George Datseris and Ulrich Parlitz

Nonlinear Dynamics

A concise introduction interlaced with code

Tuesday 27th July, 2021

Springer

Contents

Part I Introduction

1 Dynamical systems 3
1.1 What is a dynamical system? 3
1.1.1 Some example dynamical systems 4
1.1.2 Trajectories, flows, uniqueness and invariance 6
1.1.3 Notation 7
1.1.4 Nonlinearity 7
1.2 Poor man's definition of deterministic chaos 8
1.3 Computer code for dynamical systems 9
1.3.1 Associated repository: tutorials, exercise data, apps . 10
1.3.2 A notorious trap 10
1.4 The Jacobian, linearized dynamics and stability 11
1.5 Dissipation, attractors, and conservative systems 13
1.5.1 Conserved quantities 15
1.6 Poincaré surface of section and Poincaré map 16

2 Non-chaotic continuous dynamics 21
2.1 Continuous dynamics in 1D 21
2.1.1 A simple model for Earth's energy balance 21
2.1.2 Preparing the equations 22
2.1.3 Graphical inspection of 1D systems 23
2.2 Continuous dynamics in 2D 24
2.2.1 Fixed points and limit cycles 24
2.2.2 Self-sustained oscillations, limit cycles and phases . . 25
2.2.3 Finding a stable limit cycle 26
2.2.4 Nullclines and excitable systems 26
2.3 Poincaré-Bendixon theorem 30

2.4 Quasiperiodic motion 30

Part II Methods

3 Defining and measuring chaos 37
3.1 Sensitive dependence on initial conditions 37
3.1.1 Largest Lyapunov exponent 38
3.1.2 Predictability horizon 39
3.2 Fate of state space volumes 40
3.2.1 Evolution of an infinitesimal uncertainty volume 40
3.2.2 Lyapunov spectrum 41
3.2.3 Properties of the Lyapunov exponents 43
3.2.4 Essence of chaos: stretching and folding 44
3.2.5 Distinguishing chaotic and regular evolution 45

4 Bifurcations and routes to chaos 51
4.1 Bifurcations 51
4.1.1 Hysteresis 52
4.1.2 Local bifurcations in continuous dynamics 53
4.1.3 Local bifurcations in discrete dynamics 54
4.1.4 Global bifurcations 55
4.2 Numerically identifying bifurcations 56
4.2.1 Orbit diagrams 56
4.2.2 Bifurcation diagrams 58
4.2.3 Continuation of bifurcation curves 59
4.3 Some universal routes to chaos 61
4.3.1 Period doubling 61
4.3.2 Intermittency 63

5 Entropy and fractal dimension 69
5.1 Information and entropy 69
5.1.1 Information is amount of surprise 69
5.1.2 Formal definition of information and entropy 70
5.1.3 Generalized entropy 70
5.2 Entropy in the context of dynamical systems 71
5.2.1 Amplitude binning (histogram) 71
5.2.2 Nearest neighbor kernel estimation 72
5.3 Fractal dimension of sets 73
5.3.1 Fractals and fractal dimension 74
5.3.2 Chaotic attractors and self-similarity 75
5.3.3 Fractal basin boundaries 75
5.3.4 Mathematical and real-world fractals 76

5.4 Estimating the fractal dimension 77
5.4.1 Why care about the fractal dimension? 79
5.4.2 Practical remarks on estimating the dimension 79
5.4.3 Impact of noise 80
5.4.4 Lyapunov (Kaplan-Yorke) dimension 81
5.5 Information generation from chaotic dynamics 81

6 Delay coordinates 87
6.1 Getting more out of a timeseries 87
6.1.1 Delay coordinates embedding 88
6.1.2 Theory of state space reconstruction 89
6.2 Finding optimal delay reconstruction parameters 90
6.2.1 Choosing the delay time 91
6.2.2 Choosing the embedding dimension 92
6.3 Advanced delay embedding techniques 94
6.3.1 Spike trains and other event-like timeseries 94
6.3.2 Generalized delay embedding 94
6.3.3 Unified optimal embedding 95
6.4 Some nonlinear timeseries analysis methods 95
6.4.1 Nearest neighbor predictions (forecasting) 95
6.4.2 Largest Lyapunov exponent from a sampled trajectory 96
6.4.3 Permutation entropy 97

7 Information across timeseries 101
7.1 Mutual information 101
7.2 Transfer entropy 103
7.2.1 Practically computing the transfer entropy 104
7.2.2 Excluding common driver 105
7.3 Dynamic influence, causality 106
7.3.1 Convergent cross mapping 107
7.4 Surrogate timeseries 109
7.4.1 A surrogate example 110

Part III Applications

8 Billiards, conservative systems and ergodicity 117
8.1 Dynamical billiards 117
8.1.1 Boundary map 118
8.1.2 Mean collision time 119
8.1.3 The circle billiard (circle map) 119
8.2 Chaotic conservative systems 120
8.2.1 Chaotic billiards 120

8.2.2 Chaotic scattering 121
8.2.3 Mixed state space 122
8.2.4 Conservative route to chaos: the condensed version . 123
8.3 Ergodicity and natural measure 123
8.3.1 Some practical comments on ergodicity 124
8.4 Recurrences 126
8.4.1 Poincaré recurrence theorem 126
8.4.2 Kac's lemma 126
8.4.3 Recurrence quantification analysis 126

9 Periodically forced oscillators and synchronization of dynamical systems 133
9.1 Nonlinear resonances and bifurcation structure of periodically driven passive oscillators 133
9.1.1 Resonance curves 133
9.1.2 Symmetry breaking bifurcations 135
9.1.3 Structure of the bifurcation set 137
9.2 Synchronization of self-sustained oscillators 138
9.2.1 Periodically driven van der Pol oscillator 138
9.2.2 Circle map 140
9.3 Synchronization of chaotic systems 140
9.3.1 Identical sychronization and transversal instabilities 140
9.3.2 Chaotic phase synchronization 140
9.3.3 Generalized synchronisation of uni-directionally coupled systems 140
9.3.4 Phase diagrams 140

10 Complex networks and epidemics 143
10.1 Networks 143
10.1.1 Relevant network properties 143
10.1.2 Random networks of the real world 144
10.2 Network topology reconstruction 144
10.3 Complex networks for timeseries analysis 144
10.4 Modelling epidemics 144
10.4.1 The SIR model 144
10.4.2 Impact of network topology 144

11 Pattern formation and spatio-temporal chaos 147
11.1 Spatiotemporal systems and pattern formation 147
11.1.1 Reaction diffusion systems: The Brusselator 147
11.1.2 Linear stability analysis of extended systems 148
11.1.3 Wave number selection due to boundary conditions . 149
11.1.4 Oscillatory patterns (in the Brusselator) 150

11.1.5 Numerical solution of PDEs using finite differences . 151
11.2 Excitable media and spiral waves 153
11.2.1 The Fitzhugh-Nagumo model 153
11.2.2 Phase singularities 154
11.3 Spatio-temporal chaos 156
11.3.1 Extensive Chaos - KS-equation 156
11.3.2 Chaotic spiral waves and cardiac arrhythmias 156

12 Nonlinear dynamics of weather and climate 159
12.1 Hyper-complex systems, chaos & prediction 159
12.2 Tipping points in dynamical systems 162
12.2.1 Tipping mechanisms 162
12.2.2 Basin stability and resilience 165
12.2.3 Tipping probabilities 167
12.3 Nonlinear dynamics applications in climate 167
12.3.1 Excitable carbon cycle & extinction events 167
12.3.2 Climate attractors 169

References 175

Index 189

Preface

When we started writing this textbook, there were two goals we wanted to attain. The first goal was having a small, but up to date textbook, that can accompany a lecture (or a crash-course) on nonlinear dynamics with heavy focus on practical application. The target audiences therefore are on one hand students wanting to start working on applications of nonlinear dynamics, and on the other hand researchers from a different field that want to use methods from nonlinear dynamics in their research. Both groups require a concise, illustrative summary of the core concepts and a much larger focus on how to apply them to gain useful results.

The second goal for this book is to emphasize the role of computer code and explicitly include it in the presentation. As you will see, many (in fact, most) things in nonlinear dynamics are not treatable analytically. Therefore, most published papers in the field of nonlinear dynamics run plenty of computer code to produce their results. Unfortunately, even to this day, most of these papers do not publish their code. This means that reproducing a paper, or simply just implementing the proposed methods, is typically hard work, while sometimes even impossible. Textbooks do not fare much better either, with only a couple textbooks on the field providing and discussing code. We find this separation between the scientific text and the computer code very unfitting for the field of nonlinear dynamics, especially given how easy it is to share and run code nowadays. We believe that the best way to solve the problem is teaching new generations to embrace code as part of the learning subject, and this is exactly what we are trying to do with this book.

In a nutshell

There are many textbooks on nonlinear dynamics, some of which are exceptionally good theoretical introductions to the subject. We think our

approach will add something genuinely new to the field and bridge theory with real-world usage due to being:

- *practical*. We discuss quantities and analyses that are beneficial to compute in practice. Since just defining a quantity is typically not equivalent with computing it, we also provide algorithms and discuss limitations, difficulties and pitfalls of these algorithms.
- *full of runnable code*. The algorithms and figures are accompanied with real, high-quality runnable code snippets *in-text*. Real code eliminates the inconsistencies of pseudo-code, as well as the hours (if not days) needed to implement it, while also enabling instant experimentation.
- *up-to-date*. Each chapter of the book typically highlights recent scientific progress on the subject of the chapter. In addition, the last chapters of the book are devoted to applications of nonlinear dynamics in topics relevant to the status quo, for example epidemics and climate.
- *small*. This book is composed out of 12 chapters, and we enforced ourselves that each chapter is small enough that it can be taught during a standard lecture week (we tested it!).

Structure

At the beginning of each chapter a summary is presented (abstract), also pointing out where and why the concepts of the chapter are useful. The main text then firstly explains the basic concepts conceptually but still accurately, without going too deep into the details. We then proceed to discuss handpicked topics, algorithms, applications and practical considerations. A "Further reading" section then provides historical overviews and relevant references as well as sources where one can go into more depth regarding the discussed concepts. Each chapter ends with selected exercises, some analytic while some applied, which aid understanding and sharpen practical skills.

We thematically divided this book into three parts: introduction, methods, and applications. The "methods" are in general detached from the specifics of what the underlying dynamical system could be, and discuss concepts that are of broad scope and of general value for all kinds of dynamical systems. These methods are routinely used in research regarding dynamical systems. The "applications" are in fact not that much different from the methods, since we typically still introduce new theoretical and practical concepts in each chapter. The difference is that we discuss subjects that are a bit more specific in scope, but this discussion is done in an applied manner. A specific class of dynamical systems is discussed in more detail, where the subject of relevance is applied to analyse or understand

this class. For example, in Chap. 12 we treat climate as a dynamical system and use this opportunity to introduce and use concepts such as tipping points, which are also important in other fields like ecology or economy.

Usage in a lecture hall

The precise size of the book was chosen on purpose so that each chapter spans a lecture week, and typical semesters span 12 to 13 weeks. Thus it covers a full introductory course on nonlinear dynamics on an undergraduate or graduate program, depending on the student background. We strongly recommend active teaching, where students are involved as much as possible in the lecture, e.g. by presenting multiple choice questions during the course. We also strongly recommend to run code live during the lecture, use the interactive applications we have developed in parallel with this book (see Sect. 1.3.1) and solve exemplary exercises. Pay special attention to the exercises that are oriented towards actively writing and using computer code. This also teaches being critical of computer results and being aware of common pitfalls.

The first two parts of the book should be taught in the sequence they are written, as later chapters have some dependence on previous chapters. From the third part ("applications"), the lecturer can choose arbitrarily the presented order (or may even choose to exchange some of them with discussing e.g. papers on nonlinear dynamics applications on the relevant context of the course).

While preliminaries on calculus, linear algebra, probability, differential equations and basic familiarity with programming are assumed throughout the book, its light mathematical tone requires only the basics. As a sidenote, we have purposely written the book in a casual tone, even adding jokes here and there. This is not only in line with our characters "in real life", but also because we believe it makes the book more approachable. We hope you enjoy it, and have fun with it (as much fun as one can have with a textbook anyways)!

Hamburg and Göttingen,
July, 2021

George Datseris
Ulrich Parlitz

Acknowledgements

We would like to thank several expert reviewers that helped us ensure the rigor of each of our chapters. Specifically, Michael Wiltzek for Chapters 1 and 3. All contributors of the JuliaDynamics organization, especially those who have contributed to the DynamicalSystems.jl library, a list of which can be found online.

Part I

Introduction

1

Dynamical systems

Summary. This introductory chapter defines dynamical systems, stresses their widespread applications, and introduces the concept of "deterministic chaos". Since computer simulations are valuable and even necessary for studying nonlinear dynamics we also show in this chapter examples of runnable code snippets that will be used throughout the book. We then look at fixed points, and when they are stable. To do so we employ linear stability analysis and subsequently discuss how volumes in the state space grow or shrink. We close by reviewing the Poincaré surface of section, a technique to convert a continuous system into discrete ones, useful for visualizing higher-dimensional dynamics.

1.1 What is a dynamical system?

The term "dynamical system" can describe a wide range of processes and can be applied in seemingly all areas of science. Simply put, a dynamical system is a set of variables, or quantities, whose values change with time according to some predefined rules. These rules can have stochastic components, but in this book we will be considering systems without random parts, i.e., *deterministic dynamical systems*.

The variables that define the system are formally called the *state variables* of the system, and constitute the *state* or *state vector* $\mathbf{x} = (x_1, x_2, \ldots, x_D)$. For example, state variables can be the positions and velocities of planets moving in a gravitational field, the electric current and voltage of an electronic circuit, temperature and humidity of a weather model, to name a few. The space that $\mathbf{x}$ occupies is called the *state space* or *phase space* $\mathcal{S}$ and has dimension[1] D, with D the amount of variables that compose $\mathbf{x}$.

[1] Typically the state space $\mathcal{S}$ is an Euclidean space $\mathbb{R}^D$. But in general, it can be any arbitrary D-dimensional manifold. For example if some variables are by nature periodic, like the angle of a pendulum, the state space becomes toroidal.

Dynamical systems can be classified according to the nature of time. In this book we will be mainly considering two classes of dynamical systems, both having a continuous state space. This means that each component of $\mathbf{x}$ is a real number[2]. The first class is called *discrete* dynamical system

$$\mathbf{x}_{n+1} = f(\mathbf{x}_n) \tag{1.1}$$

with f being the *dynamic rule* specifying the temporal evolution of the system. Here time is a discrete quantity, like steps, iterations, generations or other similar concepts. At each time step f takes the current state and maps it to the next state. Discrete systems are usually given by iterated maps as in Eq. (1.1), where the state $\mathbf{x}_n$ is plugged into f to yield the state at the next step, $\mathbf{x}_{n+1}$. In our discussion here and throughout this book all elements of the vector-valued function f are real. Any scenario where f inputs and outputs complex values can be simply split into more variables containing the real and imaginary parts, respectively.

In *continuous* dynamical systems

$$\dot{\mathbf{x}} \equiv \frac{d\mathbf{x}}{dt} = f(\mathbf{x}) \tag{1.2}$$

time is a continuous quantity and f takes the current state and returns the rate of change of the state. Continuous systems are defined by a set of coupled ordinary differential equations (ODEs). f is also sometimes called *vector field* in this context. Equation (1.2) is a set of *autonomous* ordinary differential equations, i.e. f does not explicitly depend on time t. This is the standard form of continuous dynamical systems. In general, non-autonomous systems can be written as autonomous systems with an additional state variable corresponding to time. We will demonstrate this extension in Chap. 9 when discussing the dynamics of periodically driven oscillators. Furthermore, Eq. (1.2) contains only first order time derivatives. All expressions with higher order derivatives (e.g., $\ddot{x} = ...$, obtained in Newtonian mechanics) can be re-written as first-order systems by introducing new variables as derivatives, e.g., $y = \dot{x}$. Notice the fundamental difference between continuous and discrete systems: in the latter f provides the direct change, while in the former f provides the rate of change.

1.1.1 Some example dynamical systems

To put the abstract definition of dynamical systems into context, and also motivate them as tools for studying real world scenarios, let's look at some of the dynamical systems that we will be using in the subsequent chapters of this book. We visualize some of them in Fig. 1.1.

The simplest system, and arguably the most famous one, is the logistic map, which is a discrete system of only one variable defined as

[2] In Chap. 11, we will look at spatiotemporal systems, where each component of $\mathbf{x}$ is an entire spatial field that is evolved in time.

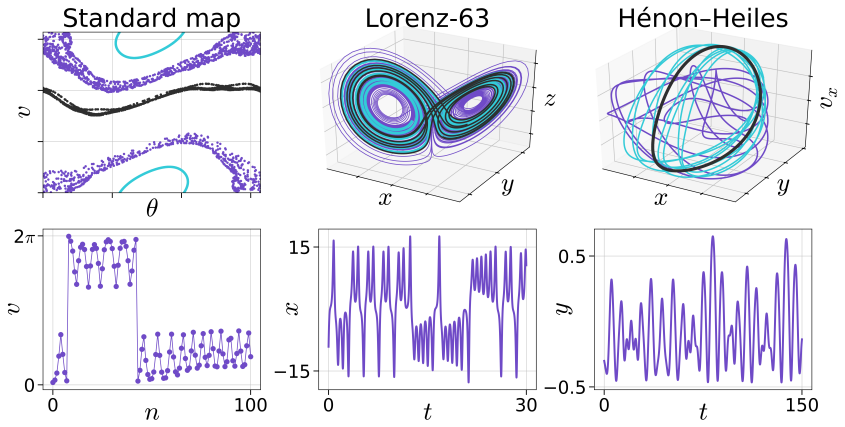


Fig. 1.1: Top row: state space of example systems (for Hénon-Heiles only 3 of the 4 dimensions are shown). Three different initial conditions (different colors) are evolved and populate the state space. Bottom row: example timeseries of one of the variables of each system (only one initial condition is used). Each column is a different system, evolved using Code. 1.1. Animations and interactive applications for such plots can be found online at animations/1/trajectory_evolution.

$$x_{n+1} = rx_n(1 - x_n), \quad r \in [0, 4], \quad x \in [0, 1]. \tag{1.3}$$

This dynamical equation has been used in different contexts as an example or a prototype of a simple system that displays very rich dynamics nevertheless, with a prominent role in population dynamics. There it describes how a (normalized) population x of, let's say, rabbits changes from generation to generation. At each new generation n the number of rabbits increases because of rx, with r a growth factor (which is of course the available amount of carrots). But there is a catch; if the rabbit population becomes too large, there aren't enough carrots to feed all of them! This is what the factor $(1 - x)$ represents. The higher the population x_n at generation n, the more penalty to the growth rate.

Another well-studied prototypical two dimensional discrete system is the standard map, that is given by

$$\begin{aligned} \theta_{n+1} &= \theta_n + v_n + k\sin(\theta_n) \\ v_{n+1} &= v_n + k\sin(\theta_n) \end{aligned} \tag{1.4}$$

with θ the angle and v the velocity of an oscillating pendulum on a tabletop (gravity-free). The pendulum is periodically kicked at every time unit by a force of strength k ($k = 1$ unless noted otherwise), which has a specific direction (thus the term sin). You may wonder why this is a discrete system when clearly an oscillating pendulum is operating in continuous time. The reason is simple: since the kick is applied only at each step n, and the motion between kicks is free motion (linear dynamics), we only need to record the momentum and angle values at the time of the kick.

Yet another famous dynamical system, continuous in this case, is the Lorenz-63 system, given by

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= -xz + \rho x - y \\ \dot{z} &= xy - \beta z. \end{aligned} \tag{1.5}$$

It is a simplified model for atmospheric convection, representing a two-dimensional fluid layer. x is the rate of convection and y and z are the temperature variations in the first and second dimension of the fluid, respectively. The parameters ρ, β, σ are related to the properties of the fluid like its viscosity (we use $\rho = 28, \beta = 8/3, \sigma = 10$ unless noted otherwise).

The last system that we list here is the Hénon–Heiles system

$$\begin{aligned} \dot{x} &= v_x, \quad & \dot{v}_x &= -x - 2xy \\ \dot{y} &= v_y, \quad & \dot{v}_y &= -y - (x^2 - y^2). \end{aligned} \tag{1.6}$$

This is a simplification of the motion of a star (represented by a point particle with positions x, y and velocities v_x, v_y) around a galactic center (positioned at $(0, 0)$). Although we will discuss how to simulate dynamical systems in detail in Sect. 1.3, let's see how these systems "look like" when evolved in time. In Fig. 1.1 we show both the state space of some systems (populated by evolving three different initial conditions) as well as an example timeseries of one of the variables.

It may seem that these examples were handpicked to be as diverse as possible. This is not at all the case and we were honest with you in the introduction. The framework of dynamical systems applies in seemingly arbitrary parts of reality.

1.1.2 Trajectories, flows, uniqueness and invariance

A *trajectory* (commonly also called an *orbit*) represents the evolution from an initial condition $\mathbf{x}_0$ in the state space. *Fixed points*, defined by $f(\mathbf{x}^*) = \mathbf{x}^*$ (discrete dynamics) or $f(\mathbf{x}^*) = 0$ (continuous dynamics), are technically trajectories that consist of a single point, i.e., a state that never changes in time. Besides these fixed points, for discrete systems a trajectory consists of a finite or infinite, but ordered, number of points. For continuous systems, trajectories are curves in the state space, which are closed in case of periodic oscillations.

In most cases (and for all systems we will consider in this book), the Picard-Lindelöf theorem states that for each initial condition $\mathbf{x}(t = 0) = \mathbf{x}_0$ a unique solution $\mathbf{x}(t)$ of the ODE(s) (1.2) exists for $t \in (-\infty, \infty)$ if the vector field f fulfills a mild smoothness condition, called Lipschitz continuity[3] which is met by most systems of interest. Only in cases where the solution diverges to infinity in finite time (that will not be considered in this book) $\|\mathbf{x}(t \to T_\pm)\| \to \infty$, the existence limits are finite, $t \in (T_-, T_+)$. The uniqueness property implies that

[3] A function $f : \mathcal{S} \to \mathcal{S}$ is called Lipschitz continuous if there exists a real constant $k \geq 0$ so that $\|f(\mathbf{x}) - f(\mathbf{y})\| \leq k\|\mathbf{x} - \mathbf{y}\|$ for all $\mathbf{x}, \mathbf{y} \in \mathcal{S}$. Every continuously differentiable function is Lipschitz continuous.

any trajectory $\{\mathbf{x}(t) \in \mathcal{S} : -\infty < t < \infty\}$ never intersects itself (except for fixed points or periodic orbits) or trajectories resulting from other initial conditions, i.e., the time evolution is unique.

For discrete systems forward-time uniqueness is always guaranteed for any f, regardless of continuity, simply from the definition of what a function is. To have the same feature backward in time f has to be invertible, in contrast to continuous systems where this is not required.

In simulations (or with measured data) trajectories of continuous systems are sampled at discrete time points which creates a set of states $A = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_N\}$. If only a single component of the state vectors $\mathbf{x}_n$ or a scalar function of the states is considered this provides a *uni-variate timeseries* or just *timeseries*. Measuring several observables simultaneously yields a *multivariate timeseries*.

Another concept which is useful for describing the temporal evolution is the *flow* $\Phi^t(\mathbf{x})$, a function mapping any state $\mathbf{x}$ to its future ($t > 0$) or past ($t < 0$) image $\mathbf{x}(t) = \Phi^t(\mathbf{x})$ (here t could also be n, i.e. the same concept holds for both continuous or discrete systems). Note that $\Phi^0(\mathbf{x}) = \mathbf{x}$ and $\Phi^{t+s}(\mathbf{x}) = \Phi^t(\Phi^s(\mathbf{x})) = \Phi^s(\Phi^t(\mathbf{x}))$. An *invariant set* is by definition a set A which satisfies $\Phi^t(A) = A$, $\forall t$. This means that the dynamics maps the set A to itself, i.e., it is "invariant" under the flow. Invariant sets play a crucial role in the theoretical foundation of dynamical systems.

1.1.3 Notation

In the remainder of this book we will typically use capital letters A, X, Y, Z, Ω to denote sets (sampled trajectories are ordered sets), bold lowercase upright letters $\mathbf{a}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \boldsymbol{\omega}$ to notate vectors, and lowercase letters a, x, y, z, ω to notate timeseries, variables or constants. A few exceptions exist, but we believe there will be no ambiguities due to the context. The symbols f, D, p, P are used exclusively to represent: the dynamic rule (also called equations of motion or vector field), the dimensionality of the state space or set at hand, an unnamed parameter of the system, a probability. Accessing sets or timeseries at a specific *index* is done either with subscripts x_n or with brackets $x[n]$ (n is always integer here). The syntax $x(t)$ means "the value of x at time t" in the case where t is continuous time, and it means that x is a function of t. The symbol $\|\mathbf{x}\|$ means the norm of $\mathbf{x}$, which depends on the metric of $\mathcal{S}$, but most often is the Euclidean norm. Set elements are enclosed in brackets $\{\}$.

1.1.4 Nonlinearity

A linear equation (or system) f by definition satisfies the superposition principle $f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$, $\alpha, \beta \in \mathbb{R}$ and thus any two valid solutions can be added to generate a new solution. Furthermore, the norm of the state $\mathbf{x}$ is irrelevant, as a solution can be given for any scale by changing α, and thus only its orientation matters. Because of these two properties linear dynamical systems can be solved in closed form and do not exhibit any complex dynamics.

You may have noticed that the rules f that we have described in Sect. 1.1.1 are *nonlinear* functions of the state $\mathbf{x}$. In contrast to linear systems, nonlinear systems must be *considered as a whole* instead of being able to be split into smaller and simpler parts, and on a *scale-by-scale basis*, as different scales are dominated by different dynamics within the same system. Therefore, nonlinear systems display a plethora of amazing features and dominate in almost all natural processes around us. That is why it is crucial for nonlinear dynamics to be studied and understood by scientists of all disciplines, from physics and chemistry to economy and social sciences.

1.2 Poor man's definition of deterministic chaos

The systems we are considering in this book are nonlinear and *deterministic*. Given an initial condition and the rule f we are in theory able to tell everything about the future of the system by evolving it forward in time. But one must not confuse deterministic systems with simple or easy-to-predict behaviour. While sometimes indeed deterministic dynamics will lead to simple, periodic behavior, this is not always the case as illustrated in the bottom row of Fig. 1.1.

Interestingly, the timeseries shown *never repeat themselves* in a periodic manner[4], even though the systems are deterministic! On the other hand, the timeseries aren't random either, as they seem to be composed of easily distinguishable patterns. For example, in the case of the standard map, one sees that sequences of points form triangles, and they are either near 2π pointing down or near 0 pointing up. Similar patterns exist in the Lorenz-63 and Hénon-Heiles model, where there are oscillations present, but they are non-periodic. It feels intuitive (and is also precisely true) that these patterns have a dynamic origin in the rule of the system f. Their exact sequence (i.e., when will the triangles in the standard map switch from up to down) may appear random, however, and sometimes may indeed be statistically equivalent to random processes.

This is called *deterministic chaos*: when the evolution of a system in time is non-periodic, apparently irregular, difficult to predict, but still deterministic, full of patterns which result in *structure in the state space*, as is seen clearly in the top row of Fig. 1.1. The sets that are the result of evolving a chaotic system (or more precisely, an initial condition that leads to chaotic dynamics in that system) will be called *chaotic sets* in this book. The fact that these sets have structure in the state space is the central property that distinguishes deterministic chaos from pure randomness: even if the sequence (timeseries) appears random, in the state space there is structure (of various forms of complexity). We will discuss deterministic chaos in more detail in Chap. 3.

[4] This might not be obvious from the small time window shown in the plot, so for now you'll have to trust us or simulate the systems yourself based on Sect. 1.3.1.

1.3 Computer code for dynamical systems

As nonlinear dynamical systems are rarely treatable fully analytically, one always needs a computer to study them in detail or to analyze measured timeseries. Just a computer though is not enough, but one also needs code to run! This is the reason why we decided to write this book so that its pages are interlaced with real, runnable computer code. Having a "ready-to-go" piece of code also enables instant experimentation on the side of the reader, which we believe to be of utmost importance.

To write real code, we chose the Julia programming language because we believe it is highly suitable language for scientific code and even more so for the context of this book. Its simple syntax allows us to write code that corresponds line-by-line to the algorithms that we describe in text. Furthermore, Julia contains an entire software organization for dynamical systems, JuliaDynamics whose software we use throughout this book. The main software we'll be using is DynamicalSystems.jl, a software library of algorithms for dynamical systems and nonlinear dynamics, and InteractiveDynamics.jl, which provides interactive graphical applications suitable for the classroom. To demonstrate, let's introduce an example code snippet in Code. 1.1, similar to the code that we will be showing in the rest of the book. We think that the code is intuitive even for readers unfamiliar with Julia.

Code 1.1 Example code defining the Lorenz-63 system, Eq. (1.5) in Julia, and obtaining a trajectory for it using DynamicalSystems.jl.

```julia
using DynamicalSystems # load the library

function lorenz_rule(u, p, t)
    σ, ρ, β = p
    x, y, z = u
    dx = σ*(y - x)
    dy = x*(ρ - z) - y
    dz = x*y - β*z
    return SVector(dx, dy, dz) # Static Vector
end

p  = [10.0, 28.0, 8/3] # parameters: σ, ρ, β
u₀ = [0, 10.0, 0]      # initial state
# create an instance of a `DynamicalSystem`
lorenz = ContinuousDynamicalSystem(lorenz_rule, u₀, p)

T  = 100.0 # total time
Δt = 0.01  # sampling time
A  = trajectory(lorenz, T; Δt)
```

Code is presented in mono-spaced font with a light purple background, e.g., `example`. Code. 1.1 is an important example, because it shows how one defines a "dynamical system" in Julia using DynamicalSystems.jl[5]. Once such a dynamical system is defined in code, several things become possible through DynamicalSystems.jl. For example, in Code. 1.1 we use the function `trajectory` to evolve the initial condition of a `DynamicalSystem` in time, by solving the ODEs through DifferentialEquations.jl.

1.3.1 Associated repository: tutorials, exercise data, apps

Julia, DynamicalSystems.jl, and in fact anything else we will use in the code snippets are very well documented online. As this is a textbook about nonlinear dynamics, and not programming, we will not be teaching Julia here. Besides, even though we use Julia here, everything we will be presenting in the code snippets could in principle be written in other languages as well.

Notice however that Julia has an in-built help system and thus most snippets can be understood without consulting online documentations. Simply put, in the Julia console you can type question mark and then the name of the function that you want to know more about, e.g., `?trajectory`. Julia will then display the documentation of this function, which contains detailed information about exactly what the function does and how you can use it. Another reason not to explain code line-by-line here is because programming languages and packages get updated much, much faster than books do, which runs the risk of the explanations in a book becoming obsolete.

There is an online repository associated with this book, `https://github.com/JuliaDynamics/NonlinearDynamicsTextbook`. There we have collected a number of tutorials and workshops that teach core Julia as well as major packages. This repository also contains all code snippets and all code that produces the figures that make up this book, and there we can update code more regularly than in the book. The same repository contains the datasets that are used in the exercises of every chapter, as well as animations and graphical interactive applications that can also be used in lectures. It also includes several multiple choice questions that can be used during lectures (e.g., via an online polling service), but also in exams. Perhaps the most important thing in this repository are scripts that launch interactive, GUI-based apps that elucidate core concepts. These are present in the folder animations and are linked throughout the book. For convenience, a recorded video is also provided for each app.

1.3.2 A notorious trap

Almost every algorithm that we will be presenting in this book has a high quality performant implementation in DynamicalSystems.jl (or some other package). In

[5] In the following chapters we will not be defining any new dynamical systems in code, but rather use predefined ones from the `Systems` submodule.

addition, we strongly believe that code should always be shared and embraced in scientific work in nonlinear dynamics and new algorithms should be published with code implementation (hopefully directly in DynamicalSystems.jl). However, one must be aware that having code does not replace having knowledge. Having a pre-made implementation of an algorithm can lead into the trap of "just using it", without really knowing what it does or what it means. Thus, one must always be vigilant, and only use the code once there is understanding. We recommend students to attempt to write their own versions of the algorithms we describe here, and later compare with DynamicalSystems.jl implementations. Since DynamicalSystems.jl is open source, it allows one to look inside every nook and cranny of an algorithm implementation. Another benefit of using Julia for this book is how trivial it is to access to source code of any function. Simply preface any function call with `@edit`, e.g. `@edit trajectory(...)`, and this will bring you to the source code of that function (using `@which` instead will print the file path and line of code).

1.4 The Jacobian, linearized dynamics and stability

Previously we've mentioned *fixed points*, that satisfy $f(\mathbf{x}^*) = 0$ for continuous systems and $f(\mathbf{x}^*) = \mathbf{x}^*$ for discrete systems. In the rest of this section we care to answer one simple, but important question: when is a fixed point of a dynamical system "stable"? We will answer this question by coming up with an intuitive definition of stability, in terms of what happens infinitesimally close around a state space point $\mathbf{x}$ as time progresses.

Let's take a fixed point $\mathbf{x}^*$ and perturb it by an infinitesimal amount $\mathbf{y}$. We are interested in the dynamics of $\mathbf{x} = \mathbf{x}^* + \mathbf{y}$ and whether $\mathbf{x}$ will go further away or towards $\mathbf{x}^*$ as time progresses. For simplicity let's see what is happening in the continuous time case, and because $\mathbf{y}$ is "very small", we can linearize the dynamics with respect to $\mathbf{x}^*$, using a Taylor expansion

$$\dot{\mathbf{x}} = \dot{\mathbf{y}} = f(\mathbf{x}^* + \mathbf{y}) = f(\mathbf{x}^*) + J_f(\mathbf{x}^*) \cdot \mathbf{y} + \dots \tag{1.7}$$

where $J_f(\mathbf{x}^*)$ stands for the Jacobian matrix of f at $\mathbf{x}^*$ with elements

$$J_f(\mathbf{x}^*)_{ij} = \left.\frac{\partial f_i(\mathbf{x})}{\partial \mathbf{x}_j}\right|_{\mathbf{x}=\mathbf{x}^*} \quad \text{or} \quad J_f(\mathbf{x}^*) = \left[\begin{matrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D}{\partial x_1} & \cdots & \frac{\partial f_D}{\partial x_D} \end{matrix}\right]_{\mathbf{x}=\mathbf{x}^*} \tag{1.8}$$

and we have also stopped expanding terms beyond first order (the Jacobian represents the first order).

Since $f(\mathbf{x}^*) = 0$ by definition, the dynamics for small $\mathbf{y}$ and for short times is approximated by the behaviour of $\dot{\mathbf{y}} = J_f(\mathbf{x}^*) \cdot \mathbf{y}$, which is a *linear* dynamical system. This equation is also called the *linearized* dynamics, or the *tangent* dynamics, or the dynamics in the *tangent space*. Notice that because $\dot{\mathbf{y}} = J_f(\mathbf{x}^*) \cdot \mathbf{y}$

3

Defining and measuring chaos

Summary. Deterministic chaos is difficult to define in precise mathematical terms, but one does not need advanced mathematics to understand its basic ingredients. Here we define chaos through the intuitive concepts of sensitive dependence on initial conditions and stretching and folding. We discuss how volumes grow in the state space while defining Lyapunov exponents and focusing on practical ways to compute them and use them to distinguish chaotic from periodic trajectories.

3.1 Sensitive dependence on initial conditions

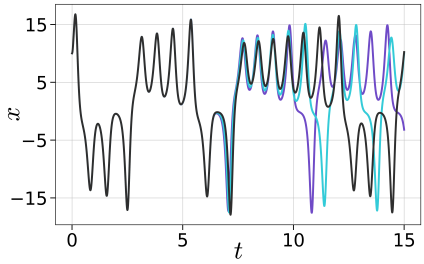
> For want of a nail the shoe was lost.
> For want of a shoe the horse was lost.
> For want of a horse the rider was lost.
> For want of a rider the battle was lost.
> For want of a battle the kingdom was lost.
> And all for the want of a horseshoe nail. [25]

This proverb has been in use over the centuries to illustrate how seemingly minuscule events can have unforeseen and grave consequences. In recent pop culture the term *butterfly effect* is used more often, inspired by the idea that the flap of the wings of a butterfly could lead to the creation of a tornado later on[1]. The butterfly effect is set in the spirit of exaggeration, but as we will see in this chapter, it has a solid scientific basis. It means that tiny differences in the initial condition of a system can lead to large differences as the system evolves. All systems that are chaotic display this effect (and don't worry, we will define chaos concretely in Sect. 3.2.2).

To demonstrate the effect, we consider the Lorenz-63 model from Eq. (1.5). In Fig. 3.1 we create three initial conditions with small differences between them and

[1] A flap of butterfly wings can never have such consequences. The statement is more poetic than scientific.

evolve all three of them forward in time, plotting them in purple, cyan and black color in Fig. 3.1. Initially there is no visual difference, and only the black curve (plotted last) is seen. But after some time $t \approx 8$ the three different trajectories become completely separated. A 3D animation of this is available online at animations/3/trajectory_divergence. It is also important to note that decreasing the initial distance between the trajectories would not eliminate the effect. It would take longer, but eventually the trajectories would still separate (see Sect. 3.1.2). This is the simplest demonstration of *sensitive dependence on the initial condition*, the mathematically precise term to describe the butterfly effect. It means that the *exact* evolution of a dynamical system depends sensitively on the *exact* initial condition. It should be noted however that if we are interested in average characteristics of a dynamical system instead of the exact evolution of a single trajectory starting at some particular initial value, then sensitive dependence isn't much of a problem due to ergodicity, see Chap. 8.



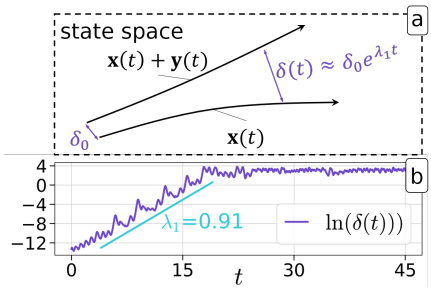

```
using DynamicalSystems, PyPlot
u0 = [10,10,10]
lo = Systems.lorenz(u0)
for i in 1:3
    u = u0 .+ i*1e-3
    tr = trajectory(lo, 15, u)
    plot(0:0.01:15, tr[:, 1])
end
```

Fig. 3.1: Sensitive dependence on the initial condition illustrated for the Lorenz-63 system simulated with three different, but very similar, initial conditions.

3.1.1 Largest Lyapunov exponent

Let's imagine a state $\mathbf{x}(t)$ of a chaotic dynamical system evolving in time. At time $t = 0$ we create another trajectory by perturbing the original one with an infinitesimal perturbation $\mathbf{y}$, with $\delta = ||\mathbf{y}||$, as shown in Fig. 3.2. If the dynamics is chaotic, the original and the perturbed trajectories separate more and more in time as they evolve. For small perturbations, this separation is happening approximately exponentially fast. After some time t the two trajectories will have a distance $\delta(t) \approx \delta_0 \exp(\lambda_1 t)$. This number λ_1 is called the *largest Lyapunov exponent* and quantifies the *exponential divergence* of nearby trajectories.

After some time of exponential divergence, the distance between the two trajectories will saturate and stop increasing exponentially, as is also visible in Fig. 3.2. Precisely why this happens will be explained in more detail in Sect. 3.2.4, but it stems from the fact that f is nonlinear and the asymptotic dynamics is bounded in state space. The important point here is that *exponential* divergence




```
using DynamicalSystems
using LinearAlgebra, PyPlot
lo = Systems.lorenz([20,20,20.0])
X₁ = trajectory(lo, 45)
u₂ = get_state(lo) .+ 1e-6
X₂ = trajectory(lo, 45, u₂)
δ  = norm.(X₂.data .- X₁.data)
plot(0:0.01:45, log.(δ))
λ = lyapunov(lo, 100000)
plot([4, 19], λ.*[0, 15] .- 13)
```

Fig. 3.2: (a) Sketch of exponential divergence of neighbouring trajectories characterized by the largest Lyapunov exponent λ_1. (b) The evolution of $\delta(t)$ for the Lorenz-63 system, produced via the code on the right.

of nearby trajectories occurs *only for small distances between trajectories*. This fact is the basis of the algorithm described in Fig. 3.3, which estimates λ_1, and is also implemented as `lyapunov` in DynamicalSystems.jl.

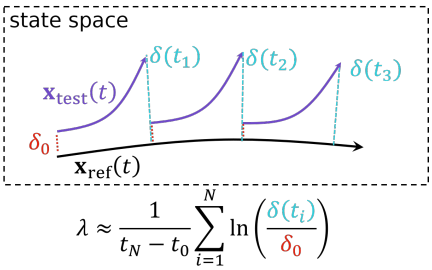


Fig. 3.3. Computing the largest Lyapunov exponent λ_1. The initial distance δ_0 of the two trajectories $\mathbf{x}_{\text{ref}}$ and $\mathbf{x}_{\text{test}}$ which are evolved in parallel has to be chosen very small. Their distance increases with time, and at times $t_i = i \cdot \Delta t$ the orbit $\mathbf{x}_{\text{test}}$ is re-scaled to have again distance δ_0 from $\mathbf{x}_{\text{ref}}$ with the transformation $\mathbf{x}_{\text{test}} \to \mathbf{x}_{\text{ref}} + (\mathbf{x}_{\text{test}} - \mathbf{x}_{\text{ref}})(\delta_0/\delta(t_i))$. λ_1 is approximated by time-averaging the logarithm of the ratio of distances.

3.1.2 Predictability horizon

The value of λ_1 has immediate practical impact. Imagine for a moment that there exists a "real" physical system that we want to predict its future. Let's also assume that we have a mathematical model that perfectly describes the real system. This is never true in reality, and the imperfect mathematical model introduces further uncertainty, but for now let's just pretend.

We now make a measurement $\tilde{\mathbf{x}}$ of the state of the real system $\mathbf{x}$. Our goal is to e.g., plug $\tilde{\mathbf{x}}$ into our mathematical model and evolve it in time on a computer, much faster than it would happen in reality. But there is a problem; our measurement of the state $\mathbf{x}$ is not perfect. For various reasons the measurement $\tilde{\mathbf{x}}$ is accompanied by an error $\mathbf{y}$, $\tilde{\mathbf{x}} = \mathbf{x} + \mathbf{y}$. Therefore, our measured state $\mathbf{y}$ has an initial distance from the real state $\delta = ||\mathbf{y}||$ and so the measured state and the real state will exponentially diverge from each other as they evolve in time as we have discussed so far. Let's say that if the orbits separate by some tolerance Δ, our prediction has too much error and becomes useless. This will happen after time $t_\Delta = \ln(\Delta/\delta)/\lambda_1$. This explains why chaotic systems are "unpredictable", even though deterministic, since any attempt at prediction will eventually become unusable.

The characteristic time scale $1/\lambda_1$ is often called the *Lyapunov time* and defines the *predictability horizon* t_Δ. One more difficulty for predicting chaotic systems is that because of the definition of t_Δ, in order to increase this horizon linearly (e.g. double it), you need to increase the measurement accuracy exponentially (e.g. square it). Although the reasoning regarding the predictability horizon is so far valid, we should be careful with how generally we apply it, see Chap. 12 for further discussion.

3.2 Fate of state space volumes

3.2.1 Evolution of an infinitesimal uncertainty volume

So far we have represented "uncertainty" of a measurement as a single perturbation of an initial condition. While this is useful for conceptualizing divergence of trajectories, a more realistic scenario is to represent uncertainty as an infinitesimal volume of perturbed initial conditions around the state of interest $\mathbf{x}_0$. Then, we are interested to see how this infinitesimal volume changes size and shape as we evolve the system.

To simplify, we will consider the initial volume as a D-dimensional hypersphere around an initial condition $\mathbf{x}_0$. Once we evolve the infinitesimal hypersphere, we will see it being deformed into an ellipsoid as shown in Fig. 3.4. Movies of this process for continuous and discrete systems in 3D can be seen online in animations/3/volume_growth. The ellipsoid will also rotate over time, but for this discussion we only care about the change in its size. We can quantify the change in size by looking at the growth, or shrinking, of the axes of the ellipsoid. What we find, for almost all initial conditions $\mathbf{x}_0$, is that the i-th axis of the ellipsoid will approximately grow, or shrink, exponentially fast in time, as $\delta e^{\lambda_i t}$ with δ being the initial radius of the infinitesimal hypersphere. Keep in mind that this will only occur for small perturbation volumes. Evolving a hypersphere with a finite-sized, but still small, radius δ means that the exponential increase along the directions with $\lambda_i > 0$ will sooner or later saturate, exactly as it has been discussed in the Sect. 3.1.1.

6.3.3 Unified optimal embedding

So far we've only talked about how to find the optimal embedding dimension and delay time for a single input timeseries and with constant τ. In addition, these methods (mutual information and AFNN) were separated from each other. As far as the embedding theorems are concerned an optimal combination (d, τ) simply provides an appropriately good coordinate system in which a diffeomorphism F maps the original set A to an embedded set R. A necessary condition is that the coordinates of the reconstruction space are sufficiently independent.

Splitting the task of finding independent coordinates into two tasks (delay time and embedding dimension) is mostly artificial and we therefore introduce in the following concepts of *unified* optimal delay coordinates based on the generalized embedding of the previous section. They try to create the most optimal (but also most minimal) generalized embedding that most accurately represents the original invariant set A from which the measured timeseries w_j originate from. As a result they typically yield better embeddings than the conventional approach discussed in Sect. 6.2. These methods work by coming up with a statistic that quantifies how "good" the current embedding is. Then, they proceed through an iterative process. Starting with one of the possible timeseries, additional entries are added to the generalized embedding one by one. One checks all offered timeseries w_j of the measurement X, each having a delay time from a range of possible delay times. The combination of (w_i, τ_i) that, when added to the (ongoing) embedding, leads to the most reduction (or increase, depending on the definition) of the statistic is picked. This iterative process terminates when the statistic cannot be minimized (or maximized) further by adding more entries to the generalized embedding.

Okay, that all sounds great, but what's a statistic that quantifies how "good" the current embedding is? Unfortunately, describing this here would bring us beyond scope, so we point to the Further reading section. However, to give an example, one of the earliest such statistics was a quantification of how much functionally independent are the coordinates of the (ongoing) generalized embedding. Once adding an additional delayed coordinate with any delay time cannot possibly increase this independence statistic, an optimal embedding has been reached. A numerical method that performs unified delay embedding is provided in DynamicalSystems.jl as `pecuzal_embedding`.

6.4 Some nonlinear timeseries analysis methods

Throughout this book we perform nonlinear timeseries analysis, e.g. in Sect. **??**, but in the current section we list some methods that are tightly linked with delay coordinate embeddings.

6.4.1 Nearest neighbor predictions (forecasting)

What is the simplest possible way to make a prediction (or a forecast) of a deterministic chaotic timeseries w without explicitly modelling the dynamics?

(Hint: it is *not* machine learning.) The simplest way will become apparent with the following thought process: Our reconstruction R, via delay embedding w, represents a chaotic set, provided we picked proper (d, τ). Assuming we satisfy the standard criteria for delay embedding (stationarity and convergence), then R is invariant with respect to time: moving all initial states forward makes them stay in R. In addition, if we have some nearby points, when we evolve them forward in time (for an appropriately small amount of time), these points stay together in the reconstruction.

And this is where the trick lies: the last point of the timeseries w_N has been embedded in R as a point $\mathbf{q}_N$. This is our "query point". We then find the nearest neighbors of $\mathbf{q}_N$ in R and propagate them one step forward in time. All points but the last one have an image one step forward in time already existing in R so this step is trivial. Then, we make a simple claim: the average location $\tilde{\mathbf{q}}$ of the forward-images of the neighbors of $\mathbf{q}$ is the future of $\mathbf{q}$, as illustrated in Fig. 6.6. The last entry of $\tilde{\mathbf{q}}$ (or the first, depending if you did a forward or backward embedding) is the new predicted point of the timeseries w_{N+1}. We then embed the extended timeseries w_{N+1} again in R and continue using this algorithm iteratively.

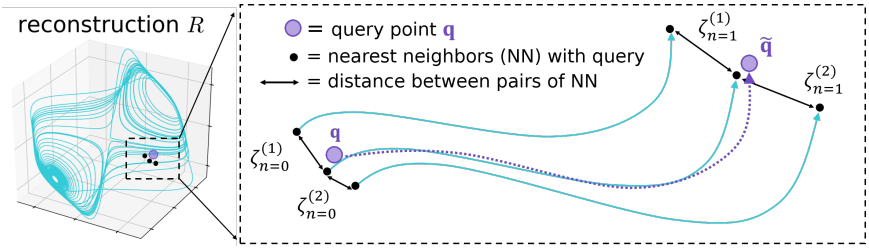


Fig. 6.6: Demonstration of basic nonlinear timeseries analysis: nearest neighbor prediction and Lyapunov exponent from a timeseries. Notice that the flow lines exist only to guide the eye, in the application only the points are known.

6.4.2 Largest Lyapunov exponent from a sampled trajectory

Following from the previous section, it is also easy to get a value for the largest Lyapunov exponent given a sampled trajectory R (which is typically the result of delay embedding of a timeseries w). We start by finding all pairs of nearest neighbors in R. Each pair i has a distance $\zeta^{(i)}_{n=0}$. We then go to the future image of all points, simply by incrementing the time index by 1, and check again their distance at the next time point: $\zeta^{(i)}_{n=1}$. Continue the process while incrementing the time index for as much time as is reasonable, say k time steps. At each step we compute the average of the logarithms of these distances, $\xi_n = \frac{1}{M}\sum_{i=1}^{M} \log(\zeta^{(i)}_n)$ with $M = N - k$.

As we have learned in Chap. 3, this ξ_n can be expressed versus n as $\xi_n \approx n \cdot \lambda \cdot \delta t + \xi_0$ with δt the sampling time for continuous systems and λ the maximum Lyapunov exponent exactly as in Fig. 3.2. This happens until some maximum threshold time k where ξ_n will typically saturate due to the folding of the nonlinear dynamics exactly like we discussed in Chap. 3. One then simply finds the slope of the curve $\xi_n/\delta t$ versus n, which approximates λ. The function `numericallyapunov` from DynamicalSystems.jl does this for a dataset.

6.4.3 Permutation entropy

Permutation entropy connects the concepts of Chap. 5 with delay coordinates. Given a timeseries it outputs a quantifier of complexity that is similar to the maximum Lyapunov exponent. It works by looking at the probabilities of relative amplitude relations of d successive points in a timeseries. For $d = 2$ the first point can be either larger or smaller than the previous[4]. This gives two possibilities, and one can associate a probability to each by counting the pairs that fall into this possibility and then dividing by the total. For $d = 3$ there are 6 permutations (see Fig. 6.7), while for general d the number of possible permutations equals $d!$. Each one of the possible permutations then gets associated a probability p_i based on how many d-tuples of points fall into this permutation. The permutation entropy is simply the Shannon entropy, Eq. (5.2), of p_i. Sometimes d is called the order of the permutation entropy.

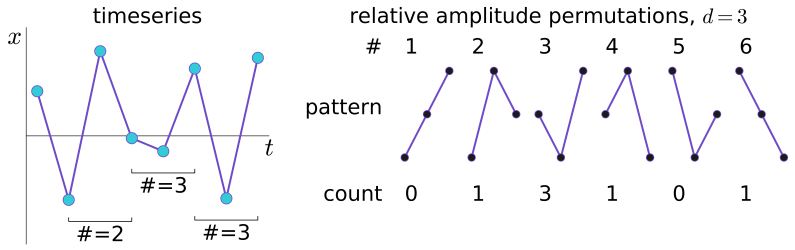


Fig. 6.7: Permutations of relative amplitude relations for $d = 3$ based on a timeseries. Segments of length d are then mapped to the corresponding permutation of relative amplitudes (also called ordinal patterns). Use the function `permentropy` from DynamicalSystems.jl to obtain the value of the permutation entropy.

And how does this connect with delay embedding? Well, what we do here is look at the relative amplitude relations within the points of a d-dimensional embedding with delay $\tau = 1$. Instead of doing a histogram of the delay embedded set, we partition it differently according to the possibilities of the relative

[4] In the case of equality, best practice is to choose randomly between $<$ or $>$.

amplitudes, resulting in $d!$ possible "boxes" to distribute the data in. In fact, one can extent permutation entropy to larger delays $\tau > 1$ and the approach remains valid. While permutation entropy does not have a straightforward dynamical interpretation as λ_1 has, it comes at the benefit of being quite robust versus noise in the data and simple to compute. Permutation entropy can also be used to detect non-stationary in the data, see the exercises.

Further reading

Excellent textbooks for delay coordinate embeddings and all topics related to nonlinear timeseries analysis are *Nonlinear Time Series Analysis* by H. Kantz and T. Schreiber [68] and *Analysis of Observed Chaotic Data* by Abarbanel [94], both having a practical focus and discussing real world data implications. See also Ref. [95] for more recent review of nonlinear timeseries analysis.

Delay coordinates embedding of timeseries was originally introduced in [96] by Packard, Crutchfield, Farmer and Shaw, stimulated by Ruelle (see [97]). It then obtained an initial rigorous mathematical footing from the embedding theorem of Takens [98], based on Whitney [99]. That is why in the literature the delay embedding theorem is often called *Takens theorem*. However, very important generalizations of Takens' theorem came from Sauer, Yorke and Casdagli in [100, 101] and brought it in the form presented in this book, which describes the embedding condition based on the fractal dimension Δ. The impact of noise on the validity of these theorems is discussed in [102]. Adaption to event-like timeseries was done by Sauer in [103]. That is why we decided in this book to drop the term "Takens theorem" and use the term "delay embedding theorem". Ref. [104] by Letellier et al. discusses the influence of symmetries on the reconstruction problem and see also [105, 106, 107] for the impact of the observation function h.

For signals $w(t)$ from continuous systems, as an alternative to delay coordinates, *derivative coordinates* $\mathbf{v}(t) = (w(t), \dot{w}(t), \ddot{w}(t), ...)$ can be used for state space reconstruction. This approach is justified by embedding theorems similar to those for delay coordinates [98, 100]. Since estimating higher temporal derivatives from noisy (measurement) data can be very challenging, this method of state space reconstruction is suitable for low dimensional reconstructions only. An alternative method to overcome or bypass the issue of selecting a proper time delay was introduced by D. Broomhead and G. King [108], combining embedding in a very high dimensional space in combination with linear dimensional reduction (principal component analysis).

The references regarding the conventional approach to optimal embedding are too numerous to list all of them here. Important to point out is the work of Kennel, Brown and Abarbanel [109], which was the first to come up with a way to estimate an optimal embedding dimension based on the concept of false nearest neighbors. Then Cao [110] provided an improved method to do the same. More historical references and reviews of the conventional approach to delay embeddings and can be found in [102, 111]. Details on embeddings of event-like timeseries can be found in [103]. The impact of linear filtering a signal prior to

delay embedding has be investigated by Broomhead et al. [112] and Theiler and Eubank [113].

Unified approaches to delay coordinate reconstructions have been developed only recently, starting with the work of Pecora et al. [111] that we described loosely in Sect. 6.3.3 (which, by the way, also contains excellent historical overview of delay embeddings). Further improvements have been proposed since then, see e.g. Krämer et. al. [114] and references therein. All unified approach based methods (to date) are implemented in DynamicalSystems.jl.

The framework of nearest neighbor prediction described in Sect. 6.4.1 is implemented in the software TimeseriesPrediction.jl. Permutation entropy, originally introduced by Bandt and Pompe [115], has been extended in various ways [116, 117, 118].

Selected exercises

6.1 Implement the method of false nearest neighbors described in Sect. 6.2.2. To find nearest neighbors in higher dimensional spaces, you can still write a simple brute-force-search that compares all possible distances. However it is typically more efficient to rely on an optimized structure like a KDTree. Simulate a timeseries of the towel map (e.g. pick the first coordinate)

$$\begin{aligned} x_{n+1} &= 3.8x_n(1-x_n) - 0.05(y_n + 0.35)(1-2z_n) \\ y_{n+1} &= 0.1\left((y_n + 0.35)\,(1+2z_n) - 1\right)(1-1.9x_n) \\ z_{n+1} &= 3.78z_n(1-z_n) + by_n \end{aligned} \tag{6.5}$$

and from it try to find the optimal embedding dimension. Compare your code with the function `delay_afnn` from DynamicalSystems.jl to ensure its accuracy. *Hint: since this is a discrete system,* $\tau = 1$ *is the delay time choice.*

6.2 Simulate a timeseries of the Rössler system, Eq. (4.1), for some sampling time δt. Using the auto-correlation function or self-mutual information (we strongly recommend to use the mutual information version), estimate the optimal embedding time for your timeseries. How do you think this should depend on δt? Re-do your computation for various δt and provide a plot of $\tau(\delta t)$.

6.3 Load timeseries from exercise datasets 1, 3, 5, 12 (and for multidimensional data use the first column). For each try to achieve an optimal delay coordinates embedding using the conventional approach.

6.4 Repeat the previous exercise for datasets 6, 9, 10, 13.

6.5 Implement the method of Sect. 6.4.2 for computing the maximum Lyapunov exponent from a given dataset. Apply the result to dataset 2, which has a sampling time of 0.05. Compare your code with the function `numerical_lyapunov` from DynamicalSystems.jl to ensure its accuracy.

6.6 Repeat the previous exercise for datasets 3, 6, 10. The sampling times are 0.01, 1, 0.01, respectively.

6.7 For the datasets you have performed the two previous exercises, apply the 0-1 test for chaos (to the first timeseries). Do your results coincide? *Hint: assume that if you find $\lambda > 0$, then the original timeseries is chaotic.*

6.8 Simulate a 10,000-length timeseries of the Hénon map, Eq. (3.3), and record the first variable as a timeseries. Attempt a prediction of 100 points using the algorithm of the simple nearest-neighbour-based forecasting of Sect. 6.4.1.

6.9 Load dataset 7, which is a non-stationary timeseries with observational noise. Use the permutation entropy (write your own function to compute it!) and report its value for various orders d. Compare the results with the function `permentropy` from DynamicalSystems.jl.

6.10 Analyze the timeseries of the previous exercise by splitting it into windows of length W each with overlap $w < W$. For each window calculate permutation entropies of different orders, and plot all these results versus time. Can you identify time-windows of characteristically different behavior from these plots? How do your results depend on W, w and the order of the permutation entropy? In the literature, this process is called *identifying regime shifts*, and the permutation entropy is one of several tools employed to find them.