includes adapting the center and unstable contributions to boids. In the best case, this is mainly an engineering problem of making expensive computations run with a reasonable time and memory budget. In the worst case, there may be theoretical obstacles preventing the linear response from existing, such as a severe lack of hyperbolicity. It is possible that gradient-free methods are best suited for optimizing swarming simulations.

It might also be possible to avoid chaos in swarming when taking long time averages by relaxing the optimization problem. For example, the parameters of each boid could be allowed to vary independently, and instead of being concerned with all possible starting states, we could just focus on one trajectory. A loss could focus on shaping just this trajectory towards a control path. Inverse design methods for swarming would streamline artistic design and enable biological modeling. If these methods involve derivatives, the chaotic nature of swarming means that such methods could also apply to other important chaotic systems, such as fluid simulations and weather prediction.

Oops protocols are not executable, because they do not encode all the information necessary to be physically run by humans or robots. One action in Oops can map to multiple real lab operations. One protocol represents what can happen to a single sample, whereas a real protocol batches together many samples to execute operations on simultaneously. Oops could be lowered to a finer-grained protocol language used by an existing biofoundry. Implementing a system like Oops in a biofoundry would enable a feedback loop between the model and the real-world process.

Failing fast (Section 12.4) is only one metric to explore the design space of protocols. With a formalized model of how a protocol can fail, attaching costs (in time or money) to each action would unlock simulations of the cost to achieve successful outcomes and variance in product. By searching over the space of protocols with different controls and verification steps enabled, Oops could optimize costs.

Finally, the ideal feedback loop between Oops and reality would allow Oops to learn priors on errors from real execution traces. This would likely require a fully automated pipeline carrying out protocols at mass scale to determine failure probabilities from experience rather than setting them based on intuition.

Both problems work towards a computational model of the natural world, from the chaotic dynamics underpinning weather, fluids, and swarms, to the scientific process we use to experiment.

BIBLIOGRAPHY

[1] Base API — pint 0.24.3.dev1+g79e636c documentation. https://pint.readthedocs.io/en/stable/api/base.html.

[2] Online Lab Notebook (ELN) Software | Benchling. https://www.benchling.com/notebook.

[3] Alessandro Abate, Luca Cardelli, Marta Kwiatkowska, Luca Laurenti, and Boyan Yordanov. Experimental Biological Protocols with Formal Semantics, May 2018.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. Compilers: Principles, Techniques, & Tools. Pearson Addison-Wesley, Boston Munich, 2. ed., pearson internat. ed edition, 2007.

[5] Vaishnavi Ananthanarayanan and William Thies. Biocoder: A programming language for standardizing and automating biology protocols. Journal of Biological Engineering, 4(1):13, November 2010.

[6] Bryan Bartley, Jacob Beal, Miles Rogers, Daniel Bryce, Robert P. Goldman, Benjamin Keller, Peter Lee, Vanessa Biggers, Joshua Nowak, and Mark Weston. Building an Open Representation for Biological Protocols. ACM Journal on Emerging Technologies in Computing Systems, 19(3):1–21, July 2023.

[7] Giancarlo Benettin, Luigi Galgani, Antonio Giorgilli, and Jean-Marie Strelcyn. Lyapunov Characteristic Exponents for smooth dynamical systems and for hamiltonian systems; A method for computing all of them. Part 2: Numerical application. Meccanica, 15(1):21–30, March 1980.

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[9] Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. A Language for Modeling and Optimizing Experimental Biological Protocols. Computation, 9(10):107, October 2021.

[10] Andrea Cavagna, Irene Giardina, Alberto Orlandi, Giorgio Parisi, Andrea Procaccini, Massimiliano Viale, and Vladimir Zdravkovic. The STARFLAG handbook on collective animal behaviour: Part I, empirical methods, February 2008.

[11] Nisha Chandramoorthy and Qiqi Wang. Efficient computation of linear response of chaotic attractors with one-dimensional unstable manifolds, January 2022.

[12] Douglas Densmore, Nathan J. Hillson, Eric Klavins, Chris Myers, Jean Peccoud, and Giovanni Stracquadanio. Introduction to the Special Issue on BioFoundries and Cloud Laboratories. ACM Journal on Emerging Technologies in Computing Systems, 19(3):1–2, July 2023.

[13] Luca Dieci, Robert D. Russell, and Erik S. Van Vleck. On the Computation of Lyapunov Exponents for Continuous Dynamical Systems. SIAM Journal on Numerical Analysis, 34(1):402–423, 1997.

[14] Timothy M Errington, Alexandria Denis, Nicole Perfito, Elizabeth Iorns, and Brian A Nosek. Challenges for assessing replicability in preclinical cancer biology. eLife, 10:e67995, December 2021.

[15] Leonard P. Freedman, Iain M. Cockburn, and Timothy S. Simcoe. The Economics of Reproducibility in Preclinical Research. PLOS Biology, 13(6):e1002165, June 2015.

[16] Peter Grassberger and Itamar Procaccia. Measuring the strangeness of strange attractors. Physica D: Nonlinear Phenomena, 9(1):189–208, October 1983.

[17] Alp Uzman (https://math.stackexchange.com/users/169085/alp uzman). What is a "physical measure"? Mathematics Stack Exchange. URL: https://math.stackexchange.com/q/4493352 (version: 2022-07-15).

[18] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 703–718, San Diego CA USA, June 2022. ACM.

[19] Pavel V. Kuptsov. Fast numerical test of hyperbolic chaos. Physical Review E, 85(1):015203, January 2012.

[20] Pavel V. Kuptsov and Ulrich Parlitz. Theory and Computation of Covariant Lyapunov Vectors. Journal of Nonlinear Science, 22(5):727–762, October 2012.

[21] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. ACM Comput. Surv., 52(6):113:1–113:39, October 2019.

[22] Edward N. Lorenz. Deterministic Nonperiodic Flow. March 1963.

[23] Edward N. Lorenz. Predictability – a problem partly solved. In Renate Hagedorn and Tim Palmer, editors, Predictability of Weather and Climate, pages 40–58. Cambridge University Press, Cambridge, 2006.

[24] Ben Miles and Peter L. Lee. Achieving Reproducibility and Closed-Loop Automation in Biological Experimentation with an IoT-Enabled Lab of the Future. SLAS Technology, 23(5):432–439, October 2018.

[25] Opentrons. Ot-2 python protocol api version 2. https://docs.opentrons.com/v2/index.html.

[26] Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. BioScript: Programming safe chemistry on laboratories-on-a-chip. Proceedings of the ACM on Programming Languages, 2(OOPSLA):1–31, October 2018.

[27] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, June 2013. Association for Computing Machinery.

[29] Craig Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. SIGGRAPH Comput. Graph., 21(4):25–34, 8 1987.

[30] Craig Reynolds. Boids (Flocks, Herds, and Schools: A Distributed Behavioral Model). https://www.red3d.com/cwr/boids/, July 2007.

[31] David Ruelle. A review of linear response theory for general differentiable dynamical systems. Nonlinearity, 22(4):855–870, April 2009.

[32] Michael I. Sadowski, Chris Grant, and Tim S. Fell. Harnessing QbD, Programming Languages, and Automation for Reproducible Biology. Trends in Biotechnology, 34(3):214–227, March 2016.

[33] Samuel Schoenholz and Ekin Dogus Cubuk. JAX MD: A Framework for Differentiable Physics. In Advances in Neural Information Processing Systems, volume 33, pages 11428–11441. Curran Associates, Inc., 2020.

[34] Adam A. Śliwiak, Nisha Chandramoorthy, and Qiqi Wang. Ergodic sensitivity analysis of one-dimensional chaotic maps. Theoretical and Applied Mechanics Letters, 10(6):438–447, November 2020.

[35] Adam Andrzej Sliwiak. Leveraging the Linear Response Theory in Sensitivity Analysis of Chaotic Dynamical Systems and Turbulent Flows. Thesis, Massachusetts Institute of Technology, June 2023.

[36] Venkat Subramaniam. Programming Groovy 2: Dynamic Productivity for the Java Developer. The Pragmatic Programmers. The Pragmatic Bookshelf, Dallas, Texas, second edition edition, 2013.

[37] Leonid Teytelman, Alexei Stoliartchouk, Lori Kindler, and Bonnie L. Hurwitz. Protocols.io: Virtual Communities for Protocol Development and Discussion. PLOS Biology, 14(8):e1002538, August 2016.

[38] Rebecca Tirabassi. Foundations of Molecular Cloning - Past, Present and Future. https://www.neb.com/en-us/tools-and-resources/feature-articles/foundations-of-molecular-cloning-past-present-and-future.

[39] Billyana Tsvetanova, Lansha Peng, Xiquan Liang, Ke Li, Jian-Ping Yang, Tony Ho, Josh Shirley, Liewei Xu, Jason Potter, Wieslaw Kudlicki, Todd Peterson, and Federico Katzen. Chapter fourteen - Genetic Assembly Tools for Synthetic Biology. In Christopher Voigt, editor, Methods in Enzymology, volume 498 of Synthetic Biology, Part B, pages 327–348. Academic Press, January 2011.

[40] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming, October 2021.

[41] Justin Vrana, Orlando de Lange, Yaoyu Yang, Garrett Newman, Ayesha Saleem, Abraham Miller, Cameron Cordray, Samer Halabiya, Michelle Parks, Eriberto Lopez, Sarah Goldberg, Benjamin Keller, Devin Strickland, and Eric Klavins. Aquarium: Open-source laboratory software for design, execution and data management. Synthetic Biology, 6(1):ysab006, January 2021.

[42] Yu-Neng Wang, Glenn Cowan, Ulrich Rührmair, and Sara Achour. Design of Novel Analog Compute Paradigms with Ark. In Proceedings of the 29th ACM International Conference on Architectural Support for

Programming Languages and Operating Systems, Volume 2, volume 2 of ASPLOS '24, pages 269–286, New York, NY, USA, April 2024. Association for Computing Machinery.

[43] Amie Wilkinson. What are Lyapunov exponents, and why are they interesting? Bulletin of the American Mathematical Society, 54(1):79–105, September 2016.

[44] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. Puddle: A Dynamic, Error-Correcting, Full-Stack Microfluidics Platform. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183–197, Providence RI USA, April 2019. ACM.

[45] A. D. Wyner. A definition of conditional mutual information for arbitrary ensembles. Information and Control, 38(1):51–59, July 1978.

[46] Lai-Sang Young. What are srb measures, and which dynamical systems have them? Journal of Statistical Physics, 108:733–754, 01 2002.

[47] Lai-Sang Young. Mathematical theory of Lyapunov exponents. Journal of Physics A: Mathematical and Theoretical, 46(25):254001, June 2013.

Appendix A
BOID PARAMETERS

Unless stated otherwise, data was collected from the following parameters:

Parameter	**Value**
`num_boids`	12
`box_size`	150
D_{align}	25
D_{avoid}	15
D_{cohere}	20
J_{align}	1
J_{avoid}	10
J_{cohere}	1000
s	50
h	$1/120$
k	3

Table A.1: Default boid simulation parameters.

Appendix B

CODE

B.1 Benettin's Algorithm

```
def jvp(x, v, run_params):
  f_wrapped = lambda y: f(y, run_params)
  return jax.jvp(f_wrapped, (x,), (v,))[1]

def lyapunov_step(state, run_params):
  x, Q = state
  P = jax.vmap(jvp, (None, 0, None))(x, Q.T, run_params).T
  Q, R = jnp.linalg.qr(P)
  le = jnp.log(jnp.abs(jnp.diag(R)))
  return (f(x, run_params), Q), le

@partial(jax.jit, static_argnames=("run_params", "num_le", "steps_transient",
    "steps_main"))
def spectrum(key, run_params, num_le, steps_transient, steps_main):
  boids = init(run_params.box_size, run_params.boid_count, key)
  # boids.pos.shape = (boid_count, dim)

  x0 = jnp.hstack([boids.pos.flatten(), boids.theta])  # (boid_count * dim +
      boid_count)
  ndims = len(x0)  # System dimension
  Q0 = jnp.eye(ndims, num_le, dtype=x0.dtype)

  scan_fn = lambda carry, _: lyapunov_step(carry, run_params)
  x_Q, _  = jax.lax.scan(scan_fn, (x0, Q0), length=steps_transient)
  x_Q, le = jax.lax.scan(scan_fn, x_Q     , length=steps_main)
  # le.shape = (num_steps, k, num_le)
  le /= run_params.dt
  step_indices = jnp.arange(1, le.shape[0] + 1)[..., jnp.newaxis]
  les = jnp.cumsum(le, axis=0) / step_indices
  return les

def spectrum_parallel(key, batch_size, run_params, num_le, steps_transient,
    steps_main):
```

```
    keys = jax.random.split(key, batch_size)
    if not IN_COLAB:
        assert batch_size % jax.device_count() == 0
        keys = jax.device_put(keys, NamedSharding(mesh, PartitionSpec('C')))
    les = jax.vmap(spectrum, in_axes=(0, None, None, None, None))(keys,
        run_params, num_le, steps_transient, steps_main)
    return les
```

B.2 Stable Contribution

```
def df_dtheta(x, a, run_params):
    g = lambda x, a0: f(x, a + a0, run_params)
    return jax.jacfwd(g, argnums=1)(x, 0.)

def jvp(x, v, a, run_params):
    wrapped_f = lambda y: f(y, a, run_params)
    return jax.jvp(wrapped_f, (x,), (v,))[1]

State = namedtuple("State", ["s", "le", "x", "v", "Q", "R"])

def reduced_step(state, run_params, a, stat_fn, warmup=False):
    s = state.s
    le = state.le
    if not warmup:
        wrapped_S = lambda y: stat_fn(y, run_params)
        s += jax.jvp(wrapped_S, (state.x,), (state.v,))[1]
        le += jnp.log(jnp.abs(jnp.diag(state.R)))
    P = jax.vmap(jvp, (None, 0, None, None))(state.x, state.Q.T, a,
        run_params).T
    Q, R = jnp.linalg.qr(P)
    r = jvp(state.x, state.v, a, run_params) + df_dtheta(state.x, a,
        run_params)
    c = Q.T @ r
    v = r - Q @ c
    x = f(state.x, a, run_params)
    return State(s, le, x, v, Q, R), state

@partial(jax.jit, static_argnames=("run_params", "stat_fn", "num_unstable", "
    steps_transient", "steps_main"))
def stable_sens(key, run_params, a, stat_fn, num_unstable, steps_transient,
    steps_main):
    boids = init(run_params.box_size, run_params.boid_count, key)
    x0 = jnp.hstack([boids.pos.flatten(), boids.theta])
```

```
    ndims = len(x0)
    Q0 = jnp.eye(ndims, num_unstable, dtype=x0.dtype)

    warmup_fn = lambda _, carry: reduced_step(carry, run_params, a, stat_fn,
        warmup=True)[0]
    le0 = jnp.zeros(num_unstable)
    v0 = jnp.zeros_like(x0)
    Q0 = jnp.eye(ndims, num_unstable)
    R0 = jnp.eye(num_unstable)
    sens = State(s=0., le=le0, x=x0, v=v0, Q=Q0, R=R0)
    sens = jax.lax.fori_loop(0, steps_transient, warmup_fn, sens)

    scan_fn = lambda carry, _: reduced_step(carry, run_params, a, stat_fn,
        warmup=False)
    sens, state = jax.lax.scan(scan_fn, sens, length=steps_main)
    state = state._replace(s=state.s / jnp.arange(1, steps_main + 1))
    return state

@partial(jax.jit, static_argnames=("batch_size", "run_params", "stat_fn", "
    num_unstable", "steps_transient", "steps_main"))
def stable_sens_parallel(key, batch_size, run_params, a, stat_fn,
    num_unstable, steps_transient, steps_main):
    keys = jax.random.split(key, batch_size)
    # vmapped version seems to be faster -- uncomment for parallel
    # if not IN_COLAB:
    #     assert batch_size % jax.device_count() == 0
    #     keys = jax.device_put(keys, NamedSharding(mesh, PartitionSpec('C'))
        )
    sens_parallel = jax.vmap(stable_sens, in_axes=(0, None, None, None, None,
         None, None))
    sens = sens_parallel(keys, run_params, a, stat_fn, num_unstable,
        steps_transient, steps_main)
    return sens
```