DYNAMICAL PROPERTIES OF A GENERALIZED COLLISION RULE FOR MULTI-PARTICLE SYSTEMS

by

Joseph Dinius

A Dissertation Submitted to the Faculty of the

Graduate Interdisciplinary Program in Applied Mathematics

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 1 4

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Joseph Dinius entitled
Dynamical Properties of a Generalized Collision Rule for Multi-Particle Systems
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

______________________________ Date:
Joceline Lega

______________________________ Date:
Hermann Flaschka

______________________________ Date:
Ricardo Sanfelice

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

______________________________ Date: April 2, 2014
Joceline Lega

Statement by Author

Dedication

For my wife, Jessica. I would not have been able to accomplish this without you. I consider myself blessed each and every day I am with you. Thanks for all of your love and support.

Acknowledgments

First and foremost, thanks go to my family for always encouraging me in all of my personal, academic and professional pursuits. My parents' work ethic has been a constant motivating force to achieve. I am very thankful to have been pushed and nurtured when it was necessary to do so. I look back on the time when my Aunt Kathleen achieved her doctorate, and I remember that being the moment I decided that I wanted to achieve something so meaningful for myself.

So many inspiring forces have helped me on my academic journey. As an undergraduate at Northern Arizona University, I remember wonderful discussions with Drs. David Cornelison, Randy Dillingham, Janet McShane and James Swift regarding the nature of research and physical inquiry. Dr. Gus Hart was a particularly strong influence on me. His "brute-force" approach to the numerical and mathematical methods of physics provided me with the tools and the strength to tackle complex problems methodically. Since coming to the University of Arizona, there have been a number of additional professors and graduate students with whom I have had many a fruitful discussion. Particular thanks go to my committee members: Drs. Joceline Lega, Hermann Flaschka and Ricardo Sanfelice. Joceline has helped me to discover the path to meaningful scientific research, though I must admit it was slow-going at times. Drs. Flaschka and Sanfelice have helped me to realize what are, and are not, meaningful details regarding the work of this dissertation. Special thanks are owed to Dr. Kevin Lin, who has provided needed guidance and helpful advice throughout the research process. I would like to also extend special thanks to Dr. Christoph Dellago. Without a response from him in 2011 regarding a bug I was having with my codes, I might not have continued the study of this very interesting problem. I am also grateful for his agreeing to be the external reviewer of this body of work. I consider myself deeply indebted to each of these individuals.

I would like to extend thanks to my colleagues and management at Raytheon Systems

Company for providing me with the professional and financial support for completing this work. Particularly, I would like to thank Ron Reid and Tom McCarthy for believing in me and recommending me for the Advanced Scholars Program. I would also like to thank Matt Rothman, Sam Pisani, and Andy Dolphin for allowing me the flexibility with regards to my working schedule to pursue this achievement.

Table of Contents

List of Figures **10**

List of Tables **11**

Abstract **12**

Motivation and Introduction **13**

Chapter 1. Mathematical Preliminaries **17**
- 1.1. Basics from Dynamical Systems 17
  - 1.1.1. What is a Dynamical System? 17
  - 1.1.2. Discrete-time Dynamics and Maps 19
  - 1.1.3. Continuous-time Dynamics and Flows as Maps 19
  - 1.1.4. Continuous Flows Interrupted by Discrete Jumps 20
  - 1.1.5. A Motivating Example: The Bouncing Ball-Platform System 21
- 1.2. Tangent Space Dynamics and Lyapunov Exponents 23
  - 1.2.1. Dynamics of Perturbations in Discrete-time 23
  - 1.2.2. Singular Value Decomposition 25
  - 1.2.3. The Multiplicative Ergodic Theorem of Oseledec 28
- 1.3. Further Reading 32

Chapter 2. Numerical Considerations and Algorithms **34**
- 2.1. Notional “Numerical” Scheme for Computing Lyapunov Exponents 34
- 2.2. *QR* Factorization and Benettin’s Algorithm 36
  - 2.2.1. *QR* Factorization and Gram-Schmidt Orthonormalization 37
  - 2.2.2. Benettin’s Algorithm 40
  - 2.2.3. Worked Examples 42
- 2.3. Lyapunov Vectors 47
  - 2.3.1. Orthogonal Lyapunov Vectors 49
  - 2.3.2. Covariant Lyapunov Vectors 52
- 2.4. Further Reading 57

Chapter 3. A Generalized Impulsive Collision Rule for Interacting Particles . . . **60**
- 3.1. Phase Space Dynamics 61
  - 3.1.1. Free Flight 62
  - 3.1.2. Collision 62
- 3.2. Inverse (Backward-Time) Collision Rule 67
- 3.3. Linearized Dynamics of Perturbations 69

Table of Contents—*Continued*

3.3.1. Free Flight 70
3.3.2. Collision 70
3.3.3. Constructing the Transformation **S** 73
3.4. Concluding Remarks 78

Chapter 4. Dynamical Properties of Particle Systems Undergoing Elastic Collisions **80**
4.1. Lyapunov Exponents for Hard Disks 82
4.1.1. Zero Exponents 86
4.2. Measured Lyapunov Vectors for Hard Disk Systems 86
4.2.1. Localization 87
4.3. Coherent Vector Structure Near the Center Index 88
4.3.1. Mode Classification 91
4.4. Covariant Lyapunov Vectors 97
4.4.1. Localization and Mode Classification 97
4.4.2. Hyperbolicity 101
4.5. Further Reading 102

Chapter 5. Dynamical Properties of Generalized Collisions **105**
5.1. Lyapunov Exponents 106
5.2. Properties of Orthogonal Vectors (Modes) 109
5.2.1. Localization 110
5.2.2. Observing Coherent Structures 111
5.3. Hyperbolicity 118

Conclusions and Future Work **120**

Appendices **123**

Appendix A. Justification for Important Results **124**
A.1. The Multiplicative Ergodic Theorem 124
A.2. Support for Claims in Chapter 3 133
A.2.1. Proof that $\beta \in [0, 1]$ 133
A.2.2. Proof that $\inf \alpha = \beta$ 135

Appendix B. Computational Code **137**
B.1. Code Availability 137
B.2. Source Code Listings for Chapter 1 137
B.3. Source Code Listings for Chapter 2 139
B.3.1. Lorenz Example 139
B.3.2. Bouncing Ball-Platform Example 141

TABLE OF CONTENTS—*Continued*

B.3.3. Hénon Example 144

B.4. Source Code Listings for Chapters 4 & 5 149

B.4.1. Building the Code 150

B.4.2. Running the Code 151

B.5. Global Variable Description 151

B.5.1. utilities.h 151

B.6. Detailed Module Description 158

B.6.1. main.cpp 158

B.6.2. utilities.cpp 182

B.7. File I/O Description 277

B.7.1. Sample Input Files 278

B.7.2. Reading the Data in Matlab® 284

REFERENCES **289**

List of Figures

Figure 1.1. Bouncing Ball-Platform System 22
Figure 1.2. The ball-platform system is chaotic 24
Figure 1.3. Evolution of hyperellipsoid in tangent space 26

Figure 2.1. Lyapunov exponents vs. ρ for the Lorenz system 45
Figure 2.2. Lorenz Systems: Behavior of trajectories vs. ρ 46
Figure 2.3. Bifurcation diagram and Lyapunov exponents for the bouncing ball . . 48
Figure 2.4. Hénon attractor 57
Figure 2.5. Covariant Lyapunov Vectors and Invariant Manifolds for the Hénon map. 58

Figure 3.1. Collision in forward- and backward-time. 67
Figure 3.2. Collision offset in perturbed trajectory. 72

Figure 4.1. Hard Disks: The conjugate-pairing rule 82
Figure 4.2. Hard Disks: Maximum and smallest positive exponents vs. $1/\sqrt{N}$ . . . 83
Figure 4.3. Hard Disks: Entropy production per particle vs. $1/\sqrt{N}$. 85
Figure 4.4. Hard Disks: Localization Width $\mathcal{W}^{(i)}$ 89
Figure 4.5. Hard Disks: Full Lyapunov spectrum for $N = 780$ 90
Figure 4.6. Hard Disks: Average cosine of the angle between $\delta\mathbf{Q}$ and $\delta\mathbf{P}$. 92
Figure 4.7. Hard Disks: Lyapunov Modes as Vector Fields. 93
Figure 4.8. Hard Disks: Lyapunov Exponents from Benettin and Ginelli Methods. . 98
Figure 4.9. Hard Disks: Comparison of Measures from Benettin and Ginelli Methods.100
Figure 4.10. Hard Disks: Angles Separating Stable and Unstable Subspaces. 103

Figure 5.1. Average of collision parameter α vs. β. 107
Figure 5.2. Lyapunov exponents and KS entropy vs. β. 108
Figure 5.3. Lyapunov spectrum vs β. 110
Figure 5.4. Per particle phase space contraction rate vs. β. 111
Figure 5.5. Localization measure $\mathcal{W}^{(i)}$ vs. β: Full spectrum 112
Figure 5.6. Localization measure $\mathcal{W}^{(i)}$ vs. β. 113
Figure 5.7. Decay in orthogonal Lyapunov vector coherent structure vs. β. 114
Figure 5.8. Fluctuation of local (in time) Lyapunov exponents for different β. . . . 116
Figure 5.9. Measure of dynamic isolation of Oseledec Subspaces p_i^{split}. 117
Figure 5.10. Minimum separation in angle of the unstable and stable subspaces vs. β. 119

List of Tables

Table 3.1. Description of offset quantities 71

Table 4.1. Basis elements of the scalar modulation function A 94
Table 4.2. Basis elements of $T(\mathbf{n})$ subspace. 95
Table 4.3. Basis elements of $L(\mathbf{n})$ and $P(\mathbf{n})$ subspaces. 96

Abstract

The theoretical basis for the Lyapunov exponents of continuous- and discrete-time dynamical systems is developed, with the inclusion of the statement and proof of the Multiplicative Ergodic Theorem of Oseledec. The numerical challenges and algorithms to approximate Lyapunov exponents and vectors are described, with multiple illustrative examples. A novel generalized impulsive collision rule is derived for particle systems interacting pairwise. This collision rule is constructed to address the question of whether or not the quantitative measures of chaos (e.g. Lyapunov exponents and Kolmogorov-Sinai entropy) can be reduced in these systems. Major results from previous studies of hard-disk systems, which interact via elastic collisions, are summarized and used as a framework for the study of the generalized collision rule. Numerical comparisons between the elastic and new generalized rules reveal many qualitatively different features between the two rules. Chaos reduction in the new rule through appropriate parameter choice is demonstrated, but not without affecting the structural properties of the Lyapunov spectra (e.g. symmetry and conjugate-pairing) and of the tangent space decomposition (e.g. hyperbolicity and domination of the Oseledec splitting). A novel measure of the degree of domination of the Oseledec splitting is developed for assessing the impact of fluctuations in the local Lyapunov exponents on the observation of coherent structures in perturbation vectors corresponding to slowly growing (or contracting) modes. The qualitatively different features observed between the dynamics of generalized and elastic collisions are discussed in the context of numerical simulations. Source code and complete descriptions for the simulation models used are provided.

Motivation and Introduction

This work is concerned with systems of particles that interact in pairs via simple impulsive collisions. By impulsive, it is meant that the collision interaction between a pair of particles is localized in both space and time. The exact structure of this rule will discussed in Chapter 3. The most studied collision rule is the *elastic* rule, which conserves kinetic energy and both linear and, in the case of systems with rotation, angular momenta. Particle systems in two-dimensions interacting through elastic collisions are called *hard-disk systems*[1]. Impulsive collision rules have been used since the time of Boltzmann [7] as a simplified model for understanding complex physical systems with many degrees-of-freedom (e.g. dilute gases) and are used as the basis for successful perturbation theories of dense gases and fluids [4, 42]. Following the work of Alder and Wainwright [1, 2], studies pertaining to macroscopic quantities such as transport coefficients [23, 25, 76] and autocorrelation functions [19, 24, 41] of these particle systems have been carried out for many years. Over the last two decades, an approach concerned with quantifying the degree of microscopic chaos present in these systems has emerged. Since the pioneering work of Dellago *et. al* [16], many very nice results have been presented demonstrating the conjugate-pairing [27] of the *Lyapunov exponents*, which quantify the rate of exponential separation of nearby trajectories, the localization [91] and presence of coherent structures [21, 31] in orthonormal vectors obtained from the computation of the Lyapunov exponents and, most recently, the hyperbolic properties of the tangent space decomposition along typical trajectories [8, 68, 69].

Recently, a study by Lega demonstrated the emergence of coherent structures in systems of particles interacting with a generalized collision rule [59]. The physical motivation for the construction of this generalized rule is the observation that bacterial systems that appear to have no means of long-range communication organize themselves into collective modes

[1]Similarly, in three-dimensions, they are called *hard-sphere systems*

of motion [63]. Therefore, a reasonable model for the interactions between bacteria should be localized in both space and time. This work seeks to provide quantitative and qualitative analyses of a class of generalized collision rules, of which the rule presented in Lega [59] is an element, using the analytical methods developed for elastic collision rules.

This dissertation is organized into five chapters, with two appendices at the end. Chapters 1, 2 and 4 provide an overview of existing theoretical developments for investigating dynamical systems with many degrees-of-freedom. At the end of these chapters, a list of relevant readings is provided should readers feel compelled to pursue research along a path tangential to the author's. Chapters 3 and 5 contain the novel contributions of this thesis. The appendices provide support for claims made throughout the body of this work and the author's own code implementating all algorithms in either Matlab® or C++. Appendix A.1 presents the statement and proof of the Multiplicative Ergodic Theorem of Oseledec, which provides the necessary conditions for the existence of Lyapunov exponents of differentiable maps. This is a nice addition, as the proof of this theorem helps to build intuition regarding the development of numerical algorithms for practical computations. Appendix B presents Matlab® code for all examples in Chapters 1 and 2, of which there are several, along with the C++ code used for generating data presented in Chapters 4 and 5. The C++ code is presented in a block-by-block fashion with thorough explanations of code function.

Chapter 1 is concerned with the theoretical basis necessary for the remainder of the dissertation. The fundamentals of dynamical systems, including the definition of what a dynamical system is, are presented, along with properties of flows (in continous-time) and maps (in discrete-time). The dynamics of perturbations are discussed, in detail, as a means of understanding the behavior of closely-separated trajectories in nonlinear systems, along with the concept of Lyapunov exponents. The connection between these exponents and the singular values of a matrix is made clear. The chapter concludes with the motivation behind and the statement of the Multiplicative Ergodic Theorem of Oseledec.

The numerical algorithms used to compute the Lyapunov exponents are discussed in Chapter 2. The most widely used algorithm in the literature is due to Benettin *et. al*

[5, 82, 98]. The details of the implementation, and also the limitations of the algorithm when it comes to constructing the so-called *Oseledec splitting*, which refers to the structure of the tangent space decomposition along typical trajectories, are thoroughly described. To address these limitations, the recently developed algorithm of Ginelli *et. al* [36, 37] is shown to recover elements of the Oseledec splitting using both forward- and backward-time propagation of typical trajectories. The methods discussed in this chapter are then applied to well-known examples to illustrate their application.

With all of the necessary dynamical systems theory in place, Chapter 3 defines and derives the dynamics (both forward- and backward-time) of generalized collisions for interacting particle pairs. The elastic collision rule is shown to be a special case of this broader class of generalized collisions. As Benettin's algorithm requires modification for systems of interacting particles (see [16] for more details), the changes required for the propagation of the linearized dynamics of perturbations is detailed.

Chapter 4 summarizes some of the major results for hard-disk systems. The dependence of the magnitude of the Lyapunov exponents on packing density and particle number is shown, along with the symmetric structure of the full set of Lyapunov exponents, called the *Lyapunov spectrum*. The structure of the orthonormal vectors computed in Benettin's algorithm are demonstrated to fall into two classes:

1. Localized, incoherent vectors corresponding to the fastest growing (decaying) directions

2. Smooth, delocalized vectors corresponding to the most slowly growing (decaying) directions.

A quantitative means of assessing the localization properties of these vectors due to Taniguchi and Morriss [91] is shown to quantify the average number of particles contributing to each orthonormal vector. Some theoretical consideration is made to the delocalized vectors, which are consequences of the continuous symmetries of the system thus indicating a natural connection to kinetic theory [61, 62]. The chapter concludes with the computation of the angular separation of the unstable and stable subspaces along typical trajectories, which are obtained as direct sums of the covariant vectors output from Ginelli's algorithm.

Finally, Chapter 5 applies the methods from the previous chapters to make quantitative assessments of generalized collision rules. The consequences of time-irreversibility of generalized collision rules (except for special cases) are demonstrated in the Lyapunov spectra computed and measures of vector localization. Fluctuations in local growth (and contraction) rates are correlated to the loss of the coherent, smooth structure present in vectors from elastic collision simulations. A novel splitting measure, similar to one presented in Yang and Radons [100], is developed for probabilistically assessing the presence of delocalized structures in the orthonormal vectors. The hyperbolicity of the tangent space decomposition is shown to be related to the fluctuations observed in the local growth (and contraction) terms.

Chapter 1
Mathematical Preliminaries

The goal of this chapter is to provide the necessary mathematical background for understanding the remainder of this work. The formal definition of an abstract dynamical system is provided, along with concrete examples illustrating key concepts. The geometric notions of irregularity and chaos in dynamical systems are presented via a simple, low-dimensional example: the bouncing ball-platform system. Upon observation of this phenomena, the notion of exponential separation of trajectories is presented as a means of quantifying chaos, culminating in the statement of Oseledec's Multiplicative Ergodic Theorem and its consequences.

This chapter is intended to be self-contained. However a minimal understanding of probability theory would be helpful (see [55]) . Additionally, some knowledge of differentiable manifolds (see [57, 58, 94]) and a thorough understanding of linear algebra (see [39, 93]) would serve the reader well.

1.1 Basics from Dynamical Systems

1.1.1 What is a Dynamical System?

According to [46]: "A *dynamical system* is a way of describing the passage in time of all points of a given space [X]." (The change from the original $\mathcal{S}$ in the text to X is to ensure continuity of subsequent definitions and arguments). This definition is by no means mathematically rigorous, however it motivates the physical setting. To mathematically describe this "passage of time" requires the definition of a space, X, known as the *phase space*, a transformation, $\mathbf{M}$, and a set of times Π over which $\mathbf{M}$ is defined. Therefore, mathematically, a dynamical system can be described by the triple $(X, \Pi, \mathbf{M})$. Additionally, we are concerned with systems that preserve a probability measure μ over measurable subsets of

X. Such measure-preserving systems are of fundamental importance in statistical physics [18, 35, 56, 95], ergodic theory [12, 53, 96], and Hamiltonian mechanics [71, 74]. First, some definitions:

Definition 1.1.1. (Invariant measure) [71] Let $(X, \mathcal{B})$ be a measurable space and $f : X \to X$ be a measurable function. A measure μ is said to be **invariant under M** if, for every measurable set $B \in \mathcal{B}$,

$$\mu(\mathbf{M}^{-1}(B)) = \mu(B),$$

where $\mathbf{M}^{-1}(B)$ is the preimage, or inverse image, of the set B.

Definition 1.1.2. (Measure-preserving transformation) [12] Let $(X, \mathcal{B}, \mu)$ be a probability space. A transformation $\mathbf{M} : X \to X$ is a **measure-preserving transformation** if, for every $B \in \mathcal{B}$,

$$\mu(\mathbf{M}^{-1}(B)) = \mu(B).$$

The statements "μ is an invariant measure for $\mathbf{M}$" and "$\mathbf{M}$ preserves the measure μ" are equivalent. Now, we can fully classify a measure-preserving dynamical system as $(X, \mathcal{B}, \mu, \Pi, \mathbf{M})$, where for our purposes, it is understood that:

- X is a C^∞, or smooth, Riemannian manifold with a global differentiable structure. X is known as the *phase space* for the system.
- $\mathcal{B}$ is a σ-algebra: a collection of subsets of X with positive measure with respect to the measure μ. In less abstract terms, $B \in \mathcal{B}$ is an observable of the system.
- μ is an invariant probability measure on X.
- Π is the set of all times over which the system evolves.
- $\mathbf{M} : X \to X$ is the *evolution rule* over time ($t \in \Pi$). $\mathbf{M}$ is measure-preserving with respect to the measure μ.

The most common sets for Π are $\mathbb{Z}$, the set of integers, and $\mathbb{R}$, the real numbers. When $\Pi = \mathbb{Z}$, the dynamical system is *discrete-time*, and when $\Pi = \mathbb{R}$, the dynamical system is *continuous-time*.

1.1.2 Discrete-time Dynamics and Maps

A *map* is a transformation $\mathbf{M} : X \to X$ for which the following properties hold:

$$\begin{aligned} \mathbf{M}^0(\mathbf{\Gamma}) &:= I\mathbf{\Gamma} = \mathbf{\Gamma} \qquad \text{(Identity)} \\ \mathbf{M}^{m+n}(\mathbf{\Gamma}) &= \mathbf{M}^m \circ \mathbf{M}^n(\mathbf{\Gamma}) \quad \text{(Group)} \end{aligned} \tag{1.1.3}$$

where $\mathbf{M}^n := \overbrace{\mathbf{M} \circ \mathbf{M} \circ \cdots \circ \mathbf{M}}^{n \text{ times}}$; $m, n \in \mathbb{N}$. The group property of (1.1.3) expresses the idea that the dynamics can be restarted at any point $\mathbf{M}^n(\mathbf{\Gamma})$ along an orbit to end up at the same point $\mathbf{M}^{m+n}(\mathbf{\Gamma})$.

If the map is invertible, an additional property is added:

$$\mathbf{M}^{-1} \circ \mathbf{M} = \mathbf{M} \circ \mathbf{M}^{-1} = I. \tag{1.1.4}$$

Define $\mathbf{M}^{-n} := \overbrace{\mathbf{M}^{-1} \circ \mathbf{M}^{-1} \circ \cdots \circ \mathbf{M}^{-1}}^{n \text{ times}}$. Invertible maps that are continuous with continuous inverses are known as *homeomorphisms*. If, in addition to continuity, an invertible map is smooth [1] with smooth inverse, the map is known as a *diffeomorphism*.

1.1.3 Continuous-time Dynamics and Flows as Maps

The time evolution of many dynamical systems is governed by ordinary differential equations (ODE), which can be represented by an initial value problem

$$\begin{aligned} \frac{d}{dt}\mathbf{\Gamma}(t) &= \mathbf{F}(\mathbf{\Gamma}(t)) \\ \mathbf{\Gamma}(0) &\in X. \end{aligned} \tag{1.1.5}$$

[1] "Smooth" means at least C^1.

The smooth, real-valued vector field $\mathbf{F}$ assigns a point $\mathbf{\Gamma}$ in an open subset $U \subset X$ to a vector in the space tangent to X at $\mathbf{\Gamma}$, known as the *tangent space* $T_{\mathbf{\Gamma}}X$. The tangent space will be discussed further in Section 1.2.1.

Under general conditions, solutions of (1.1.5) exist and are unique for all time [3]. Moreover, the solution at each time $t \in \mathbb{R}$ depends smoothly on t and the initial condition $\mathbf{\Gamma}(0)$. The solution to (1.1.5), $\mathbf{\Gamma}(t)$, is called the *phase flow*. Another way of interpreting the solution is that the vector field $\mathbf{F}$ and initial condition $\mathbf{\Gamma}(0)$ induce the *time-t flow map* $\mathbf{\Phi}^t_{\mathbf{\Gamma}(0)}$,

$$\mathbf{\Phi}^t_{\mathbf{\Gamma}(0)}(\mathbf{\Gamma}(0)) = \mathbf{\Gamma}(t), \tag{1.1.6}$$

which satisfies the same properties as Equations (1.1.3, 1.1.4) along with differentiability (1.1.7),

$$\frac{d}{dt}\Big(\mathbf{\Phi}^t_{\mathbf{\Gamma}(0)}(\mathbf{\Gamma}(0))\Big) = \mathbf{F}\Big(\mathbf{\Phi}^t_{\mathbf{\Gamma}(0)}(\mathbf{\Gamma}(0))\Big). \tag{1.1.7}$$

See §35 in [3] for a more in-depth treatment of solutions to initial value problems on differentiable manifolds.

1.1.4 Continuous Flows Interrupted by Discrete Jumps

The systems considered in this work have two distinct periods of the dynamics: *free-flight* and *collision*. During free-flight, a system evolves continuously through a system of ordinary differential equations until a collision condition is met (e.g. two interacting disks come into contact). When a collision occurs, the system undergoes a discrete jump in state. Let $\mathbf{\Gamma}_i, \mathbf{\Gamma}_f$, denote the state vectors before and after a collision, and let $\mathbf{M}$ be the collision map that takes $\mathbf{\Gamma}_i$ to $\mathbf{\Gamma}_f$:

$$\mathbf{\Gamma}_f = \mathbf{M}(\mathbf{\Gamma}_i). \tag{1.1.8}$$

The flow map $\mathbf{\Phi}^{\tau_n-\tau_{n-1}}_{\mathbf{\Gamma}_f(\tau_{n-1})}$ takes the state immediately after the $(n-1)^{\text{st}}$ collision (at time τ_{n-1}) to the state immediately before the n^{th} collision (at time τ_n). Assuming n collisions take place up until time t, the state at time t is given by (1.1.9),

$$\mathbf{\Gamma}(t) = \mathbf{\Phi}^{t-\tau_n}_{\mathbf{\Gamma}_f(\tau_n)} \circ \mathbf{M}^{\tau_n} \circ \mathbf{\Phi}^{\tau_n-\tau_{n-1}}_{\mathbf{\Gamma}_f(\tau_{n-1})} \circ \cdots \circ \mathbf{\Phi}^{\tau_2-\tau_1}_{\mathbf{\Gamma}_f(\tau_1)} \circ \mathbf{M}^{\tau_1} \circ \mathbf{\Phi}^{\tau_1}_{\mathbf{\Gamma}(0)}(\mathbf{\Gamma}(0)). \tag{1.1.9}$$

1.1.5 A Motivating Example: The Bouncing Ball-Platform System

A simple example to illustrate the concept of dynamical systems with continuous flows interrupted by discrete jumps is the Bouncing Ball-Platform system [38, 40, 64]. Let $\mathbf{\Gamma} \coloneqq (x_\mathrm{b}\ v_\mathrm{b}\ x_\mathrm{p}\ v_\mathrm{p})^\top$ be the state vector of the ball-platform system illustrated in Figure 1.1. In-between impacts, the ball moves under the influence of gravity while the platform is constrained to move sinusoidally. Assuming unity mass of the ball, the ball-platform system between impacts can be modeled as an ODE system (1.1.10),

$$\dot{\mathbf{\Gamma}} = \mathbf{F}(\mathbf{\Gamma}) = \begin{pmatrix} v_\mathrm{b} \\ -g \\ v_\mathrm{p} \\ -\omega_\mathrm{p}^2 x_\mathrm{p} \end{pmatrix}, \tag{1.1.10}$$

where ω_p is the oscillation frequency of the platform. Equation (1.1.10), along with initial condition $\mathbf{\Gamma}_0 = (x_\mathrm{b}^0\ v_\mathrm{b}^0\ x_\mathrm{p}^0\ v_\mathrm{p}^0)^\top$, induces the time-$t$ flow map (1.1.11)

$$\mathbf{\Phi}^t_{\mathbf{\Gamma}_0}(\mathbf{\Gamma}_0) = \begin{pmatrix} x_\mathrm{b}^0 + v_\mathrm{b}^0 t - \frac{1}{2} g t^2 \\ v_\mathrm{b}^0 - gt \\ x_\mathrm{p}^0 \cos \omega_\mathrm{p} t + \frac{v_\mathrm{p}^0}{\omega_\mathrm{p}} \sin \omega_\mathrm{p} t \\ -x_\mathrm{p}^0 \omega_\mathrm{p} \sin \omega_\mathrm{p} t + v_\mathrm{p}^0 \cos \omega_\mathrm{p} t \end{pmatrix}. \tag{1.1.11}$$

It is assumed that the mass of the platform is much greater than the mass of the ball. Therefore, only the velocity of the ball is affected by impacts. The state vector after an

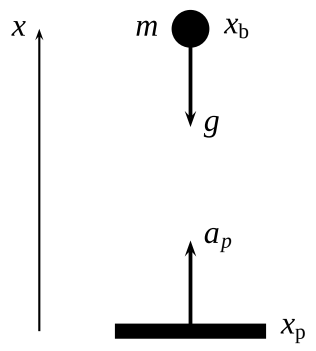


FIGURE 1.1. The bouncing ball-platform system. A ball, of mass m, accelerates due to gravity g between impacts with the platform, which accelerates sinusoidally $a_p = -\omega_p^2 x_p$.

impact $\mathbf{\Gamma}^+$ can be derived using conservation of momentum and the law of restitution[2] [64]:

$$\mathbf{\Gamma}^+ = \mathbf{M}(\mathbf{\Gamma}) = \begin{pmatrix} x_\mathrm{b} \\ -\alpha v_\mathrm{b} + (1+\alpha) v_\mathrm{p} \\ x_\mathrm{p} \\ v_\mathrm{p} \end{pmatrix} \tag{1.1.13}$$

where α is the restitution coefficient, $0 < \alpha \leq 1$.

For the ball-platform system defined by Equations (1.1.10, 1.1.13), the phase space $X \subset \mathbb{R}^4$. Figure 1.2 shows a few different modes of the dynamics. In this figure, the classic period doubling route to chaos seen in so many elementary treatments of nonlinear dynamics is present [46, 87]. What is particularly interesting about the ball-platform system is that the two maps (1.1.11) and (1.1.13) are linear in the components of the state vector and well-behaved (i.e. not chaotic), but the combination of the two maps can produce chaotic behavior. An important method for how to quantify such behavior is covered in

[2]The law of restitution states that the absolute value of the ball's velocity immediately after impact, V, is related to the absolute value of the ball's velocity immediately before impact, U, the absolute value of the platform's velocity at impact, W, and the coefficient of restitution, α ($0 < \alpha \leq 1$), via the following relationship:

$$V - W = -\alpha(U - W). \tag{1.1.12}$$

1.2 Tangent Space Dynamics and Lyapunov Exponents

One of the characteristics of chaotic systems is sensitivity to initial conditions. Trajectories which are initially infinitesimally-close to one another can separate exponentially along certain directions in phase space. The exponential rates at which such trajectories separate or approach one another are known as Lyapunov exponents, and are a very important tool for classifying nonlinear systems. The most conventional definition of a chaotic system is one that has (at least) one positive Lyapunov exponent, meaning that there is (at least) one direction in phase space in which two initially nearby trajectories separate exponentially in time.

1.2.1 Dynamics of Perturbations in Discrete-time

Consider an N-dimensional discrete time dynamical system[3] with C^1 map $\mathbf{M} : X \to X$ starting from the point $\mathbf{\Gamma}_0$:

$$\begin{aligned} \mathbf{\Gamma}_{n+1} &= \mathbf{M}(\mathbf{\Gamma}_n) \\ &= \mathbf{M}^n(\mathbf{\Gamma}_0). \end{aligned} \tag{1.2.1}$$

If the system is slightly perturbed, then the dynamics in (1.2.1) are necessarily modified

$$\mathbf{\Gamma}_{n+1} + \delta\mathbf{\Gamma}_{n+1} = \mathbf{M}(\mathbf{\Gamma}_n + \delta\mathbf{\Gamma}_n). \tag{1.2.2}$$

It is desired to find a relationship between $\delta\mathbf{\Gamma}_{n+1}$ and $\delta\mathbf{\Gamma}_0$, the initial perturbation. Assuming $\delta\mathbf{\Gamma}_{n+1}$ is small, Taylor's theorem is valid and the following linearization of (1.2.2) holds

$$\mathbf{\Gamma}_{n+1} + \delta\mathbf{\Gamma}_{n+1} = \mathbf{M}(\mathbf{\Gamma}_n) + J_{\mathbf{M}}(\mathbf{\Gamma}_n)\delta\mathbf{\Gamma}_n + \mathcal{O}(\|\delta\mathbf{\Gamma}_n\|^2), \tag{1.2.3}$$

[3]The following arguments can be extended to continuous flows by the time-one map: the discrete map that takes a continuous flow from some initial time to the state one time unit later.

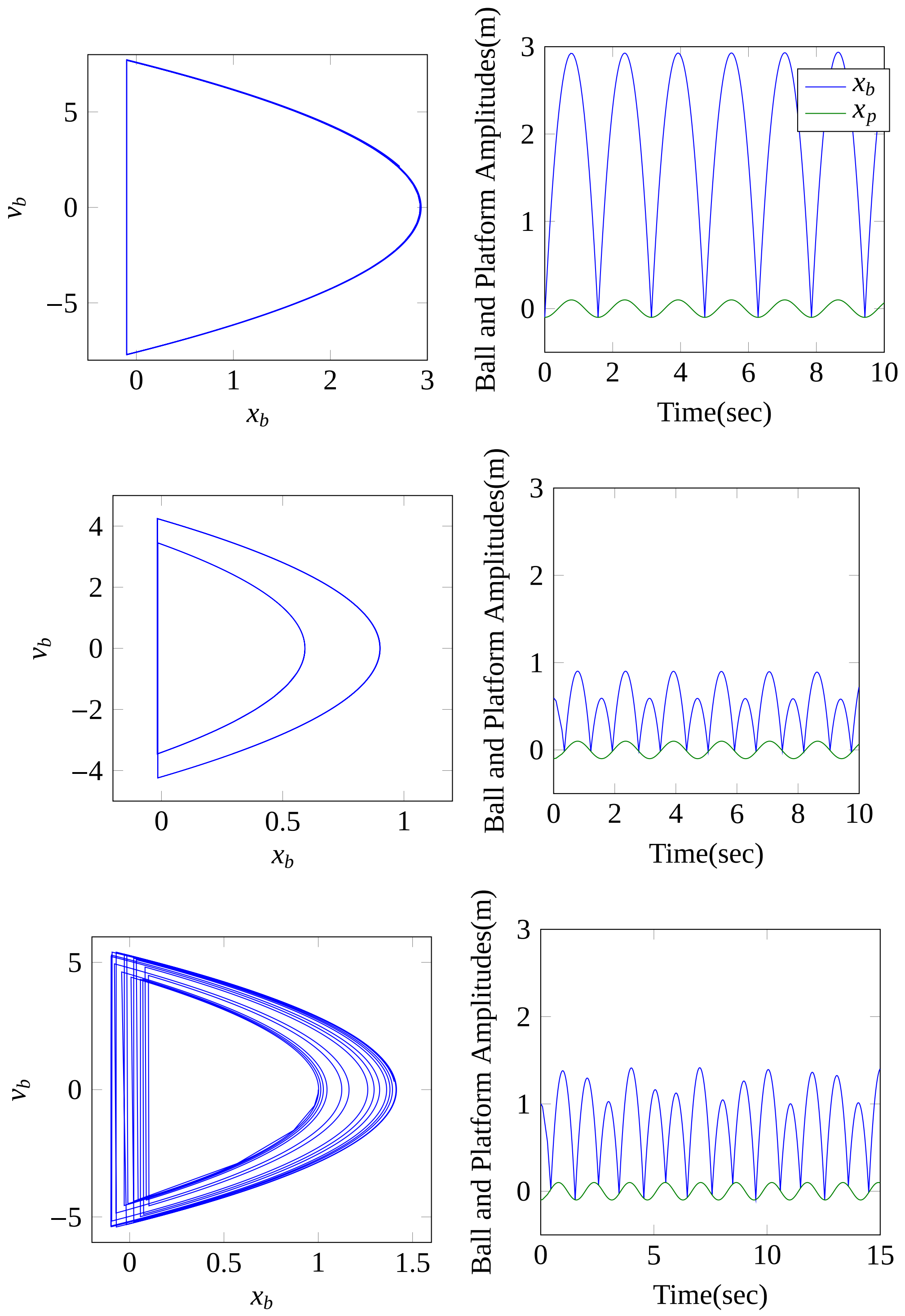


FIGURE 1.2. Demonstration of chaos in the ball-platform system ($\alpha = 1$). From top to bottom, periods 1, 2 and chaotic behaviors are observed. The figures at left show two-dimensional projections of the phase space: ball velocity v_{b} vs. ball position x_{b}. The figures at right show time-series data of ball and platform amplitudes, x_{b} and x_{p}, respectively.

where $J_{\mathbf{M}}(\mathbf{\Gamma}_n) := \frac{\partial \mathbf{M}}{\partial \mathbf{\Gamma}}\Big|_{\mathbf{\Gamma}=\mathbf{\Gamma}_n}$ is the Jacobian of $\mathbf{M}$ evaluated at $\mathbf{\Gamma}_n$. Combining (1.2.1) and (1.2.3) yields the evolution equation for an infinitesimal perturbation to the dynamical system (1.2.1),

$$\begin{aligned} \delta\mathbf{\Gamma}_{n+1} &= J_{\mathbf{M}}(\mathbf{\Gamma}_n)\delta\mathbf{\Gamma}_n + O(\|\delta\mathbf{\Gamma}_n\|^2) \\ &= J_{\mathbf{M}}(\mathbf{M}^n(\mathbf{\Gamma}_0))\cdots J_{\mathbf{M}}(\mathbf{M}(\mathbf{\Gamma}_0))J_{\mathbf{M}}(\mathbf{\Gamma}_0)\delta\mathbf{\Gamma}_0 + O(\|\delta\mathbf{\Gamma}_0\|^2), \\ &:= A_n(\mathbf{\Gamma}_0)\delta\mathbf{\Gamma}_0 + O(\|\delta\mathbf{\Gamma}_0\|^2). \end{aligned} \tag{1.2.4}$$

In the limit as $\delta\mathbf{\Gamma}_0$ approaches 0, the perturbation $\delta\mathbf{\Gamma}_n$ exists in the tangent space $T_{\mathbf{\Gamma}_n}X$. Recall that the tangent space $T_{\mathbf{\Gamma}}X$ is a finite-dimensional, in this case, N-dimensional, real Euclidean vector space with a positive definite norm $\|\mathbf{v}\| = \sqrt{\mathbf{v}^\top\mathbf{v}}$ and an inner product, $(\mathbf{v},\mathbf{w}) = \mathbf{v}^\top\mathbf{w}$. For a linear mapping $A : X \to \mathrm{GL}(N,\mathbb{R})$, the set of $N \times N$ real invertible matrices, the norm is

$$\|A\| = \sup_{\mathbf{v}\neq 0} \frac{\|A\mathbf{v}\|}{\|\mathbf{v}\|}.$$

To understand the evolution of perturbations under (1.2.4), picture an infinitesimal ball of perturbations centered around an initial condition $\mathbf{\Gamma}_0$. Over time, the shape and orientation of this ball will evolve due to the action of repeated application of the Jacobian of $\mathbf{M}$ (see Figure 1.3). The expansion and contraction of the principal axes can be quantified by the *singular values* of the matrix $A_n(\mathbf{\Gamma}_0) := J_{\mathbf{M}}(\mathbf{M}^n(\mathbf{\Gamma}_0))\cdots J_{\mathbf{M}}(\mathbf{\Gamma}_0)$, which are discussed in the following section. In the limit as $n \to \infty$, some principal axes will be exponentially stretched to ∞, while others exponentially contract to 0 [12, 71, 87].

1.2.2 Singular Value Decomposition

Let A be an $N \times N$ dimensional matrix with real entries. The matrix product $A^\top A$ is symmetric positive-definite and therefore orthogonally-diagonalizable [85]. This means that there exists an orthonormal matrix V ($V^\top V = VV^\top = I$) and a diagonal matrix, Σ, such that

$$A^\top A = V^\top \Sigma V.$$

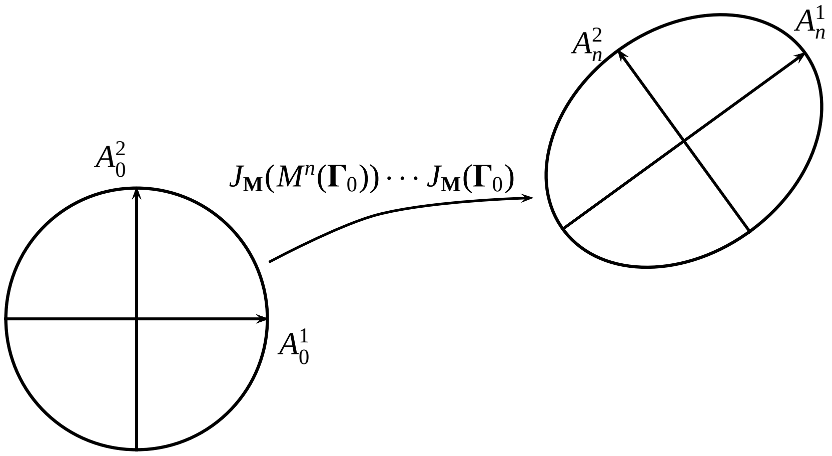

A_n^1
A_n^2
A_0^2
$J_{\mathbf{M}}(M^n(\mathbf{\Gamma}_0)) \cdots J_{\mathbf{M}}(\mathbf{\Gamma}_0)$
A_0^1

FIGURE 1.3. Two-dimensional slice of the evolution of a ball in tangent space under repeated mappings of $J_{\mathbf{M}}$. The figure demonstrates that principal axes can rotate and expand or contract under the action $A_n(\mathbf{\Gamma}_0) := J_{\mathbf{M}}(\mathbf{M}^n(\mathbf{\Gamma}_0)) \cdots J_{\mathbf{M}}(\mathbf{\Gamma}_0)$. The ball co-moves with, and is centered on, the reference trajectory in phase space. The principal axes at times 0 and n are $\{A_0^1, A_0^2\}$ and $\{A_n^1, A_n^2\}$, respectively.

The diagonal entries of Σ, $\lambda_1 \geq \cdots \geq \lambda_N$, are the eigenvalues of $A^\top A$. The *singular values* of A are the positive square roots of the eigenvalues of $A^\top A$,

$$\sigma_i = \sqrt{\lambda_i} \ \ 1 \leq i \leq N,$$

and

$$\begin{aligned} \|A\mathbf{v}_i\| &= \sqrt{(A\mathbf{v}_i, A\mathbf{v}_i)} \\ &= \sqrt{(A^\top A\mathbf{v}_i, \mathbf{v}_i)} \\ &= \sqrt{\lambda_i} \\ &= \sigma_i. \end{aligned}$$

The vector $\mathbf{v}_i$ is chosen as the i^{th} column vector of the matrix V, which is the eigenvector of $A^\top A$ associated with the eigenvalue λ_i. Since $A^\top A$ is symmetric positive-definite, all eigenvalues are real and positive [85]; as are the singular values of A. The columns of V, $\{\mathbf{v}_1, \cdots, \mathbf{v}_N\}$, form an orthonormal basis for $\mathbb{R}^N$, therefore the following is true ($1 \leq i < j \leq$

N):

$$\begin{aligned}(A\mathbf{v}_i, A\mathbf{v}_j) &= (A^\top A\mathbf{v}_i, \mathbf{v}_j)\\ &= \lambda_i(\mathbf{v}_i, \mathbf{v}_j)\\ &= 0.\end{aligned}$$

Therefore, $\{A\mathbf{v}_1, \cdots, A\mathbf{v}_N\}$ is pairwise-orthogonal and, if all σ_i are non-zero, forms another basis for $\mathbb{R}^N$. Define the set $\{\mathbf{u}_1, \cdots, \mathbf{u}_N\}$ by

$$\begin{aligned}\mathbf{u}_i &:= \frac{A\mathbf{v}_i}{\|A\mathbf{v}_i\|}\\ &= \frac{A\mathbf{v}_i}{\sigma_i}.\end{aligned} \tag{1.2.5}$$

From (1.2.5), it is clear that $A\mathbf{v}_i = \sigma_i\mathbf{u}_i$. Let U be the orthonormal matrix whose columns are $\{\mathbf{u}_1, \cdots, \mathbf{u}_N\}$, then it follows from (1.2.5) that $AV = U\Sigma$. Equivalently, A can be expressed as

$$A = U\Sigma V^\top. \tag{1.2.6}$$

A nice geometric interpretation of the singular value decomposition follows [12]. Let S be the unit sphere in $\mathbb{R}^N$,

$$S = \left\{\sum_{i=1}^{N} x_i\mathbf{v}_i \mid \sum_{i=1}^{N} x_i^2 = 1\right\}.$$

The mapping A applied to the set S can now be written as

$$\begin{aligned}A(S) &= \left\{A\mathbf{x}|\mathbf{x} \in S\right\}\\ &= \left\{\sum_{i=1}^{N} x_i\sigma_i\mathbf{u}_i \mid \sum_{i=1}^{N} x_i^2 = 1\right\}.\end{aligned}$$

The matrix A acting on the set S transforms S to an ellipsoid with principal axes $A\mathbf{v}_i = \sigma_i\mathbf{u}_i$. The original basis $\{\mathbf{v}_i\}$ for S has been rotated and stretched.

The preceding argument matches the rationale for the stretching and contraction of perturbation vectors under repeated mappings of the Jacobian $J_\mathbf{M}$ from Equation (1.2.4).

However, there are some technical concerns. It may be that the limit as $n \to \infty$ of $A_n(\mathbf{\Gamma}_0) := J_{\mathbf{M}}(\mathbf{M}^n(\mathbf{\Gamma}_0)) \cdots J_{\mathbf{M}}(\mathbf{\Gamma}_0)$ is not well-defined. There may, however, be other limits of interest that are well-defined. As will be demonstrated, one such set of limits is the set of *Lyapunov exponents*, $\{L^{(1)}(\mathbf{\Gamma}_0), \cdots, L^{(r)}(\mathbf{\Gamma}_0)\}$ $(1 \leq r \leq N)$,

$$L^{(i)}(\mathbf{\Gamma}_0) \quad := \quad \lim_{n\to\infty} \frac{1}{n} ||A_n(\mathbf{\Gamma}_0)\delta\mathbf{\Gamma}_0^i||, \tag{1.2.7}$$

for an appropriately chosen initial perturbation $\delta\mathbf{\Gamma}_0^i$ and $1 \leq i \leq r$. The set of Lyapunov exponents is commonly referred to as the Lyapunov spectrum. It should be noted that some exponents may have multiplicities greater than 1, hence $r \leq N$[4]. The existence of the Lyapunov spectrum is a consequence of the Multiplicative Ergodic Theorem of Oseledec, which is discussed in the next section.

1.2.3 The Multiplicative Ergodic Theorem of Oseledec

For practical purposes, the growth rates of volumes of subspaces spanned by perturbation vectors, not the growth rates of the perturbation vectors themselves, are considered. Assume n is large, and let $\sigma_n^{(1)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(r)}(\mathbf{\Gamma})$ be the r distinct singular values of $A_n(\mathbf{\Gamma})$. Define

$$\phi_n^{(k)}(\mathbf{\Gamma}) = \log\Big(\sup_{\dim W=k} \Big|\det(A_n(\mathbf{\Gamma})|_W)\Big|\Big). \tag{1.2.8}$$

In (1.2.8), $\sup_{\dim W=k} |\det(A_n(\mathbf{\Gamma})|_W)|$ is interpreted as the supremum over volume elements of k-dimensional subspaces W spanned by columns of $A_n(\mathbf{\Gamma})$. The supremum is obtained when W is spanned by the eigenvectors of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ corresponding to the k largest singular values of $A_n(\mathbf{\Gamma})$ [12]. This step requires some justification. From Equation (1.2.6) in the

[4]Non-degenerate Lyapunov exponents and singular values will be referenced by superscripts, as in $L^{(j)}$. Degenerate quantities will be referenced by a subscript, as in L_j. Assuming the j^{th} distinct exponent has multiplicity $m(L^{(j)})$, then

$$L^{(j)} = L_{f(j)+1} = \cdots = L_{f(j)+m(L^{(j)}))},$$

where $f(j) = \sum_{i=1}^{j-1} m(L^{(i)})$ is the sum of all multiplicities of exponents with index smaller than j.

last section, $A_n(\mathbf{\Gamma})$ has a singular value decomposition from which it follows that[5]

$$\begin{aligned}|\det A_n(\mathbf{\Gamma})| &= |\det U(\mathbf{\Gamma})\det\Sigma(\mathbf{\Gamma})\det V(\mathbf{\Gamma})^\top| \\ &= |\det\Sigma(\mathbf{\Gamma})| \\ &= (\sigma_n^{(1)}(\mathbf{\Gamma}))^{m(\sigma_n^{(1)}(\mathbf{\Gamma}))}\cdots(\sigma_n^{(r)}(\mathbf{\Gamma}))^{m(\sigma_n^{(r)}(\mathbf{\Gamma}))}.\end{aligned}$$

The multiplicity of each singular value, $\sigma_n^{(i)}(\mathbf{\Gamma})$, is represented by $m(\sigma_n^{(i)}(\mathbf{\Gamma}))$. The determinant of the restriction of $A_n(\mathbf{\Gamma})$ to k-dimensional subspaces is then the product of k singular values. This quantity is maximized when the space spanned by the columns of V associated with the k largest singular values of $A_n(\mathbf{\Gamma})$ is considered[6].

The k largest singular values of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ are $\sigma_n^{(1)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(j)}(\mathbf{\Gamma})$, where j is the number of distinct singular values up to $\sigma_k(\mathbf{\Gamma})$:

$$\sum_{i=1}^{j} m(\sigma_n^{(i)}(\mathbf{\Gamma})) = k.$$

From the previous arguments, $\phi_n^{(k)}(\mathbf{\Gamma})$ is reduced to

$$\begin{aligned}\phi_n^{(k)}(\mathbf{\Gamma}) &= \log((\sigma_n^{(1)}(\mathbf{\Gamma}))^{m(\sigma_n^{(1)}(\mathbf{\Gamma}))}\cdots(\sigma_n^{(j)}(\mathbf{\Gamma}))^{m(\sigma_n^{(j)}(\mathbf{\Gamma}))}) \\ &= \sum_{i=1}^{j} m(\sigma_n^{(i)}(\mathbf{\Gamma}))\log\sigma_n^{(i)}(\mathbf{\Gamma}).\end{aligned} \tag{1.2.10}$$

It is desired that the average growth rate of $\phi_n^{(k)}(\mathbf{\Gamma})$ is well-defined as $n \to \infty$; that is,

$$\lim_{n\to\infty}\frac{1}{n}\phi_n^{(k)}(\mathbf{\Gamma}) = \phi_*^k(\mathbf{\Gamma}), \tag{1.2.11}$$

[5]The singular values used in subsequent arguments are such that $\sigma_n^{(1)}(\mathbf{\Gamma}) > \sigma_n^{(2)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(r)}(\mathbf{\Gamma})$.

[6]Due to multiplicity, k can only take values from the sequence $\{J_1(\mathbf{\Gamma}), J_2(\mathbf{\Gamma}), \cdots, J_r(\mathbf{\Gamma})\}$, where the elements of the sequence $\{J_i\}$ are related to the multiplicity of the singular values

$$\begin{aligned}J_0 &:= 0 \\ J_1 &:= m(\sigma_n^{(1)}(\mathbf{\Gamma})) \\ J_2 &:= m(\sigma_n^{(1)}(\mathbf{\Gamma})) + m(\sigma_n^{(2)}(\mathbf{\Gamma})) \\ &\vdots \\ J_r &:= \sum_{i=1}^{r} m(\sigma_n^{(i)}(\mathbf{\Gamma})).\end{aligned} \tag{1.2.9}$$

In other words, only volumes spanned by all eigenvectors of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ with degenerate eigenvalue are well-defined [21].

for some measurable $\phi_*^k(\mathbf{\Gamma})$. This fact follows from a corollary (see Appendix A.1) of the Subadditive Ergodic Theorem due to Kingman [54, 86].

Equation (1.2.11) gives the result that the long-time average growth rate of subspace volumes spanned by orthonormal eigenvectors of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ converges to a limit. This result can now be used to motivate a general result due to Oseledec [70] (see [12, 77] for more details).

Theorem 1 (The Multiplicative Ergodic Theorem for Not-Necessarily Invertible Transformations[7] [12, 70, 77])**.** Let $\mathbf{M}$ be an N-dimensional (not necessarily invertible) measure-preserving transformation on a probability space (X, μ). Let $A : X \to GL(N, \mathbb{R})$ be measurable with

$$\int_X \log^+(\|A\|)d\mu < \infty,$$
$$\log^+(\cdot) := \max(0, \log(\cdot)).$$

Require further that $A^{-1}(\mathbf{\Gamma})$ is almost surely invertible. Define

$$A_n(\mathbf{\Gamma}) := A(\mathbf{M}^{n-1}(\mathbf{\Gamma}))\cdots A(\mathbf{M}(\mathbf{\Gamma}))A(\mathbf{\Gamma}).$$

For $\mathbf{\Gamma} \in X$ there almost surely exist numbers (known as the *Lyapunov exponents*)

$$L^{(1)}(\mathbf{\Gamma}) > \cdots > L^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}), \tag{1.2.14}$$

[7]When the transformations are invertible, the tangent space can be split into a direct sum known as the *Oseledec splitting*:

$$T_{\mathbf{\Gamma}}X = \mathbb{R}^N = E^{(1)}(\mathbf{\Gamma}) \oplus E^{(2)}(\mathbf{\Gamma}) \oplus \cdots \oplus E^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}). \tag{1.2.12}$$

The elements $E^{(i)}(\mathbf{\Gamma})$ are *covariant*,

$$E^{(i)}(\mathbf{M}(\mathbf{\Gamma})) = A(\mathbf{\Gamma})E^{(i)}(\mathbf{\Gamma}), \tag{1.2.13}$$

but they are not pairwise orthogonal, in general: for $1 \le i < j \le r(\mathbf{\Gamma})$, some $\mathbf{u} \in E^{(i)}(\mathbf{\Gamma})$ and $\mathbf{v} \in E^{(j)}(\mathbf{\Gamma})$, then $\mathbf{u}^\top\mathbf{v} \neq 0$, in general.
The version of the theorem discussed here is chosen to illustrate the existence of the Lyapunov exponents, not the structure of the tangent space decomposition. The notion of covariant Lyapunov vectors and their relation to (1.2.12) will be discussed in Section 2.3.2. For more details on the Oseledec splitting, see [12, 20, 70, 73, 77] .

and a sequence of subspaces (known as the *Oseledec filtration*)[8]

$$\mathbb{R}^N = V_+^{(1)}(\mathbf{\Gamma}) \supsetneq \cdots \supsetneq V_+^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}) \supsetneq V_+^{(r(\mathbf{\Gamma})+1)}(\mathbf{\Gamma}) = \{\emptyset\}, \tag{1.2.19}$$

such that

(i) $\lim_{n\to\infty} \frac{1}{n} \log \|A_n(\mathbf{\Gamma})\mathbf{v}\|$ exists and is equal to $L^{(i)}(\mathbf{\Gamma})$ when $\mathbf{v} \in V_+^{(i)}(\mathbf{\Gamma}) \setminus V_+^{(i+1)}(\mathbf{\Gamma})$.

(ii) $A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) = V_+^{(i)}(\mathbf{M}(\mathbf{\Gamma}))$.

(iii) $r(\mathbf{\Gamma}), L^{(i)}(\mathbf{\Gamma})$ and $\dim V_+^{(i)}(\mathbf{\Gamma})$ are measurable.

(iv) $r(\mathbf{\Gamma}), L^{(i)}(\mathbf{\Gamma})$ and $\dim V_+^{(i)}(\mathbf{\Gamma})$ are constant along orbits of $\mathbf{M}$.

(v) $\lim_{n\to\infty} \log|\det A_n(\mathbf{\Gamma})| = \sum_{i=1}^{r(\mathbf{\Gamma})} L^{(i)}(\mathbf{\Gamma})(\dim V_+^{(i)}(\mathbf{\Gamma}) - \dim V_+^{(i+1)}(\mathbf{\Gamma}))$.

In the above theorem, $r(\mathbf{\Gamma})$ is the number of non-degenerate Lyapunov exponents of the transformation $\mathbf{M}$ and each exponent $L^{(i)}(\mathbf{\Gamma})$ has multiplicity $m(L^{(i)}(\mathbf{\Gamma})) = \dim V_+^{(i)}(\mathbf{\Gamma}) - \dim V_+^{(i+1)}(\mathbf{\Gamma})$. A proof of Theorem 1 expanding upon arguments in Choe [12] is presented in Appendix A.1.

For practical consideration (e.g. computing the Lyapunov exponents numerically), a few things should be discussed here. As already alluded to in the previous section, the linear transformation, $A(\mathbf{\Gamma})$, in Theorem 1 is the Jacobian matrix of the transformation $\mathbf{M}$

[8]When the transformation $\mathbf{M}$ is invertible, another filtration can be constructed:

$$\mathbb{R}^N = V_-^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}) \supsetneq \cdots \supsetneq V_-^{(1)}(\mathbf{\Gamma}) \supsetneq V_-^{(0)}(\mathbf{\Gamma}) = \{\emptyset\}, \tag{1.2.15}$$

with associated Lyapunov exponents

$$L_-^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}) > \cdots > L_-^{(1)}(\mathbf{\Gamma}), \tag{1.2.16}$$

using the transformation $\mathbf{M}^{-1}$ instead of $\mathbf{M}$. The exponents (1.2.14) and (1.2.16) are related by

$$L^{(i)}(\mathbf{\Gamma}) = L_-^{(r(\mathbf{\Gamma})-i+1)}(\mathbf{\Gamma}), \tag{1.2.17}$$

for $1 \le i \le r(\mathbf{\Gamma})$. Combining (1.2.15) and (1.2.19), the elements $E^{(i)}(\mathbf{\Gamma})$ ($1 \le i \le r(\mathbf{\Gamma})$) of the Oseledec splitting (1.2.12) are constructed as [12, 77]

$$E^{(i)}(\mathbf{\Gamma}) = V_+^{(i)}(\mathbf{\Gamma}) \cap V_-^{(i)}(\mathbf{\Gamma}). \tag{1.2.18}$$

evaluated at $\mathbf{\Gamma}$. It is important to discuss the filtration of the tangent space $T_{\mathbf{\Gamma}}X = \mathbb{R}^N$ (1.2.19). As defined, the subspace $V_+^{(1)}(\mathbf{\Gamma}) \setminus V_+^{(2)}(\mathbf{\Gamma})$ is spanned by $m(L^{(1)}(\mathbf{\Gamma})) = \dim V_+^{(1)}(\mathbf{\Gamma})$ vectors; the ones corresponding to the fastest growing perturbations under repeated mappings of A. The volume spanned by these perturbation vectors grow exponentially with rate $m(L^{(1)}(\mathbf{\Gamma}))L^{(1)}(\mathbf{\Gamma})$. The volume of a subspace spanned by vectors in $[V_+^{(1)}(\mathbf{\Gamma}) \setminus V_+^{(2)}(\mathbf{\Gamma})] \cup [V_+^{(2)}(\mathbf{\Gamma}) \setminus V_+^{(3)}(\mathbf{\Gamma})]$ grows at a rate of $m(L^{(1)}(\mathbf{\Gamma}))L^{(1)}(\mathbf{\Gamma}) + m(L^{(2)}(\mathbf{\Gamma}))L^{(2)}(\mathbf{\Gamma})$ (recall Equations (1.2.10) and (1.2.11)). Through these two steps, the first two exponents can be computed. This process can be followed iteratively to calculate the entire spectrum of Lyapunov exponents. The numerical implementation of this is discussed in Section 2.2.2.

Theorem 1 guarantees the existence of Lyapunov exponents under very general circumstances. If the state $\mathbf{\Gamma}_n$ samples a "large part" of the space X over time, then the quantities $r, V_+^{(i)}, L^{(i)}$ are independent of the initial condition $\mathbf{\Gamma}_0$ [12, 71, 78]. This notion of phase space sampling can be made more rigorous through the ideas of *ergodicity* and *mixing* [35, 53, 96]. These ideas will be discussed in later sections, but a simple definition for ergodic measures/transformations is given here as a motivation,

Definition 1.2.20. (Ergodic measure/transformation) [12] Let $(X, \mathcal{B}, \mu)$ be a probability space and $M : X \to X$. A measure μ is **ergodic** for M (equivalently, M is an **ergodic transformation**) if all M-invariant measurable sets have measure 0 or 1.

1.3 Further Reading

Although it lacks in mathematical rigor, a very nice introduction to dynamical systems is provided in Strogatz [87]. This book provides nice intuitive developments of many important aspects of dynamics, along with copious physically-relevant examples. For a more rigorous treatment, Guckenheimer and Holmes [40] gives a thorough introduction to the mathematical principles and arguments that are elementary in the study of nonlinear dynamical systems. Other nice introductions can be found in Ott [71] (particularly Sections 1.3, 1.5 and 2.3), Wiggins [97] and Gaspard [35] (Sections 1.1 and 1.2). For more on the dynamics of continuous flows interrupted by discrete jumps (and other hybrid dynamical systems), see Goebel *et. al* [38].

Much of the development of tangent space dynamics and Lyapunov exponents is taken from Choe [12]. This resource was found to be invaluable in understanding the definition and existence proofs for Lyapunov exponents. It was also greatly beneficial in building intuition about ergodic systems; in regard to both theoretical and computational concerns. For more information regarding the existence of the singular value decomposition of a matrix, see [39] (Section 2.5.3) and [93] (Lectures 4 and 5).

Chapter 2
Numerical Considerations and Algorithms

This chapter develops the motivation and algorithms needed to compute the full spectrum of Lyapunov exponents. A motivating discussion is included to demonstrate the challenges encountered when trying to implement a numerical procedure for the exponent computation. The algorithm of Benettin *et. al* [5, 98] is presented and used to approximate the full spectrum of exponents for a few well-known examples: the continuous-time Lorenz system and the Bouncing Ball-Platform system which can, under assumptions that will be discussed, be reduced to a discrete-time system.

Benettin's algorithm approximates the Lyapunov exponents, but vectors spanning the unstable, neutral and stable subspaces are also output during the computation. The relationship of these vectors to the Oseledec filtration and splitting is discussed in detail. In general, these vectors do not directly coincide with elements of the Oseledec splitting, therefore the method of Ginelli is developed to recover vectors that do [36, 37].

For the systems considered in this work, the transformations are ergodic. Ergodicity guarantees the values computed for the Lyapunov exponents are independent of the initial condition, $\mathbf{\Gamma}_0$ [12, 35, 71]. Therefore, unless explicitly stated otherwise, $L_i(\mathbf{\Gamma}) := L_i$ will be used to denote the i^{th} Lyapunov exponent.

2.1 Notional "Numerical" Scheme for Computing Lyapunov Exponents

Let X be an N-dimensional compact C^∞ Riemannian manifold, and $\mathbf{M} : X \to X$ be a C^1 ergodic transformation with Jacobian matrix $J_{\mathbf{M}}$. Consider the j^{th} (non-distinct) Lyapunov exponent

$$L_j = L^{(i)} = \lim_{n\to\infty} \frac{1}{n} \log \|A_n \mathbf{v}\|, \quad \mathbf{v} \in [V_+^{(i)} \setminus V_+^{(i+1)}], \tag{2.1.1}$$

where, recalling the sequence $\{J_m\}$ from Equation (1.2.9), $J_{i-1} < j \leq J_i$[1]. Additionally, it is again assumed that the exponents are ordered largest to smallest: $L_1 \leq \cdots \leq L_N$. The perturbation $\mathbf{v}$ in 2.1.1 can be interpreted as the component of a perturbation vector $\delta\boldsymbol{\Gamma}$ that is orthogonal to all elements of the subspace $V_+^{(i+1)}$ defined from Equation (1.2.19). For beginning the discussion of the numerical computation of L_j, consider the following approximation $\overline{L}_j$ for large n

$$\begin{aligned}\overline{L}_j &= \frac{1}{n}\log\|A_n\mathbf{v}\| \\ &:= \frac{1}{2n}\log\|\mathbf{v}^\top H_n\mathbf{v}\|, \end{aligned} \tag{2.1.2}$$

where $H_n := A_n^\top A_n$. Because H_n is symmetric positive-definite, the eigenvalues $h_{n,j}$ are real and positive and, perhaps more importantly, H_n is orthogonally diagonalizable (see Section 1.2.2). This means that the vector $\mathbf{v}$ can be chosen as the orthonormal eigenvector associated with the j^{th} eigenvalue of H_n, $h_{n,j}$. It follows that

$$\overline{L}_j = \frac{1}{2n}\log h_{n,j}. \tag{2.1.3}$$

Therefore, any perturbation vector $\delta\boldsymbol{\Gamma}$ can be expanded in the orthonormal basis defined by the eigenvectors of H_n, $\{\mathbf{u}_{n,j} | 1 \leq j \leq N\}$,

$$\begin{gathered}\delta\boldsymbol{\Gamma} = \sum_{i=1}^{N} a_i\mathbf{u}_{n,i}, \\ \sum_{i=1}^{N} a_i^2 = \|\delta\boldsymbol{\Gamma}\|^2. \end{gathered} \tag{2.1.4}$$

Now, the strategy becomes selecting the coefficients a_i such that the full spectrum of approximate exponents $\{\overline{L}_j | 1 \leq j \leq N\}$ can be computed. If $\delta\boldsymbol{\Gamma}$ is chosen arbitrarily, the following is true given Equations (2.1.3) and (2.1.4):

$$\begin{aligned}\delta\boldsymbol{\Gamma}^\top H_n\delta\boldsymbol{\Gamma} &= \sum_{i=1}^{N} a_i^2 h_{n,i} \\ &= \sum_{i=1}^{N} a_i^2 \exp[2n\overline{L}_i]. \end{aligned} \tag{2.1.5}$$

[1]Because $\mathbf{M}$ is assumed ergodic, the dependence on initial condition $\boldsymbol{\Gamma}_0$ is dropped.

In Equation (2.1.5), if n is chosen large enough, all terms except the last are insignificant in the sum due to the exponential weighting,

$$\delta\mathbf{\Gamma}^{\top} H_n \delta\mathbf{\Gamma} \approx a_1^2 \exp[2n\overline{L}_1]. \tag{2.1.6}$$

In principle, everything needed to compute an approximation to the largest Lyapunov exponent, $\overline{L}_1$, is available. After computing $\overline{L}_1$, choose a new perturbation vector $\delta\mathbf{\Gamma}'$ such that a_1, the component along an eigenvector associated with the largest eigenvalue of H_n, is zero. Repeating the calculation in Equation (2.1.6) gives the approximation to the next Lyapunov exponent, $\overline{L}_2$. This process can be followed iteratively to compute the remaining $N-2$ exponents.

The “algorithm” outlined in the previous paragraph makes sense in theory, however in dealing with large n, finite precision available computationally in matrix multiplication is an issue. Additionally, there is the question of how to select the initial perturbations when there is no *a priori* knowledge of H_n. Moreover, due to the exponential growth of the perturbation vectors, any perturbation with non-zero component lying along the direction of maximal growth (i.e. a vector in the subspace $V_+^{(1)} \setminus V_+^{(2)}$) will collapse onto that direction [98]. Jacobian matrices are not orthogonal, in general. Therefore, a vector that is initially orthogonal to a particular element, $V_+^{(i)}$, of the filtration from Equation (1.2.19) may not remain orthogonal to it for all time. The perturbation vectors can be periodically reorthonormalized to ensure that all vectors don’t align along the direction of maximal growth. This approach is known as *Benettin’s algorithm* [5, 98] and will be developed in the next section.

2.2 *QR* Factorization and Benettin’s Algorithm

From Theorem 1, particularly the definition of the filtration in Equation (1.2.19), it is clear that any computation of the full spectrum of Lyapunov exponents requires tracking growth rates of subspaces of dimension 1 to N, depending upon the exponent multiplicity. In practice, knowledge of the matrix H_n and its eigenvectors will not be present, therefore a new approach is needed. There is a technique from linear algebra that is very useful in this setting: *QR factorization*.

2.2.1 *QR* Factorization and Gram-Schmidt Orthonormalization

Assume B is an $m \times m$ matrix of full rank[2], then it is true that the columns of B define a filtration similar to the one constructed in Theorem 1 (see Equation (1.2.19)),

$$\{0\} = \langle \mathbf{b}_0 \rangle \subsetneq \langle \mathbf{b}_1 \rangle \subsetneq \langle \mathbf{b}_1, \mathbf{b}_2 \rangle \subsetneq \cdots \subsetneq \langle \mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_m \rangle = \mathbb{R}^m,$$

where $\langle \cdot, \cdots, \cdot \rangle$ means the space spanned by the enclosed vectors. For the purposes of this discussion, think of the matrix B as being a product of Jacobian matrices, as in Equation (1.2.4).

The idea behind QR factorization of B is to find an orthonormal matrix Q the columns of which have the same span as the corresponding columns of B:

$$\langle \mathbf{b}_1, \cdots, \mathbf{b}_j \rangle = \langle \mathbf{q}_1, \cdots, \mathbf{q}_j \rangle \quad 1 \leq j \leq m.$$

This amounts to the construction of an $m \times m$ orthonormal matrix, Q, and an $m \times m$ upper-triangular matrix, R, such that:

$$\begin{aligned} \mathbf{b}_1 &= r_{1,1}\mathbf{q}_1, \\ \mathbf{b}_2 &= r_{1,2}\mathbf{q}_1 + r_{2,2}\mathbf{q}_2 \\ &\vdots \\ \mathbf{b}_m &= \sum_{j=1}^{m} r_{j,m}\mathbf{q}_j. \end{aligned} \tag{2.2.1}$$

The matrix Q, as stated previously, is orthonormal. Therefore, the column vectors of Q must be scaled so that volume is preserved. The matrix terms of the upper-triangular matrix

[2]For the purposes of this work, this assumption is valid since perturbation elements of each nested subspace of the filtration have a well-defined growth rate.

R has encoded in it the necessary scaling. The matrix Q can be constructed algebraically from Equation (2.2.1),

$$\begin{aligned}\mathbf{q}_1 &= \frac{\mathbf{b}_1}{r_{1,1}}\\ \mathbf{q}_2 &= \frac{\mathbf{b}_2 - r_{1,2}\mathbf{q}_1}{r_{2,2}}\\ &\vdots\\ \mathbf{q}_m &= \frac{\mathbf{b}_m - \sum_{i=1}^{m-1} r_{i,m}\mathbf{q}_i}{r_{m,m}}.\end{aligned} \tag{2.2.2}$$

The process outlined in Equation (2.2.2) is an iterative one; the computation used for $\mathbf{q}_j$ is used in $\mathbf{q}_{j+1}, \cdots, \mathbf{q}_m$. At the j^{th} step it is desired to find a unit vector that is orthogonal to all of the previous $j-1$ vectors. To accomplish this, the orthonormalization technique known as *Classical Gram-Schmidt* can be used [39, 93]:

$$\begin{aligned}\mathbf{v}_j &= \mathbf{b}_j - (\mathbf{q}_1^\top \mathbf{b}_j)\mathbf{q}_1 - \cdots - (\mathbf{q}_{j-1}^\top \mathbf{b}_j)\mathbf{q}_{j-1},\\ \mathbf{q}_j &= \frac{\mathbf{v}_j}{\|\mathbf{v}_j\|}.\end{aligned} \tag{2.2.3}$$

Combining Equations (2.2.2) and (2.2.3) gives the coefficients of the matrix R,

$$\begin{aligned}r_{i,j} &= \mathbf{q}_i^\top \mathbf{b}_j \quad i \neq j\\ r_{j,j} &= \left\| \mathbf{b}_j - \sum_{i=1}^{j-1} r_{i,j}\mathbf{q}_i \right\|.\end{aligned}$$

The computational algorithm for Classical Gram-Schmidt is shown in Algorithm 1.

Classical Gram-Schmidt is numerically unstable due to rounding errors, meaning that the output vectors can be significantly non-orthogonal [39]. The instability is due to the fact that Classical Gram-Schmidt uses a single orthogonal projection of rank $m-(j-1)$ [93]

$$\begin{aligned}\mathbf{v}_j &= P_j \mathbf{b}_j\\ &= \mathbf{b}_j - Q_{j-1}Q_{j-1}^\top \mathbf{b}_j,\end{aligned}$$

where Q_{j-1} is the $m \times (j-1)$ matrix whose columns are the vectors $\mathbf{q}_1, \cdots, \mathbf{q}_{j-1}$. There is a minor fix that resolves this instability.

Algorithm 1 Classical Gram-Schmidt Orthonormalization (CGS) ($\{\mathbf{b}_j\}$)

```
for $j \leftarrow 1 : m$ do
    $\mathbf{v}_j = \mathbf{b}_j$
    for $i = 1 : j - 1$ do
        $r_{i,j} = \mathbf{q}_i^\top \mathbf{b}_j$
        $\mathbf{v}_j = \mathbf{v}_j - r_{i,j}\mathbf{q}_i$
    end for
    $r_{j,j} = \|\mathbf{v}_j\|$
    $\mathbf{q}_j = \frac{\mathbf{v}_j}{r_{j,j}}$
end for
return $\{\mathbf{q}_j\}, \{r_{i,j}\}$
```

The *Modified Gram-Schmidt* algorithm computes the same result as Classical Gram-Schmidt algebraically, except Modified Gram-Schmidt uses $j-1$ projections of rank $m-1$. To demonstrate, consider the orthogonal projector

$$P_{\perp \mathbf{q}_j} = I - \mathbf{q}_j \mathbf{q}_j^\top. \tag{2.2.4}$$

The projector of Equation (2.2.4) removes the component of a vector that lies along $\mathbf{q}_j$. Projectors of the type in Equation (2.2.4) can be applied successively to yield

$$P_j = P_{\perp \mathbf{q}_{j-1}} P_{\perp \mathbf{q}_{j-2}} \cdots P_{\perp \mathbf{q}_1}. \tag{2.2.5}$$

Applying the projectors gives

$$\begin{aligned} \mathbf{v}_j^{(1)} &= \mathbf{b}_j \\ \mathbf{v}_j^{(2)} &= P_{\perp \mathbf{q}_1} \mathbf{q}_1 \mathbf{v}_j^{(1)} = \mathbf{v}_j^{(1)} - \mathbf{q}_1 \mathbf{q}_1^\top \mathbf{v}_j^{(1)} \\ &\vdots \\ \mathbf{v}_j = \mathbf{v}_j^{(j)} &= P_{\perp \mathbf{q}_{j-1}} \mathbf{v}_j^{(j-1)} = \mathbf{v}_j^{(j-1)} - \mathbf{q}_{j-1} \mathbf{q}_{j-1}^\top \mathbf{v}_j^{(j-1)}. \end{aligned} \tag{2.2.6}$$

When the Modified Gram-Schmidt algorithm is implemented, $P_{\perp \mathbf{q}_i}$ can be applied to each $\mathbf{v}_j^{(i)}$ for each $j > i$ immediately after $\mathbf{q}_i$ is known (see Algorithm 2). Modified Gram-Schmidt is preferable to Classical Gram-Schmidt for implementation because the algorithm is more numerically stable [93].

Algorithm 2 Modified Gram-Schmidt Orthonormalization (MGS)($\{\mathbf{b}_i\}$)

```
for $i = 1 : m$ do
    $\mathbf{v}_i = \mathbf{b}_i$
end for
for $i = 1 : m$ do
    $r_{i,i} = ||\mathbf{v}_i||$
    $\mathbf{q}_i = \frac{\mathbf{v}_i}{r_{i,i}}$
    for $j \leftarrow i + 1 : m$ do
        $r_{i,j} = \mathbf{q}_i^\top \mathbf{v}_j$
        $\mathbf{v}_j = \mathbf{v}_j - r_{i,j}\mathbf{q}_i$
    end for
end for
return $\{\mathbf{q}_i\}, \{r_{i,j}\}$
```

From Algorithm 2, an approach for approximating the Lyapunov exponents from non-orthogonal matrices, A_n, composed of Jacobian products begins to become clear. As previously stated, due to the exponential divergence observed in many nonlinear dynamical systems, a perturbation with any component along the direction of maximal instability will quickly align along that direction [98]. If the Modified Gram-Schmidt algorithm were applied at prescribed intervals, before collapse onto the direction of maximal growth, the average values of exponential growth rates of j-dimensional volumes spanned by the first j columns of A_n could be computed. This approach is attributed to Benettin *et. al* [5, 98].

2.2.2 Benettin's Algorithm

Again, let X be an N-dimensional compact Riemannian manifold, and $\mathbf{M} : X \to X$ be a C^1 ergodic transformation with Jacobian matrix $A := J_{\mathbf{M}}$. Further, let A_n be as defined in Equation (1.2.4). The algorithm of Benettin *et. al* is used to compute the full spectrum of Lyapunov exponents for $\mathbf{M}$ [5, 98]. The algorithm is basically an extension of QR factorization, where the average exponential growth of perturbation volumes is computed using the $\{r_{i,i}\}$. The idea is to start with a set of orthonormal vectors that span the full tangent space, evolve these perturbations through the linearized dynamics Equation (1.2.4), and then iteratively remove components which have faster growth from each successive vector. The removal of these components enables the computation of the exponential growth rates of vectors spanning the tangent space $T_{\mathbf{\Gamma}}X$. To avoid loss of orthogonality, and hence ensure computation of all exponents, QR factorization is performed periodically[3].

Consider again Equations (1.2.10) and (1.2.11). Here the consideration of degenerate singular values is not necessary, as will become clear shortly. Let W be the k-dimensional subspace spanned by the k largest singular values of A_n, $\sigma_{n,1}, \cdots, \sigma_{n,k}$. The volume of W grows over time, n, with rate $\phi_*^k(\mathbf{\Gamma})$, where

$$\phi_*^k(\mathbf{\Gamma}) := \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{k} \log \sigma_{n,i}(\mathbf{\Gamma}), \tag{2.2.7}$$

where $\sigma_{n,1}(\mathbf{\Gamma}) \geq \cdots \geq \sigma_{n,k}(\mathbf{\Gamma})$. Through application of Theorem 1, Equation (2.2.7) can be written in terms of the Lyapunov exponents,

$$\phi_*^k = \sum_{i=1}^{k} L_i, \tag{2.2.8}$$

where the dependence of L_i and ϕ_*^k on the initial orbit point $\mathbf{\Gamma}$ is dropped due to $\mathbf{M}$ being ergodic.

From the previous subsection, it is clear that a QR factorization of A_n computes an orthonormal matrix Q that has the same column span as A_n and an upper-triangular matrix R that contains the information of how much the column spans of Q need to be scaled to preserve the volume defined by the corresponding columns of A_n. The time-averaged sum of factors from successive applications of Modified Gram-Schmidt gives the following approximation to the Lyapunov exponents [98]:

$$L_i = \lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \log(r_{i,i}(t)) \quad 1 \leq i \leq N, \tag{2.2.9}$$

[3]The appropriate time between successive applications of QR depends upon the system being investigated. In practice, QR should be applied frequently enough to ensure appropriate scaling of the matrix. Otherwise, Modified Gram-Schmidt or any other QR process may fail to produce an orthonormal Q [93, 98].

where the time unit t is normalized by the length of orbit iterations by **M** between Modified Gram-Schmidt applications, and $r_{i,i}(t)$ represents the (i, i) element of the R matrix from the QR factorization step at time t. The algorithmic implementation is shown in Algorithm 3.

Algorithm 3 The Algorithm of Benettin *et. al* ($\{L_i\}$)

```
time = 0
for $i = 1 : N$ do
    $\text{cum}_i = 0$
end for
while time < endtime do
    Coevolve orbit ($N$ equations) and the linearized orbits ($N * N$ equations)
    Perform Modified Gram-Schmidt (see Algorithm 2, the outputs are $\{\mathbf{q}_i(time)\}$ and $\{r_{i,j}(\text{time})\}$)
    for $i = 1 : N$ do
        $\text{cum}_i += \log(r_{i,i}(\text{time}))$
        $L_i = \frac{\text{cum}_i}{\text{time}}$
    end for
    time += timestep
end while
return $\{\mathbf{q}_i(time)\}, \{L_i\}$
```

2.2.3 Worked Examples

A few simple examples are considered next to illustrate the application of the methods of the previous section. It is hoped that by presenting these methods now, their application to more complex, larger-dimensional systems will be more easily understood.

Lorenz System Perhaps the most natural choice of a chaotic low-dimensional continuous-time dynamical system is the Lorenz system [40, 60], described by

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{2.2.10}$$

The Lorenz system is of particular interest, since the structure of *attracting* sets varies for different choices of ρ. Following Section 1.1.3, for some set $U \subset \mathbb{R}^3$ let $\mathbf{\Phi}_U^t(U)$ be the time-t flow map for all $\mathbf{\Gamma} \in U$. A closed invariant set, $A \subset \mathbb{R}^3$, is called an *attracting set* if there is some neighborhood U of A such that for all $t > 0$, $\mathbf{\Phi}^t_U(U) \subset U$ and $\cap_{t>0}\mathbf{\Phi}^t_U(U) = A$ [40]. Some different attracting sets for the Lorenz system will be presented later.

Following again from Section 1.1.3 in Chapter 1, the state vector is $\mathbf{\Gamma} = (x\ y\ z)^\top$ with vector field defined by

$$\mathbf{F}(\mathbf{\Gamma}) := \begin{pmatrix} \sigma(y-x) \\ x(\rho-z)-y \\ xy-\beta z \end{pmatrix} \tag{2.2.11}$$

The linearized flow has state vector $\delta\mathbf{\Gamma} := (\delta x\ \delta y\ \delta z)^\top$ that evolves according to

$$\delta\dot{\mathbf{\Gamma}} = J_{\mathbf{F}}(\mathbf{\Gamma}(t))\delta\mathbf{\Gamma}, \tag{2.2.12}$$

where the Jacobian of $\mathbf{F}$ evaluated at $\mathbf{\Gamma}(t)$, $J_{\mathbf{F}}(\mathbf{\Gamma}(t))$, is 2.2.13

$$\begin{aligned} J_{\mathbf{F}}(\mathbf{\Gamma}(t)) :=& \left.\frac{\partial \mathbf{F}}{\partial \mathbf{\Gamma}}\right|_{\mathbf{\Gamma}=\mathbf{\Gamma}(t)} \\ =& \begin{pmatrix} -\sigma & \sigma & 0 \\ \rho - z(t) & -1 & -x(t) \\ y(t) & x(t) & -\beta \end{pmatrix}. \end{aligned} \tag{2.2.13}$$

To compute the Lyapunov exponents requires the simultaneous integration of 3 nonlinear equations from Equation (2.2.11) plus 3×3 linear equations from Equations (2.2.12) and (2.2.13) (3 linear equations are required for each of the 3 exponents). Typically, the initial condition for the nonlinear equations, $\mathbf{\Gamma}(0)$, is chosen from the *basin of attraction* for the attracting set A [71] and the initial conditions, $\{\delta^{(1)}\mathbf{\Gamma}(0), \delta^{(2)}\mathbf{\Gamma}(0), \delta^{(3)}\mathbf{\Gamma}(0)\}$ for each of the 3 sets of linear equations is chosen so that $\langle\delta^{(1)}\mathbf{\Gamma}(0), \delta^{(2)}\mathbf{\Gamma}(0), \delta^{(3)}\mathbf{\Gamma}(0)\rangle = \mathbb{R}^3$. The simplest such choice for $\{\delta^{(1)}\mathbf{\Gamma}(0), \delta^{(2)}\mathbf{\Gamma}(0), \delta^{(3)}\mathbf{\Gamma}(0)\}$ is to take the initial conditions as elements of the standard basis of $\mathbb{R}^3$ [98],

$$\begin{aligned} \delta^{(1)}\mathbf{\Gamma}(0) =& \ (1\ 0\ 0)^\top, \\ \delta^{(2)}\mathbf{\Gamma}(0) =& \ (0\ 1\ 0)^\top, \\ \delta^{(3)}\mathbf{\Gamma}(0) =& \ (0\ 0\ 1)^\top. \end{aligned} \tag{2.2.14}$$

Froyland and Ackens [34] computed the spectrum of Lyapunov exponents for $\sigma = 10, \beta = 8/3$ and $0.1 \leq \rho \leq 520$ demonstrating the different dynamical regimes of the

Lorenz system (see Figure 2.1). When the attracting set A is not a fixed point, one Lyapunov exponent will be equal to zero due to perturbations tangent to the flow [98]; such perturbation vectors cannot grow exponentially [35]. Figure 2.2 shows the dynamical behavior of the Lorenz system for different ρ.

It is important to discuss here the connection between the Lyapunov exponents and the full nonlinear dynamics of the system. As was demonstrated in Figures 2.1 and 2.2, the Lyapunov exponents allow for a quantitative understanding of the geometric structure of attracting sets for certain classes of dynamical systems. The Lyapunov exponents are of fundamental importance for quantifying chaos, as they are used to compute other measures of instability. These other measures include the *Kolmogorov-Sinai entropy* [18, 35], which is bounded above by the sum of all positive Lyapunov exponents by Pesin's theorem [73], and the *Kaplan-Yorke dimension* [51], also called the *Lyapunov Dimension* [71], which relates the fractal dimension of an attracting set to the positive Lyapunov exponents.

Bouncing Ball-Platform System- Discrete Version A nice example of a two-dimensional discrete-time dynamical system is based upon a reduction of the Bouncing Ball-Platform system from Section 1.1.5. Under certain assumptions [40], the ball-platform system can be reduced to a two-dimensional map[4],

$$\begin{aligned} \phi_{j+1} &= \phi_j + v_j, \\ v_{j+1} &= \alpha v_j - \mu \cos(\phi_j + v_j), \\ \mu &:= \frac{2\omega_\mathrm{p}^2(1+\alpha)\beta}{g}. \end{aligned} \tag{2.2.15}$$

The variable ϕ_j at each j, defined as $\phi_j := \omega_\mathrm{p} t_j$, represents the phase of the platform at each impact time, t_j, while v_j is the normalized velocity of the ball after impact

$$v_j = \frac{2\omega_\mathrm{p} v_\mathrm{b}^+(t_j)}{g},$$

[4]This map has been demonstrated to result in physically-irrelevant results [28] (i.e. negative velocity of the ball after impact) and is presented purely to illustrate the process for computing Lyapunov exponents of a discrete map.

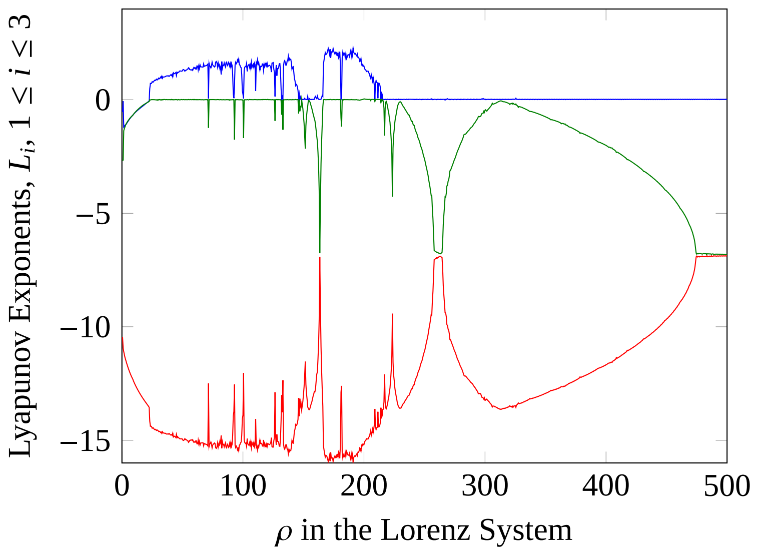


FIGURE 2.1. Lyapunov spectra (λ_1,λ_2,λ_3) for the Lorenz system as a function of the parameter ρ. The parameters σ and β in Equation (2.2.10) are held constant at 10 and 8/3, respectively, while ρ is varied between 0.5 and 500 by increments of $\Delta\rho = 0.5$. The values of the Lyapunov exponents show that the Lorenz system, for certain choices of ρ, can have *strange attractors* (one positive exponent), *stable limit cycles* (two negative exponents) or *stable fixed points* (three negative exponents). As originally observed by Froyland and Alfsen [34], the dips in spectra are due to the importance of both integrator error tolerance and the fine details of the orbit (i.e. accumulation points of the period-doubling bifurcation) at these ρ.

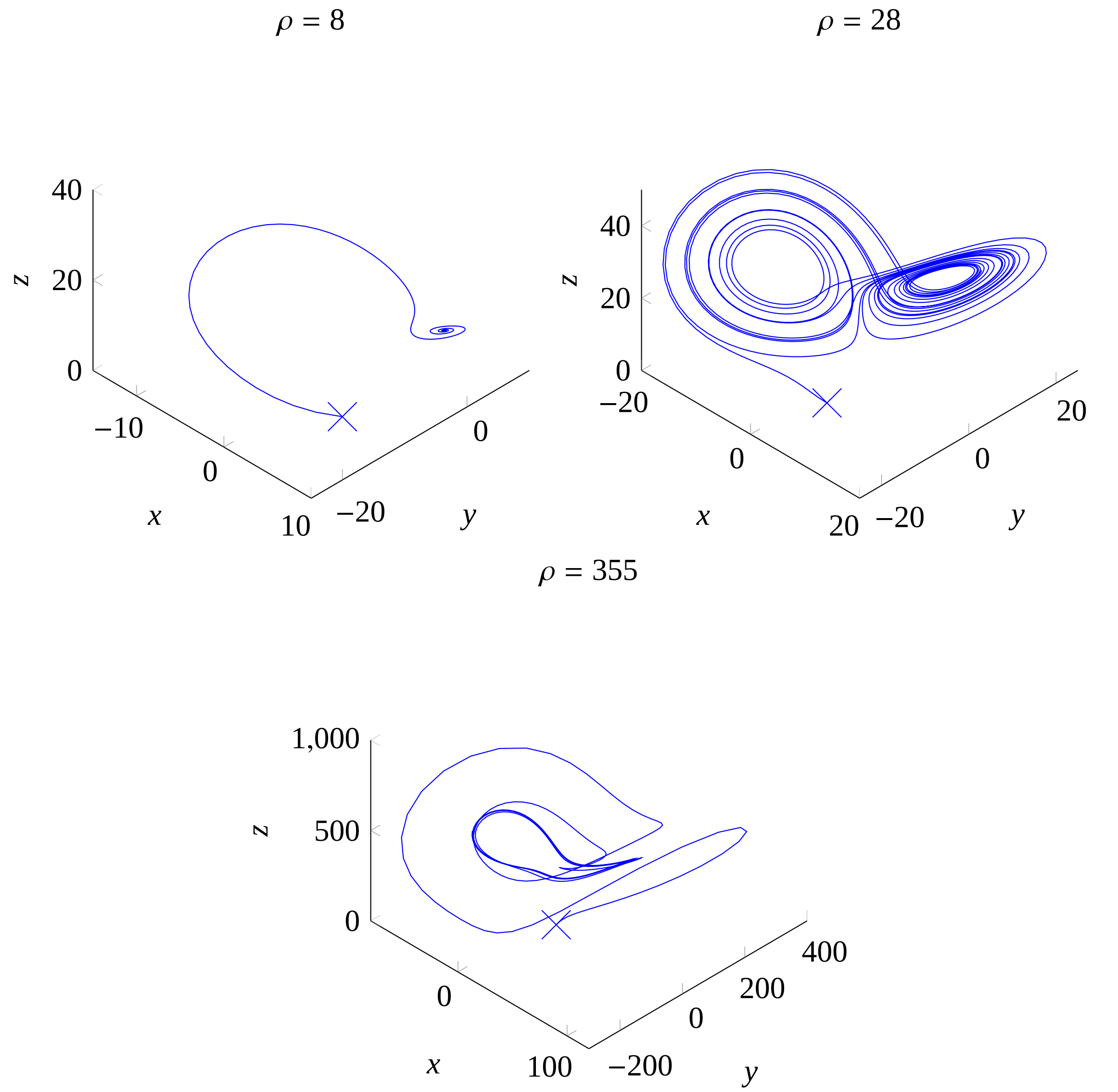


FIGURE 2.2. The parameters σ and β in Equation (2.2.10) are held constant at 10 and 8/3, respectively, while the parameter ρ is chosen to demonstrate qualitatively different attracting sets. The initial condition, $\mathbf{\Gamma}(0) = (10 \ \ -20 \ \ 14)^\top$, is marked with an $\times$. Top left: $\rho = 8$ results in a *stable fixed point*. Top right: $\rho = 28$ results in a *strange attractor*. Bottom center: $\rho = 355$ results in a *stable limit cycle*.

where v_{b}^{+} is the velocity after impact in Equation (1.1.13). All other terms are the same as was defined in Section 1.1.5. Equation (2.2.15) can be put into the desired form from Section 1.1.2

$$\begin{aligned}\mathbf{\Gamma}_n &:= \begin{pmatrix}\phi_n\\ v_n\end{pmatrix}\\ \mathbf{\Gamma}_{n+1} &= \mathbf{M}(\mathbf{\Gamma}_n) := \begin{pmatrix}\phi_j + v_j\\ \alpha v_j - \mu\cos(\phi_j + v_j)\end{pmatrix}.\end{aligned} \tag{2.2.16}$$

Computing the Lyapunov exponents requires evolving perturbations to the reference orbit defined in Equation (2.2.16) via the relation in Equation (1.2.4),

$$\begin{aligned}\delta\mathbf{\Gamma}_n &:= \begin{pmatrix}\delta\phi_n\\ \delta v_n\end{pmatrix}\\ \delta\mathbf{\Gamma}_{n+1} &= J_{\mathbf{M}}(\mathbf{\Gamma}_n)\delta\mathbf{\Gamma}_n,\end{aligned} \tag{2.2.17}$$

where the Jacobian of the map $\mathbf{M}$ evaluated at the current reference point, $J_{\mathbf{M}}(\mathbf{\Gamma}_n)$, is

$$\begin{aligned}J_{\mathbf{M}}(\mathbf{\Gamma}_n) &:= \left.\frac{\partial\mathbf{M}}{\partial\mathbf{\Gamma}}\right|_{\mathbf{\Gamma}=\mathbf{\Gamma}_n}\\ &= \begin{pmatrix}1 & 1\\ \mu\sin(\phi_j + v_j) & \alpha + \mu\sin(\phi_j + v_j)\end{pmatrix}\end{aligned} \tag{2.2.18}$$

Computing the Lyapunov exponents for the map in Equation (2.2.16) requires the simultaneous iteration of the 2 nonlinear equations in Equation (2.2.16) plus 4 linear equations from Equation (2.2.17). Following from Equation (2.2.14), choose the initial conditions for the linear equations from the standard basis on $\mathbb{R}^2$. When $\alpha < 1$, the system is dissipative and, as a consequence, the sum of the Lyapunov exponents is negative [40].

Figure 2.3 demonstrates again that the Lyapunov exponents provide a means for understanding qualitatively different dynamics in a system with varying parameters. The change in exponents for increasing μ corresponds directly to period-doubling events. When a positive exponent is present, no regularity (periodic behavior) is observed.

2.3 Lyapunov Vectors

Theorem 1 gives the assumptions necessary for the existence of the Lyapunov exponents, the Oseledec filtration (for not-necessarily invertible transformations) and the Oseledec

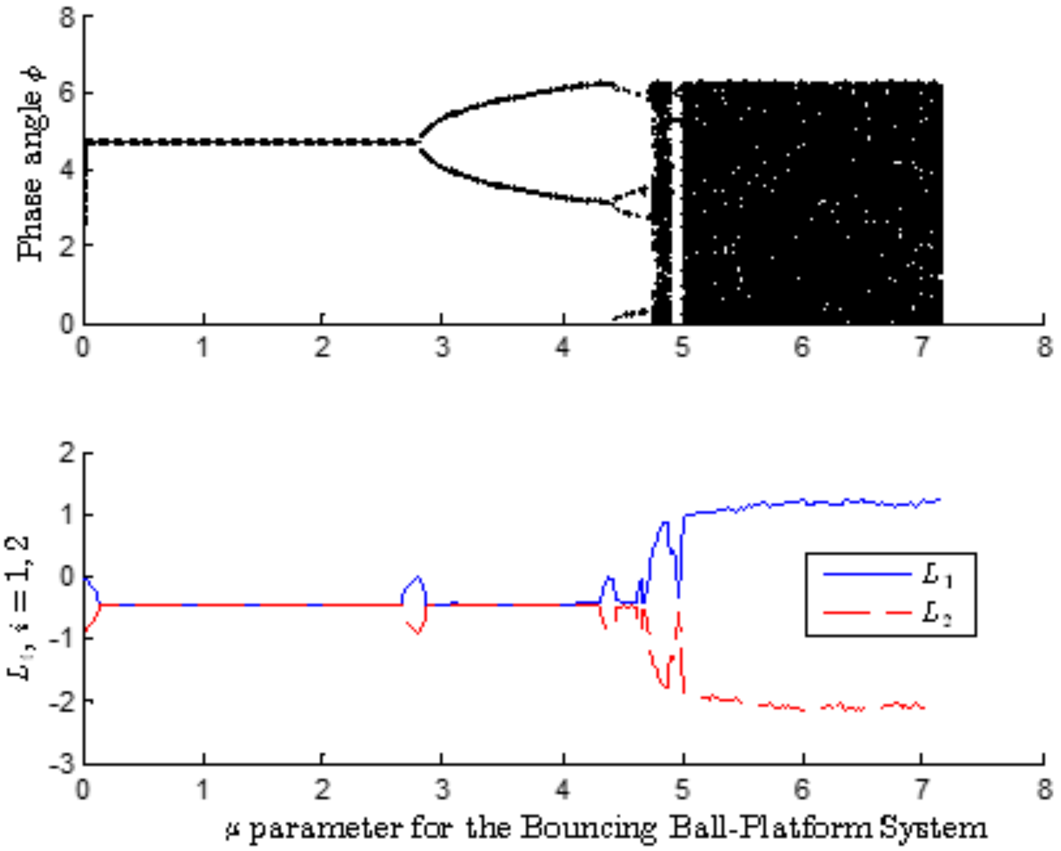


FIGURE 2.3. Top: The phase ϕ at impact points are shown as a function of μ (the table amplitude, β, is set to unity and the coefficient of restitution $\alpha = 0.4$). The last 200 iterations are shown. Bottom: The Lyapunov exponents provide a quantitative measure of instability of trajectories seen in the bifurcation diagram.

splitting (for invertible transformations). For the systems that are considered subsequently in this work, invertibility is assumed unless noted otherwise.

Let X be an N-dimensional compact Riemannian manifold, and $\mathbf{M} : X \to X$ be a diffeomorphic ergodic transformation with Jacobian matrix $A := J_{\mathbf{M}}$. Let A_n again be as defined in Equation (1.2.4). Recall that for invertible transformations, the tangent space can be split into a direct sum of subspaces as in Equation (1.2.12)

$$\mathbb{R}^m = E^{(1)} \oplus E^{(2)} \oplus \cdots \oplus E^{(r)},$$

where $r \leq N$ is the number of non-degenerate Lyapunov exponents. Benettin's algorithm is used to compute these exponents from the upper-triangular R matrix from the Modified Gram-Schmidt procedure. The question then becomes: How do the columns of the Q matrix computed relate to the splitting from Equation (1.2.12)? The columns of Q are called *orthogonal Lyapunov vectors*, *Gram-Schmidt Lyapunov vectors* [37] or *Lyapunov modes* [21, 30, 31, 47], and they are discussed in the following section.

2.3.1 Orthogonal Lyapunov Vectors

As previously mentioned in 7, the elements of the splitting $E^{(i)}$ are not, in general, pairwise orthogonal with each other. However, the column vectors of Q computed by Modified Gram-Schmidt *are* pairwise orthogonal. Accounting for degeneracies in the spectrum, the orthogonal Lyapunov vectors associated with the jth non-degenerate Lyapunov exponent are an orthogonal spanning set of the subspace[5]

$$M^{(j)} := (F^{(j-1)})^{\perp} \cap F^{(j)}, \tag{2.3.1}$$

where $1 \leq j \leq r$ and the subspaces $F^{(j)}$ are defined in terms of the elements of the Oseledec splitting,

$$F^{(j)} := E^{(1)} \oplus E^{(2)} \oplus \cdots \oplus E^{(j)}. \tag{2.3.2}$$

[5]The subsequent discussion assumes that the dynamics have been evolved far enough in time for the orthogonal Lyapunov vectors to converge to the so-called *asymptotic* basis [26, 37].

The spaces $M^{(j)}$ and $E^{(j)}$ are similar [21]. For each $1 \leq j \leq r$, $\dim M^{(j)} = \dim E^{(j)} = m(L^{(j)})$, where $m(L^{(j)})$ is the degeneracy of the j^{th} non-degenerate exponent. Combining Equations (2.3.1) and (2.3.2), it is true that

$$\begin{aligned} F^{(1)} =& M^{(1)} \\ F^{(2)} :=& E^{(1)} \oplus E^{(2)} \\ =& M^{(1)} \oplus \left[(E^{(1)})^{\perp} \cap (E^{(1)} \oplus E^{(2)}) \right] \\ =& M^{(1)} \oplus M^{(2)} \\ & \vdots \\ F^{(j)} :=& E^{(1)} \oplus \cdots \oplus E^{(j)} \\ =& M^{(1)} \oplus M^{(2)} \oplus \cdots \oplus \left[(E^{(1)} \oplus \cdots \oplus E^{(j-1)})^{\perp} \cap (E^{(1)} \oplus \cdots \oplus E^{(j)}) \right] \\ =& M^{(1)} \oplus M^{(2)} \oplus \cdots \oplus M^{(j)}. \end{aligned} \tag{2.3.3}$$

Therefore, it can be concluded that

$$E^{(1)} \oplus E^{(2)} \oplus \cdots \oplus E^{(j)} = M^{(1)} \oplus M^{(2)} \oplus \cdots \oplus M^{(j)}, \tag{2.3.4}$$

for $1 \leq j \leq r$.

Following the arguments of the proof of the Multiplicative Ergodic Theorem (see Equation (A.1.10)), the following limit exists almost surely

$$\Lambda_{\pm} := \lim_{n\to\infty} (A_{\pm n}(\mathbf{\Gamma})^{\top} A_{\pm n}(\mathbf{\Gamma}))^{\frac{1}{2|n|}}, \tag{2.3.5}$$

where the matrices $A_{\pm n}(\mathbf{\Gamma})$ are defined in terms of the Jacobian matrices of $\mathbf{M}$ and $\mathbf{M}^{-1}$,

$$\begin{aligned} A_{+n}(\mathbf{\Gamma}) &:= J_{\mathbf{M}}(\mathbf{M}^{n}(\mathbf{\Gamma})) \cdots J_{\mathbf{M}}(\mathbf{M}(\mathbf{\Gamma}))J_{\mathbf{M}}(\mathbf{\Gamma}), \\ A_{-n}(\mathbf{\Gamma}) &:= J_{\mathbf{M}^{-1}}(\mathbf{M}^{-n}(\mathbf{\Gamma})) \cdots J_{\mathbf{M}^{-1}}(\mathbf{M}^{-1}(\mathbf{\Gamma}))J_{\mathbf{M}^{-1}}(\mathbf{\Gamma}). \end{aligned} \tag{2.3.6}$$

The eigenvalues of Λ_{+} are defined in terms of the Lyapunov exponents

$$\exp(L^{(1)}) > \exp(L^{(2)}) > \cdots > \exp(L^{(r)}).$$

The eigenvalues of Λ_{-} follow similarly

$$\exp(-L^{(1)}) < \exp(-L^{(2)}) < \cdots < \exp(-L^{(r)}).$$

Because both Λ_+ and Λ_- are symmetric positive definite, the tangent space can be split into a direct sum of the eigenspaces denoted $(U_\pm)$ of each matrix [21, 85]

$$\mathbb{R}^m = U_\pm^{(1)} \oplus U_\pm^{(2)} \oplus \cdots U_\pm^{(r)}. \tag{2.3.7}$$

The $U_\pm^{(j)}$ are pairwise orthogonal but they are not covariant, in general. Again, following the arguments from the proof of the Multiplicative Ergodic Theorem (specifically Equation (A.1.11)), the following subspaces are covariant:

$$\begin{aligned} V_+^{(j)} &:= U_+^{(j)} \oplus \cdots \oplus U_+^{(r)} \\ V_-^{(j)} &:= U_-^{(1)} \oplus \cdots \oplus U_-^{(j)}. \end{aligned} \tag{2.3.8}$$

Combining this information with Equation (1.2.18) yields

$$\begin{aligned} E^{(j)} =& V_+^{(j)} \cap V_-^{(j)} \\ :=& (U_+^{(j)} \oplus \cdots \oplus U_+^{(r)}) \cap (U_-^{(1)} \oplus \cdots \oplus U_-^{(j)}). \end{aligned} \tag{2.3.9}$$

The subspace $V_+^{(j)}$ represents the $m(L^{(j)}) + \cdots + m(L^{(r)})$ most *stable* directions of Λ_+ , while the subspace $V_-^{(j)}$ represents the $m(L^{(1)}) + \cdots + m(L^{(j)})$ most *unstable* directions of Λ_-[6]. From Equation (2.3.9), it follows that

$$U_-^{(1)} \oplus U_-^{(2)} \oplus \cdots \oplus U_-^{(j)} = E^{(1)} \oplus E^{(2)} \oplus \cdots \oplus E^{(j)} = F^{(j)}. \tag{2.3.10}$$

Recalling the definition of $M^{(j)}$ in Equation (2.3.1) and the fact that the subspaces $U_-^{(j)}$ are pairwise orthogonal, it follows that $M^{(j)} = U_-^{(j)}$.

It may seem somewhat counterintuitive that the forward-time dynamics results in the computation of a basis for the tangent space in terms of the eigenspaces of the backward-time Λ_- matrix. However, it should be clear that both bases are solely determined by information from the past history of an orbit [37]. The measured orthogonal Lyapunov vectors are of some utility when special properties of the system under investigation are present, such as for systems of hard disks undergoing elastic collisions [16, 21, 30, 31].

[6]Equivalently, the most stable directions of the forward-time dynamics. These directions coincide with the $m(L^{(r-j+1)}) + \cdots + m(L^{(r)})$ most stable directions of Λ_+.

These systems will discussed in greater detail in Chapter 4. However, in a more general setting, orthogonal Lyapunov vectors are of a limited utility in understanding the local structure of stable and unstable manifolds [36, 37].

2.3.2 Covariant Lyapunov Vectors

Orthogonal Lyapunov vectors do not, in general, remain covariant under transformation by the mapping $\mathbf{M}$. This is because the Jacobian matrix $J_{\mathbf{M}}$ is not necessarily orthogonal while the Modified Gram-Schmidt process forces orthogonality of the Lyapunov vectors. For the computation of the Lyapunov exponents using Benettin's algorithm, the orthogonality of the orthogonal Lyapunov vectors is not a concern, as the volumetric growth of subspaces spanned by the column vectors is considered [5, 37, 98], not the *directions* of growing or contracting perturbations. Understanding the directions in tangent space corresponding to growing or shrinking perturbations provides an effective means of classifying dynamical systems; particularly their *hyperbolic* properties. This will be discussed further in Chapters 4 and 5. To compute these directions requires the construction of covariant Lyapunov vectors. Covariant Lyapunov vectors provide information about the local structure of *stable* and *unstable manifolds* of steady states[7] along orbits. The stable manifold of a steady state is the set of all points that approach the steady state in forward-time [71]. Similarly, the unstable manifold is the set of points that approach the steady state in backward-time. Several techniques have recently been developed for computing these vectors: a singular value decomposition (SVD) approach [32, 33], the dichotomy projector method [32, 48, 49], Wolfe's method [32, 99] and Ginelli's method [32, 36, 37]. Following the analyses of Froyland *et. al* [32], and the relative ease of implementation, Ginelli's method is chosen.

[7]A steady state is a fixed point or a closed curve traced by a periodic orbit [71].

Ginelli's Method Consider the forward evolution of an orthonormal basis $\mathbf{G}_n$ output from Benettin's algorithm from some large time n[8] forward k time steps

$$\bar{\mathbf{G}}_{n+k} = J^k_{\mathbf{M}}(\mathbf{\Gamma}_n)\mathbf{G}_n, \tag{2.3.11}$$

where $J^k_{\mathbf{M}}(\mathbf{\Gamma}_n) := J_{\mathbf{M}}(\mathbf{\Gamma}_{n+k-1})J_{\mathbf{M}}(\mathbf{\Gamma}_{n+k-2})\cdots J_{\mathbf{M}}(\mathbf{\Gamma}_n)$. In general, the matrix $\bar{\mathbf{G}}_{n+k}$ is not orthogonal, and so it must be renormalized

$$\bar{\mathbf{G}}_{n+k} = \mathbf{G}_{n+k}\mathbf{R}_{n+k}. \tag{2.3.12}$$

Ginelli's method relies on the *asymptotic* property of orthogonal Lyapunov vectors of an ergodic transformation with an invariant measure[9]: namely that, for sufficiently long time, the orthonormal basis of the tangent space obtained through periodic orthonormalization of the evolved tangent space dynamics converges exponentially to the eigenvectors of the backward Oseledec matrix, Λ_-, from Equation (2.3.5) [26, 37]. This result should come as no surprise. In the previous section, it was shown that the measured subspaces $M^{(j)}$ from the Gram-Schmidt process coincide with the subspaces spanned by eigenvectors of Λ_-.

Starting from Equation (2.3.12), covariant Lyapunov vectors can be constructed. After evolving an orbit for long enough time n[10], the system is evolved forward an additional ωk time steps. The parameters ω and k are the orthonormalization frequency and number of orthonormalizations to perform, respectively. After orthonormalization at the final time step $n + k\omega$, it is true that

$$\bar{\mathbf{G}}_{n+\omega k} = \mathbf{G}_{n+\omega k}\mathbf{R}_{n+\omega k} = J^{\omega}_{\mathbf{M}}(\mathbf{\Gamma}_{n+\omega(k-1)})\mathbf{G}_{n+\omega(k-1)}. \tag{2.3.13}$$

Define the $N \times N$ matrix

$$\mathbf{W}_k := \mathbf{G}_{n+\omega k}\mathbf{C}_k, \tag{2.3.14}$$

[8]The consideration of n large is important for subsequent arguments. More on this will come shortly.

[9]Ershov and Potapov use the term *stationary* to describe the long-time limit of the orthogonal Lyapunov vectors. This phrase is misleading, as the vectors do continue to evolve in time. Therefore, the convention referring to the long-time limit as *asymptotic* from Ginelli *et al.* is adopted [37].

[10]The concept of "long enough" is system-dependent. It is desirable to pick a time sufficiently long for initial transients due to arbitrarily chosen initial conditions of the linearized equations to settle.

where $\mathbf{C}_k$ is an initially arbitrary $N \times N$ upper-triangular matrix that contains the expansion coefficients of the covariant Lyapunov vectors in the orthogonal Lyapunov vector basis, $\mathbf{G}_{n+\omega k}$. The covariant Lyapunov vectors have normalized length, and therefore the columns of $\mathbf{C}_k$ each must have norm 1,

$$\|[\mathbf{C}_k]_j\|^2 := \sum_{i=1}^{N} c_{i,j}^2 = 1, \tag{2.3.15}$$

where $1 \leq j \leq N$ and $c_{i,j}$ is the $(i,j)^{\text{th}}$-element of $\mathbf{C}_k$. Now, the backward evolution rule for $\mathbf{C}$ is constructed to ensure the matrix $\mathbf{W}_j$ is covariant with the dynamics:

$$\mathbf{C}_{j-1} = \mathbf{R}_{n+\omega j}^{-1}\mathbf{C}_j, \tag{2.3.16}$$

where $1 \leq j \leq k$ and $\mathbf{R}_{n+\omega j}^{-1}$ is the inverse of the $\mathbf{R}$ matrix obtained from the orthonormalization of $\bar{\mathbf{G}}_{n+\omega j}$.

It is straightforward to verify that, given Equation (2.3.16), $\mathbf{W}_j$ is covariant with the dynamics for each $1 \leq j \leq k$. Combining Equations (2.3.14) and (2.3.16) yields

$$\begin{aligned} \mathbf{W}_j :=& \mathbf{G}_{n+\omega j}\mathbf{C}_j \\ =& \bar{\mathbf{G}}_{n+\omega j}\mathbf{C}_{j-1} \\ =& J_{\mathbf{M}}^{\omega}(\mathbf{\Gamma}_{n+\omega(j-1)})\mathbf{G}_{n+\omega(j-1)}\mathbf{C}_{j-1} \\ :=& J_{\mathbf{M}}^{\omega}(\mathbf{\Gamma}_{n+\omega(j-1)})\mathbf{W}_{j-1}. \end{aligned} \tag{2.3.17}$$

There is one final technical point. The $\mathbf{C}$ matrices indicate the growth of each covariant Lyapunov vector. Since these vectors grow (or shrink) exponentially, each column vector of $\mathbf{C}_j$ is scaled by its norm at each $1 \leq j \leq k$[11]. The algorithmic implementation of Ginelli's method is summarized in Algorithm 4.

[11]Benettin's algorithm is typically used to compute the Lyapunov exponents. The same information can be obtained from Ginelli's algorithm; where a "-" in the exponent calculation is used to account for time-reversal (see Algorithm 4).

Algorithm 4 Ginelli's Method for Computing Covariant Lyapunov Vectors ($\{\mathbf{W}_i\}, \{L_i\}$)

```
Run Benettin's algorithm (see Algorithm 3) until time $n$; when transients in orthogonal
Lyapunov vectors have settled.
Continue running Benettin's algorithm until time $n + \omega k$. Store the $\mathbf{R}_j, \mathbf{G}_j$ matrices at
each orthonormalization step $\omega j$, $1 \leq j \leq k$.
Initialize $\mathbf{W}_k = \mathbf{G}_k$ and $\mathbf{C}_k = \mathbf{I}_{N\times N}$, where $\mathbf{I}_{N\times N}$ is the $N \times N$ identity matrix.
for $i = 1 : N$ do
    $\mathrm{cum}_i = 0$.
end for
counter = 0.
for $i = k : -1 : 2$ do
    Compute $\mathbf{C}_{i-1}$ (see Equation (2.3.16)).
    for $j = 1 : N$ do
        $\mathrm{nrm} = \|[\mathbf{C}_k]_j\|$.
        $\mathrm{cum}_j + = \log \mathrm{nrm}$.
        $[\mathbf{C}_k]_j / = \mathrm{nrm}$.
    end for
    Compute $\mathbf{W}_{i-1}$ (see Equation (2.3.14)).
    counter+ = 1
    for $j = 1 : N$ do
        $-L_j = \mathrm{cum}_j / (\omega * \mathrm{counter})$.
    end for
end for
return $\{\mathbf{W}_i\}, \{L_i\}$
```

Covariant Lyapunov Vectors and the Hénon Map A simple illustrative example for computing covariant Lyapunov vectors is the Hénon map,

$$\begin{aligned}\mathbf{\Gamma}_{n+1} &= \mathbf{M}(\mathbf{\Gamma}_n)\\ &:= \begin{pmatrix} -ax_n^2 + y_n + 1 \\ bx_n \end{pmatrix},\end{aligned} \tag{2.3.18}$$

where $\mathbf{\Gamma}_n := (x_n\ y_n)^\top$ and $a, b \in \mathbb{R}$. The map in Equation (2.3.18) has two fixed points

$$\mathbf{p} = \begin{pmatrix} \frac{b-1+\sqrt{(b-1)^2+4a}}{2a} \\ \frac{b(b-1+\sqrt{(b-1)^2+4a})}{2a} \end{pmatrix}$$
$$\mathbf{q} = \begin{pmatrix} \frac{b-1-\sqrt{(b-1)^2+4a}}{2a} \\ \frac{b(b-1-\sqrt{(b-1)^2+4a})}{2a} \end{pmatrix}.$$

Let $(a, b) = (1.4, 0.3)$. The eigenvalues $\lambda_{1,2}$ ($|\lambda_1| > |\lambda_2|$) of the Jacobian $J_{\mathbf{M}}$ at each fixed point, are calculated by the characteristic equation

$$\lambda_{1,2}(\lambda_{1,2} + 2ax_n) - b = 0, \tag{2.3.19}$$

where $x_n = \frac{b-1\pm\sqrt{(b-1)^2+4a}}{2a}$. The eigenvalues at each fixed point are

$$\begin{aligned}\lambda_{1,2}(\mathbf{p}) &= \{-1.924, 0.156\},\\ \lambda_{1,2}(\mathbf{q}) &= \{3.260, -0.092\}.\end{aligned} \tag{2.3.20}$$

At each fixed point, one eigenvalue has absolute value greater than one and one has absolute value less than one, which means the fixed points are *hyperbolic* [40, 71, 97]. The eigenvector corresponding to λ_1 defines the dimension-1 unstable subspace at each fixed point. Similarly, the eigenvector corresponding to λ_2 defines the dimension-1 stable subspace. Points along the unstable subspace near each fixed point are repelled from it by action of **M** while points along the stable subspace are attracted to the fixed point. Figure 2.4 shows the structure of the *Hénon attractor* for the map **M** [12, 45].

Ginelli's algorithm 4 can be used to construct the covariant Lyapunov vectors for the map (2.3.18). For the technical details of the implementation, see the source code listing in Appendix B.3.3. Figure 2.5 shows the structure of the stable and unstable manifolds of

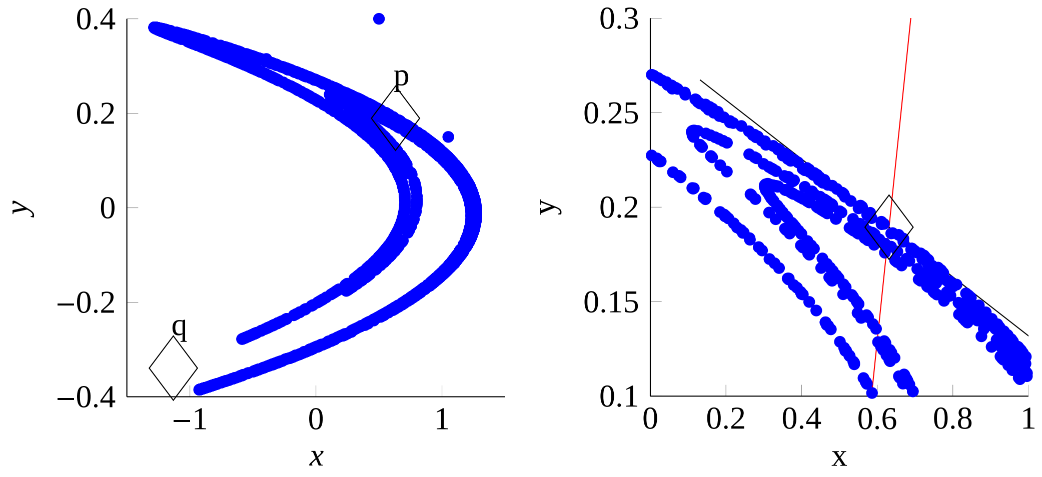


FIGURE 2.4. Plot of the *Hénon attractor* for $a = 1.4, b = 0.3$. Left: The attractor shown with both fixed points (indicated by diamonds), **p**, **q**. The fixed point **p** is on the attractor, while **q** is not. Right: The local stable (red line) and unstable (black line) subspaces of **p**. The unstable subspace is tangent to the attractor at **p**.

p. The unstable manifold, $W^u(\mathbf{p})$, of **p** is well-approximated by the attractor [12]. Approximating the stable manifold, $W^s(\mathbf{p})$, requires mapping a set, U, of points along the stable subspace at **p** backward in time by the inverse $\mathbf{M}^{-1}$,

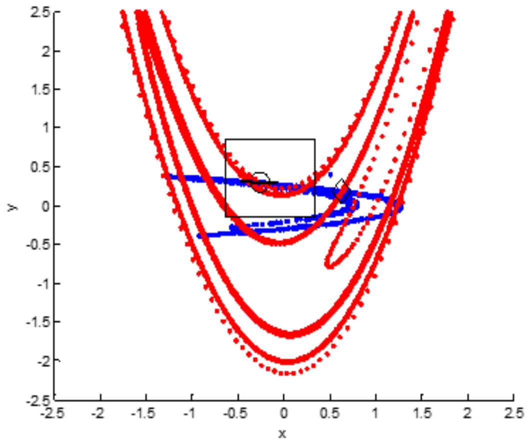
$$W^s(\mathbf{p}) = \bigcup_{n\leq 0} \mathbf{M}^n(U). \tag{2.3.21}$$

Figure 2.5 demonstrates that, for an arbitrary point along a typical[12] orbit for the map of Equation (2.3.18), Ginelli's method provides an accurate approximation of the Oseledec splitting at that point.

2.4 Further Reading

The process for computing Lyapunov exponents using volumetric growth of subspaces co-evolving with the dynamics has been used for many systems [5, 82, 98]: the Lorenz system [34, 98], the Rössler system [98] and the Duffing oscillator [104], to name only a few. In

[12]A "typical" orbit is one which starts in the basin of attraction of the attractor.



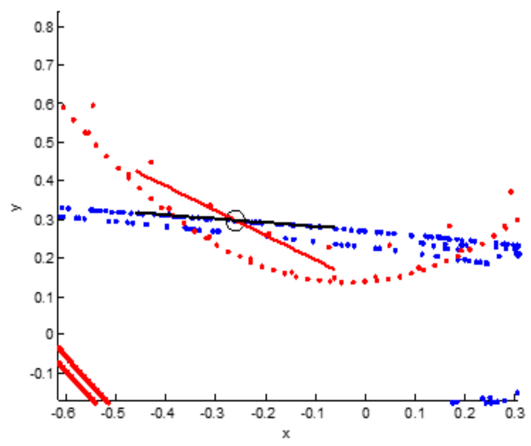


FIGURE 2.5. The structure of covariant Lyapunov vectors and invariant manifolds for the Hénon map. Left: The approximation of the unstable manifold, $W^u(\mathbf{p})$, is in blue while the approximation of the stable manifold, $W^s(\mathbf{p})$, is in red. An arbitrary point along the orbit is selected (indicated by **O**) for comparing the covariant Lyapunov vectors to the local structure of $W^s(\mathbf{p})$ and $W^u(\mathbf{p})$. Right: Zoomed-in region of the figure at left (indicated by the square region at left). The covariant Lyapunov vector associated with the negative Lyapunov exponent (in red) is tangent to $W^s(\mathbf{p})$ while the covariant Lyapunov vector associated with the positive exponent (in black) is tangent to $W^u(\mathbf{p})$.

all of these cases, the setup is very similar and follows the process used for the examples of Section 2.2.3.

The subject of Lyapunov vectors has become an active research area in the last decade [21, 31, 36, 37, 47]. Much of this research relies upon the theoretical results of [26]: that Lyapunov vectors converge exponentially to the basis of the backward Oseledec matrix Λ_-. For a further development of what is measured by the Benettin algorithm 3, see Eckmann *et. al* [21]. A complete introduction to Ginelli's method can be found in [37]. Nice applications of Lyapunov vectors (and comparisons thereof) exist for simple Hamiltonian systems [36], time-dependent fluid flow through a cylinder [32] and hard-disk systems [8, 69]. The references for hard-disk systems, along with several others, will be discussed further in the coming chapters.

Chapter 3
A Generalized Impulsive Collision Rule for Interacting Particles

Arguably, one of the most important models to understanding macroscopic properties (e.g. pressure and diffusion) of systems in statistical mechanics is the system of a large number of identical particles interacting through *elastic collisions*. When particles undergo an elastic collision, the normal component (i.e. the component perpendicular to the collision axis) of each particle's momentum is left unchanged after collision, while an impulsive separating force tangential to the collision axis is applied (see Equation (3.1.14)). Models incorporating elastic collisions have been studied extensively from both the physical (see [7, 18, 52, 95]) and dynamical systems (see [16, 17, 31, 21]) perspectives. The systems that will be considered here subsequently seek to answer two simple questions. Firstly, are there other collision rules, besides elastic collisions, that can be constructed that have the same conserved quantities as in the elastic case? If so, how do the dynamical properties (e.g. Lyapunov exponents and entropy) of such collision rules compare to those of the elastic collision rule? These collision rules will not, in general, leave the normal component of each particle's momentum unchanged, as in the elastic collision rule.

The answers to both questions from the preceding paragraph will be addressed in this chapter. Firstly, the full nonlinear dynamics of a novel generalized binary collision rule are developed in Section 3.1. To answer the second question requires the extension of a method for computing the Lyapunov exponents of dynamical systems with periods of free flight interrupted by collisions (recall Section 1.1.4). The method, attributed to Dellago *et. al* [16], has been used to study the dynamical properties of systems of identical particles interacting via elastic collisions. This method will be discussed in Section 3.3 and extended to the generalized collision rule.

Before discussing the dynamics, it is useful to consider what this new generalized collision rule represents physically. By enforcing conservation of momentum and energy at each collision, but removing the constraint that the normal component of each particle's momentum is left invariant, a degree-of-freedom in the construction of the collision rule is introduced which can be tuned to achieve a desired system response (e.g. reduction in quantitative measures of chaos).

3.1 Phase Space Dynamics

The dynamics for N particles of equal mass m and diameter σ moving on the 2-torus $\mathbb{T}^2 = \{(x \bmod L_x, y \bmod L_y)^\top | \, L_x, L_y \in \mathbb{R}\}$ is represented by the state $\mathbf{\Gamma}$,

$$\begin{aligned}\mathbf{\Gamma} &:= \begin{pmatrix} \mathbf{Q} \\ \mathbf{P} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{q}^1 \\ \vdots \\ \mathbf{q}^N \\ \mathbf{p}^1 \\ \vdots \\ \mathbf{p}^N \end{pmatrix},\end{aligned}$$

where $\mathbf{q}^j := (q^{1j} \; q^{2j})^\top$ and $\mathbf{p}^j := (p^{1j} \; p^{2j})^\top$ are the position and momentum of particle j, respectively. The vector $\mathbf{\Gamma}$ is $4N$-dimensional. Particles move continuously in the absence of any external force until two particles collide. At this time, the particles undergoing collision, and only these particles, experience an instantaneous force that (i) takes a weighted average of the momenta of the colliding particles and (ii) is repulsive along the vector connecting the colliding particles' centers-of-mass. After the application of the collision transformation, the alternating process continues. This system can be thought of as an example of a non-smooth, or hybrid, dynamical system where jump events occur at successive collision times. In the language of hybrid dynamical systems, the collision map is known as the reset map [38]. The transformations that represent the free flight and collision dynamics will be discussed in detail in the subsequent sections.

3.1.1 Free Flight

During free flight, when no external field is applied, the dynamics are very simple. During this stage, the system evolves according to the relation

$$\begin{aligned} \frac{d\mathbf{\Gamma}(t)}{dt} &= \mathbf{F}(\mathbf{\Gamma}(t)) \\ &= \begin{pmatrix} \frac{\mathbf{P}}{m} \\ \mathbf{0} \end{pmatrix} \end{aligned} \tag{3.1.1}$$

where $\mathbf{0} = (0\ 0\ \cdots\ 0)^\top$ is the $2N$-dimensional zero vector and $\mathbf{P} := \mathbf{P}(t_0)$; that is the momentum $\mathbf{P}$ is constant during free flight.. The vector field $\mathbf{F}$ generates the flow map $\mathbf{\Phi}^{s-\tau}_{\mathbf{\Gamma}(\tau)}(\mathbf{\Gamma}(\tau))$, which takes $\mathbf{\Gamma}(\tau)$, the state at time τ to the state at some other time s,

$$\begin{aligned} \mathbf{\Gamma}(s) &= \mathbf{\Phi}^{s-\tau}_{\mathbf{\Gamma}(\tau)}(\mathbf{\Gamma}(\tau)) \\ &= \mathbf{\Gamma}(\tau) + \mathbf{F}(\mathbf{\Gamma}(\tau))(s-\tau) \\ &= \mathbf{\Gamma}(\tau) + \begin{pmatrix} \frac{\mathbf{P}}{m} \\ \mathbf{0} \end{pmatrix}(s-\tau). \end{aligned} \tag{3.1.2}$$

It should be noted that s need not be greater than τ for Equation (3.1.2) to be valid, as long as there are no collisions in between times s and τ. It is important to state here that, for the numerical investigations of the subsequent chapters, simple forward Euler integration is exact. There is no truncation error due to numerical integration during free flight.

3.1.2 Collision

Let integers j, k $(1 \leq j < k \leq N)$ be the indices of particles undergoing pairwise collision[1] and let the relative position and momentum before collision be denoted $\mathbf{q} := \mathbf{q}_i^k - \mathbf{q}_i^j$ and $\mathbf{p} := \mathbf{p}_i^k - \mathbf{p}_i^j$, respectively. The collision conditions for particles j and k are that the particle centers be separated by one diameter ($\|\mathbf{q}\| = \sigma$) and that the particles be moving towards each other ($\mathbf{p} \cdot \mathbf{q} < 0$) before collision. The collision time τ can be computed exactly using

[1]Collisions involving more than two particles are not considered. Even for high particle densities, collisions involving more than two particles are highly improbable due to the fact that exact collision times are used for updating the dynamics.

the collision condition $\|\mathbf{q}\| = \sigma$,

$$\tau = \frac{-\mathbf{q}\cdot\mathbf{p} - \sqrt{(\mathbf{q}\cdot\mathbf{p})^2 - (\mathbf{p}\cdot\mathbf{p})(\mathbf{q}\cdot\mathbf{q} - \sigma^2)}}{\frac{\mathbf{p}\cdot\mathbf{p}}{m}} \tag{3.1.3}$$

All potential collisions are considered, but only the minimum collision time over all potential collisions is important for capturing the system dynamics. See Subsections *binTime* and *updateTimes* in Appendix B.6.2 for more details on Equation (3.1.3) and its implementation in code. When particles j and k undergo collision, all states except $\mathbf{p}^j$ and $\mathbf{p}^k$ are unaffected. Let $\mathbf{\Gamma}_i$ and $\mathbf{\Gamma}_f$ represent the system state before and after collision, respectively,

$$\begin{aligned} \mathbf{\Gamma}_f &:= \begin{pmatrix} \mathbf{Q}_f \\ \mathbf{P}_f \end{pmatrix}, \\ \mathbf{\Gamma}_i &:= \begin{pmatrix} \mathbf{Q}_i \\ \mathbf{P}_i \end{pmatrix}. \end{aligned}$$

The state $\mathbf{\Gamma}_f$ is updated according to the *generalized binary collision rule*

$$\begin{aligned} \mathbf{Q}_f &= \mathbf{Q}_i, \\ \mathbf{p}_f^l &= \mathbf{p}_i^l, \quad l \neq j, k, \\ \mathbf{p}_f^j &= \beta_j \mathbf{p}_i^j + \gamma_j \mathbf{p}_i^k + \alpha d\mathbf{p}, \\ \mathbf{p}_f^k &= \beta_k \mathbf{p}_i^j + \gamma_k \mathbf{p}_i^k - \alpha d\mathbf{p}, \end{aligned} \tag{3.1.4}$$

where $d\mathbf{p} := (\mathbf{p}\cdot\mathbf{q})\mathbf{q}/\sigma^2$. Subscripts i and f denote states before and after collision, respectively. The collision rule should be such that the particles are indistinguishable at collision, and therefore be invariant under change of particle identity ($j \to k, k \to j$). This requires that

$$\begin{aligned} \beta_j &= \gamma_k := \beta, \\ \beta_k &= \gamma_j := \gamma. \end{aligned} \tag{3.1.5}$$

As in the elastic case, it is desired that the collision rule described by Equation (3.1.4) conserves both momentum and kinetic energy. To conserve momentum, it is required that

$$\begin{aligned} \mathbf{p}_i^j + \mathbf{p}_i^k &= \mathbf{p}_f^j + \mathbf{p}_f^k, \\ &= (\beta + \gamma)\mathbf{p}_i^j + (\gamma + \beta)\mathbf{p}_i^k, \end{aligned}$$

and so

$$\begin{aligned}\beta+\gamma &= 1,\\ \gamma &= 1-\beta.\end{aligned}$$

Putting the above into Equation (3.1.4),

$$\begin{aligned}\mathbf{Q}_f &= \mathbf{Q}_i,\\ \mathbf{p}_f^l &= \mathbf{p}_i^l, \quad l \neq j,k,\\ \mathbf{p}_f^j &= \beta\mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p},\\ \mathbf{p}_f^k &= (1-\beta)\mathbf{p}_i^j + \beta\mathbf{p}_i^k - \alpha d\mathbf{p},\end{aligned} \tag{3.1.6}$$

The parameter β represents the amount of each particle's normal momentum to be preserved after collision; i.e. this is our tuning parameter for modifying the system's intrinsic chaos. Assuming the collision occurs at time τ, Equation (3.1.6) can also be represented by a nonlinear transformation $\mathbf{M}$ that depends on $\mathbf{\Gamma}_i$,

$$\begin{aligned}\mathbf{\Gamma}_f(\tau) &= \mathbf{M}^\tau(\mathbf{\Gamma}_i(\tau))\\ &= \begin{pmatrix} \mathbf{Q}_i(\tau) \\ \mathbf{A}\mathbf{Q}_i(\tau) + \mathbf{B}\mathbf{P}_i(\tau) \end{pmatrix}.\end{aligned} \tag{3.1.7}$$

In Equation (3.1.7), matrices $\mathbf{A}$ and $\mathbf{B}$ are $2N \times 2N$ dimensional and are each composed of N 2×2 submatrices such that

$$\mathbf{p}_f^m = \mathbf{a}_{mm}\mathbf{q}_i^m + \mathbf{a}_{mn}\mathbf{q}_i^n + \mathbf{b}_{mm}\mathbf{p}_i^m + \mathbf{b}_{mn}\mathbf{p}_i^n, \tag{3.1.8}$$

for some $1 \leq m < n \leq N$. Combining Equations (3.1.4) and (3.1.8) yields

$$\begin{aligned}\mathbf{a}_{jj} &= -\alpha\frac{(\mathbf{p}\cdot\mathbf{q})}{\sigma^2}\mathbf{I}_{2\times 2} = \mathbf{a}_{kk},\\ \mathbf{a}_{jk} &= \alpha\frac{(\mathbf{p}\cdot\mathbf{q})}{\sigma^2}\mathbf{I}_{2\times 2} = \mathbf{a}_{kj},\\ \mathbf{a}_{mn} &= \mathbf{0}_{2\times 2} \text{ if } m \neq n \text{ and } (m,n) \notin \{(j,j),(k,k)\},\\ \mathbf{b}_{mm} &= \mathbf{I}_{2\times 2} \text{ if } m \neq j \text{ and } m \neq k,\\ \mathbf{b}_{mm} &= \beta\mathbf{I}_{2\times 2} \text{ if } m = j \text{ or } m = k,\\ \mathbf{b}_{mn} &= \mathbf{0}_{2\times 2} \text{ if } (m,n) \notin \{(j,j),(j,k),(k,j),(k,k)\},\\ \mathbf{b}_{mn} &= (1-\beta)\mathbf{I}_{2\times 2} \text{ if } (m,n) \in \{(j,k),(k,j)\}.\end{aligned}$$

The matrices $\mathbf{I}_{2\times 2}$ and $\mathbf{0}_{2\times 2}$ are the 2×2 identity and zero matrices, respectively. To solve for α, combine the above with conservation of kinetic energy at collision

$$\begin{aligned} |\mathbf{p}_i^j|^2 + |\mathbf{p}_i^k|^2 &= |\mathbf{p}_f^j|^2 + |\mathbf{p}_f^k|^2, \\ &= |\beta\mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p}|^2 + |(1-\beta)\mathbf{p}_i^j + \beta\mathbf{p}_i^k - \alpha d\mathbf{p}|^2, \end{aligned} \tag{3.1.9}$$

which leads to the following quadratic equation in α:

$$\frac{(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}\alpha^2 - \frac{(2\beta-1)(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}\alpha + \beta(\beta-1)|\mathbf{p}|^2 = 0. \tag{3.1.10}$$

It is convenient to make the following definitions for subsequent calculations:

$$\begin{aligned} a_\alpha &:= \frac{(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ b_\alpha &:= -\frac{(2\beta-1)(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ c_\alpha &:= \beta(\beta-1)|\mathbf{p}|^2. \end{aligned}$$

The consequence of Equation (3.1.10) is that, in order to conserve energy, the value of α depends on the relative position and momentum at collision,

$$\begin{aligned} \alpha &= \frac{-b_\alpha + \sqrt{b_\alpha^2 - 4a_\alpha c_\alpha}}{2a_\alpha} \\ &= \frac{2\beta-1}{2} + \frac{\sqrt{b_\alpha^2 - 4a_\alpha c_\alpha}}{2a_\alpha} \end{aligned} \tag{3.1.11}$$

The choice of positive sign in Equation (3.1.11) is not arbitrary. To avoid particle overlap after collision, the projection of the relative momentum vector after collision $\mathbf{p}_f := \mathbf{p}_f^k - \mathbf{p}_f^j$ onto the relative position vector $\mathbf{q}_f = \mathbf{q}$ should be greater than or equal to 0,

$$\begin{aligned} 0 \le \mathbf{p}_f \cdot \mathbf{q} &= [(1-2\beta)\mathbf{p}_i^j + (2\beta-1)\mathbf{p}_i^k - 2\alpha d\mathbf{p}]\cdot\mathbf{q} \\ &= (2\beta-1)(\mathbf{p}\cdot\mathbf{q}) - 2\alpha\frac{(\mathbf{p}\cdot\mathbf{q})(\mathbf{q}\cdot\mathbf{q})}{\sigma^2} \\ &= (2\beta-1-2\alpha)(\mathbf{p}\cdot\mathbf{q}). \end{aligned} \tag{3.1.12}$$

Equation (3.1.12) uses the fact that when the particles are in contact, $\mathbf{q}\cdot\mathbf{q} = \sigma^2$. In order for collision to occur, the particles must have had negative relative momentum along the relative position vector before collision ($\mathbf{p}\cdot\mathbf{q} \leq 0$), and therefore, for $\mathbf{p}_f\cdot\mathbf{q} \geq 0$, it is required that

$$\begin{aligned} 2\beta - 1 - 2\alpha &\leq 0, \\ \alpha &\geq \frac{2\beta - 1}{2}. \end{aligned} \tag{3.1.13}$$

The inequality in Equation (3.1.13) justifies the use of the positive sign in Equation (3.1.11). Additionally, it is required that the discriminant $b_\alpha^2 - 4a_\alpha c_\alpha$ is non-negative to ensure real-valued velocities after collision; this constraint requires that $\beta \in [0, 1]$[2]. There are two special cases worth discussing here: $\beta = 0$ and $\beta = 1$. When $\beta = 1$, $\alpha = 1$ and the generalized collision rule of Equation (3.1.6) reduces to the elastic collision rule

$$\begin{aligned} \mathbf{Q}_f &= \mathbf{Q}_i, \\ \mathbf{p}_f^l &= \mathbf{p}_i^l, \quad l \neq j, k, \\ \mathbf{p}_f^j &= \mathbf{p}_i^j + d\mathbf{p}, \\ \mathbf{p}_f^k &= \mathbf{p}_i^k - d\mathbf{p}, \end{aligned} \tag{3.1.14}$$

which has been studied extensively [16, 21, 31]. A summary of important results for the elastic collision rule is presented in Chapter 4. These results will then be extended to the generalized collision rule in Chapter 5. When $\beta = 0$, $\alpha = 0$ and the generalized collision rule of Equation (3.1.6) results in the two colliding particles swapping momenta at collision

$$\begin{aligned} \mathbf{Q}_f &= \mathbf{Q}_i, \\ \mathbf{p}_f^l &= \mathbf{p}_i^l, \quad l \neq j, k, \\ \mathbf{p}_f^j &= \mathbf{p}_i^k, \\ \mathbf{p}_f^k &= \mathbf{p}_i^j. \end{aligned} \tag{3.1.15}$$

Putting the transformations together for the free flight and collision stages, and further assuming that n collisions occur before time t at times $\tau_1, \cdots, \tau_n$, the state at time t, $\mathbf{\Gamma}(t)$, is

$$\mathbf{\Gamma}(t) = \mathbf{\Phi}^{t-\tau_n}_{\mathbf{\Gamma}(\tau_n)} \circ \mathbf{M}^{\tau_n} \circ \mathbf{\Phi}^{\tau_n-\tau_{n-1}}_{\mathbf{\Gamma}(\tau_{n-1})} \circ \cdots \circ \mathbf{\Phi}^{\tau_2-\tau_1}_{\mathbf{\Gamma}(\tau_1)} \circ \mathbf{M}^{\tau_1} \circ \mathbf{\Phi}^{\tau_1}_{\mathbf{\Gamma}(0)}(\mathbf{\Gamma}(0)), \tag{3.1.16}$$

where $\mathbf{\Gamma}(0)$ is the initial condition at $t = 0$ and $\circ$ denotes functional composition.

[2]See Appendix A.2.1 for justification of this claim.

3.2 Inverse (Backward-Time) Collision Rule

To construct the inverse, $\mathbf{M}^{-1} : \mathbb{R}^{4N} \to \mathbb{R}^{4N}$, of the generalized collision rule of Equation (3.1.7), it is required that

$$\begin{aligned} \mathbf{\Gamma}_i &= \mathbf{M}^{-1}(\mathbf{\Gamma}_f) \\ &\coloneqq \mathbf{M}^{-1}(\mathbf{M}(\mathbf{\Gamma}_i)). \end{aligned} \tag{3.2.1}$$

Similarly, the following must also hold:

$$\begin{aligned} \mathbf{\Gamma}_f &= \mathbf{M}(\mathbf{\Gamma}_i) \\ &\coloneqq \mathbf{M}(\mathbf{M}^{-1}(\mathbf{\Gamma}_f)). \end{aligned} \tag{3.2.2}$$

Putting together Equations (3.2.1) and (3.2.2), $\mathbf{M}^{-1} \circ \mathbf{M} = \mathbf{M} \circ \mathbf{M}^{-1} = I_{4N\times 4N}$. A physical interpretation of the inverse is as the *backward-time* collision of the generalized rule from Equation (3.1.7) (see Figure 3.1).

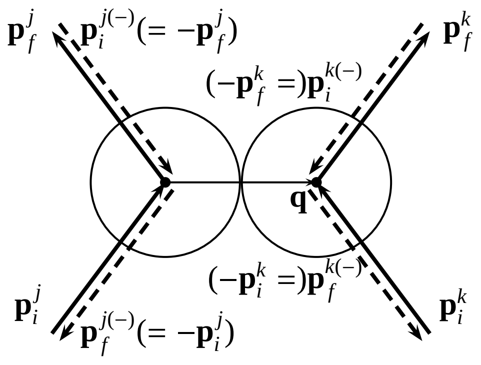

$\mathbf{p}_f^j$
$\mathbf{p}_i^{j(-)}(= -\mathbf{p}_f^j)$
$\mathbf{p}_f^k$
$(-\mathbf{p}_f^k =)\mathbf{p}_i^{k(-)}$
$\mathbf{q}$
$(-\mathbf{p}_i^k =)\mathbf{p}_f^{k(-)}$
$\mathbf{p}_i^j$
$\mathbf{p}_f^{j(-)}(= -\mathbf{p}_i^j)$
$\mathbf{p}_i^k$

FIGURE 3.1. Two-particle collision in forward- (solid lines) and backward- (dashed lines) time. Disks j and k undergo a collision in forward-time (no superscript) and in backward-time (superscript $(-)$). The inverse (backward-time) collision rule can be interpreted as the collision rule that takes the time-reversal of the states after forward-time collision to the time-reversal of the states before the same collision.

Using physical intuition, it can be stated that the backward-time collision rule must have the same form as Equation (3.1.6), that is

$$\begin{aligned}
\mathbf{Q}_f^{(-)} &= \mathbf{Q}_i^{(-)}, \\
\mathbf{p}_f^{l(-)} &= \mathbf{p}_i^{l(-)}, \quad l \neq j, k, \\
\mathbf{p}_f^{j(-)} &= \beta^{(-)}\mathbf{p}_i^{j(-)} + (1-\beta^{(-)})\mathbf{p}_i^{k(-)} + \alpha^{(-)} d\mathbf{p}^{(-)}, \\
\mathbf{p}_f^{k(-)} &= (1-\beta^{(-)})\mathbf{p}_i^{j(-)} + \beta^{(-)}\mathbf{p}_i^{k(-)} - \alpha^{(-)} d\mathbf{p}^{(-)}
\end{aligned} \tag{3.2.3}$$

The arguments from section 3.1.2 have been incorporated into Equation (3.2.3). The goal is now to characterize the coefficients $\beta^{(-)}$ and $\alpha^{(-)}$ in terms of the forward-time coefficients β and α. Again, assuming that particles j and k undergo collision, and using the fact that $\mathbf{M}^{-1} \circ \mathbf{M} = I_{4N\times 4N}$, the states after the collision defined by Equation (3.1.6) can be brought back to their respective states before collision by applying Equation (3.2.3),

$$\begin{aligned}
\mathbf{Q}_i &= \mathbf{Q}_f, \\
\mathbf{p}_i^l &= \mathbf{p}_f^l, \quad l \neq j, k, \\
\mathbf{p}_i^j &= \beta^{(-)}[\beta\mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p}] \\
&\quad + (1-\beta^{(-)})[(1-\beta)\mathbf{p}_i^j + \beta\mathbf{p}_i^k - \alpha d\mathbf{p}] \\
&\quad + \alpha^{(-)} d\mathbf{p}^{(-)}, \\
\mathbf{p}_i^k &= (1-\beta^{(-)})[\beta\mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p}] \\
&\quad + \beta^{(-)}[(1-\beta)\mathbf{p}_i^j + \beta\mathbf{p}_i^k - \alpha d\mathbf{p}] \\
&\quad - \alpha^{(-)} d\mathbf{p}^{(-)},
\end{aligned} \tag{3.2.4}$$

where $d\mathbf{p}^{(-)} := (\mathbf{p}_f \cdot \mathbf{q})\mathbf{q}/\sigma^2$ and $\mathbf{p}_f := \mathbf{p}_f^k - \mathbf{p}_f^j = (2\beta - 1)\mathbf{p} - 2\alpha d\mathbf{p}$. Working out the algebra yields $d\mathbf{p}^{(-)} = (2(\beta - \alpha) - 1)d\mathbf{p}$. The equalities in Equation (3.2.4) hold only when the following constraints are met:

$$\begin{aligned}
(2\beta - 1)\beta^{(-)} - \beta &= 0, \\
2\alpha\beta^{(-)} - \alpha + \alpha^{(-)}(2(\beta - \alpha) - 1) &= 0.
\end{aligned} \tag{3.2.5}$$

Solving for $\beta^{(-)}$ and $\alpha^{(-)}$ in Equation (3.2.5) gives (3.2.6):

$$\begin{aligned}
\beta^{(-)} &= \frac{\beta}{2\beta - 1} \\
\alpha^{(-)} &= -\frac{\alpha}{(2\beta - 1)(2(\beta - \alpha) - 1))}.
\end{aligned} \tag{3.2.6}$$

Therefore, the backward-time collision can be written in terms of a corresponding forward-time collision (and vice-versa). From Equation (3.2.6), it is clear that the generalized collision rule has a well-defined inverse for $\beta \in [0,1] \setminus \{\frac{1}{2}\}$. It should be noted that when $\beta = 0$ (velocity-swap) or $\beta = 1$ (elastic collision rule), $\beta^{(-)} = \beta = \alpha = \alpha^{(-)}$. This demonstrates that the dynamics for these two special cases are time-reversible: the forward- and backward-time collision rules are equivalent. This symmetry in time manifests itself as a symmetry in the Lyapunov spectrum about a center index and, therefore, zero average growth or contraction in time of infinitesimal phase space volumes about a reference trajectory is observed [16, 21, 31].

3.3 Linearized Dynamics of Perturbations

In order to quantify the sensitivity to initial condition of the interacting particle system presented in the previous section, it is necessary to evaluate how infinitesimal perturbations evolve in time using linearization methods. There is no issue in propagating perturbations over time intervals where no collisions occur; the approach is discussed in Section 3.3.1. The issue arises at collision, as a collision in a perturbed trajectory does not necessarily occur at the same time as in the reference trajectory. The linearization technique used to circumvent this difficulty is from Dellago *et al.* [16] and is discussed discussed in Section 3.3.2.

A perturbation vector, $\delta\mathbf{\Gamma}$, can be written in the same form as the state $\mathbf{\Gamma}$

$$\begin{aligned} \delta\mathbf{\Gamma} &:= \begin{pmatrix} \delta\mathbf{Q} \\ \delta\mathbf{P} \end{pmatrix} \\ &= \begin{pmatrix} \delta\mathbf{q}^1 \\ \vdots \\ \delta\mathbf{q}^N \\ \delta\mathbf{p}^1 \\ \vdots \\ \delta\mathbf{p}^N \end{pmatrix} \end{aligned}$$

where $\delta\mathbf{q}^j := (\delta q^{1j}\ \delta q^{2j})^\top$ and $\delta\mathbf{p}^j := (\delta p^{1j}\ \delta p^{2j})^\top$ are the perturbations to the position and momentum of particle j, respectively. As for $\mathbf{\Gamma}$, $\delta\mathbf{\Gamma}$ is a $4N$-dimensional vector. As for the phase space dynamics, the linearized dynamics of perturbations is broken up into free flight and collision periods.

3.3.1 Free Flight

During free flight, an infinitesimal perturbation evolves according to

$$\frac{d(\delta\mathbf{\Gamma})}{dt} = J_{\mathbf{F}}(\mathbf{\Gamma})\delta\mathbf{\Gamma} \tag{3.3.1}$$

where $J_{\mathbf{F}}(\mathbf{\Gamma})$ is the Jacobian of $\mathbf{F}$ at $\mathbf{\Gamma}$

$$J_{\mathbf{F}}(\mathbf{\Gamma}) = \begin{pmatrix} \mathbf{0}_{2N\times 2N} & \mathbf{I}_{2N\times 2N} \\ \mathbf{0}_{2N\times 2N} & \mathbf{0}_{2N\times 2N} \end{pmatrix},$$

where $\mathbf{0}_{2N\times 2N}$ and $\mathbf{I}_{2N\times 2N}$ are the $2N \times 2N$ zero and identity matrices, respectively. It should be noticed that $J_{\mathbf{F}}$ does not depend on $\mathbf{\Gamma}$ and is everywhere constant. Therefore, the perturbation propagation map, $\mathbf{L}$, generated by $J_{\mathbf{F}}$ and $\delta\mathbf{\Gamma}(\tau)$ that takes a perturbation from time τ, $\delta\mathbf{\Gamma}(\tau)$, to the perturbation at some other time s, $\delta\mathbf{\Gamma}(s)$, is

$$\begin{aligned} \delta\mathbf{\Gamma}(s) =& \delta\mathbf{\Gamma}(\tau) + \begin{pmatrix} \frac{\delta\mathbf{P}}{m} \\ \mathbf{0} \end{pmatrix}(s-\tau) \\ =& (\mathbf{I}_{4N\times 4N} + (s-\tau)J_{\mathbf{F}}(\mathbf{\Gamma}(\tau)))\delta\mathbf{\Gamma}(\tau) \\ :=& \mathbf{L}^{s-\tau}_{\delta\mathbf{\Gamma}(\tau)}\delta\mathbf{\Gamma}(\tau). \end{aligned} \tag{3.3.2}$$

3.3.2 Collision

When collisions occur in the reference, or unperturbed, trajectory, an infinitesimal perturbation vector undergoes a discontinuous jump. For numerical implementation considerations, it is desired to evaluate the evolution of perturbations at collision using only information available from the reference trajectory. The construction of this map for elastic collisions ($\beta = 1$) is presented in [16], and will be discussed in detail and extended here.

In perturbed trajectories, collisions occur at times slightly offset from those of the reference trajectory (see Figure 3.2). The description of each quantity in Figure 3.2 is in Table

Quantity	Description
$\mathbf{\Gamma}_i$	state of reference trajectory immediately before collision
$\delta\mathbf{\Gamma}_i$	difference in state between reference and perturbed trajectories immediately before collision in reference trajectory
τ_c	time of collision in reference trajectory
$\delta\tau_c$	time difference between collisions in reference and perturbed trajectories. This quantity can be either positive or negative.
$\delta\mathbf{\Gamma}_c$	difference in state between reference trajectory before collision, at time τ_c, and perturbed trajectory before collision, at time $\tau_c + \delta\tau_c$
$\mathbf{\Gamma}_f$	state of reference trajectory immediately after collision
$\delta\mathbf{\Gamma}_f$	difference in state between reference trajectory immediately after collision and the propagation of the perturbed trajectory to τ_c.

TABLE 3.1. Description of offset quantities

3.1. To solve for the time offset $\delta\tau_c$, let $\delta\mathbf{q}_c := \delta\mathbf{q} + \frac{\mathbf{p}}{m}\delta\tau_c$ and $\delta\mathbf{q} := \delta\mathbf{q}_i^k - \delta\mathbf{q}_i^j$. At collision in the reference trajectory, particles j and k are in contact, therefore it is true that

$$\mathbf{q}\cdot\mathbf{q} = \sigma^2.$$

Similarly, in the perturbed trajectory, the particles collide at time $\tau + \delta\tau_c$ and so

$$(\mathbf{q} + \delta\mathbf{q}_c)\cdot(\mathbf{q} + \delta\mathbf{q}_c) = \sigma^2.$$

Therefore,

$$\begin{aligned} \mathbf{q}\cdot\mathbf{q} + 2\mathbf{q}\cdot\delta\mathbf{q}_c + O\left(|\delta\mathbf{q}_c|^2\right) &= \sigma^2, \\ \implies \mathbf{q}\cdot\delta\mathbf{q}_c &= O\left(|\delta\mathbf{q}_c|^2\right). \end{aligned} \tag{3.3.3}$$

It is desired to have only terms dependent upon $\mathbf{q}$, $\mathbf{p}$ and $\delta\mathbf{q}$ in Equation (3.3.3),

$$\begin{aligned} \mathbf{q}\cdot\delta\mathbf{q} + \frac{(\mathbf{q}\cdot\mathbf{p})}{m}\delta\tau_c &= O\left(|\delta\mathbf{q}|^2, \frac{(\delta\mathbf{q}\cdot\mathbf{p})}{m}\delta\tau_c, \frac{|\mathbf{p}|^2}{m^2}\delta\tau_c^2\right), \\ \implies \delta\tau_c &= -\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}/m\cdot\mathbf{q}} + O\left(|\delta\mathbf{q}|^2\right). \end{aligned} \tag{3.3.4}$$

It is assumed in Equation (3.3.4) that $\mathbf{q}$ and $\mathbf{p}$ are $o(1)$.

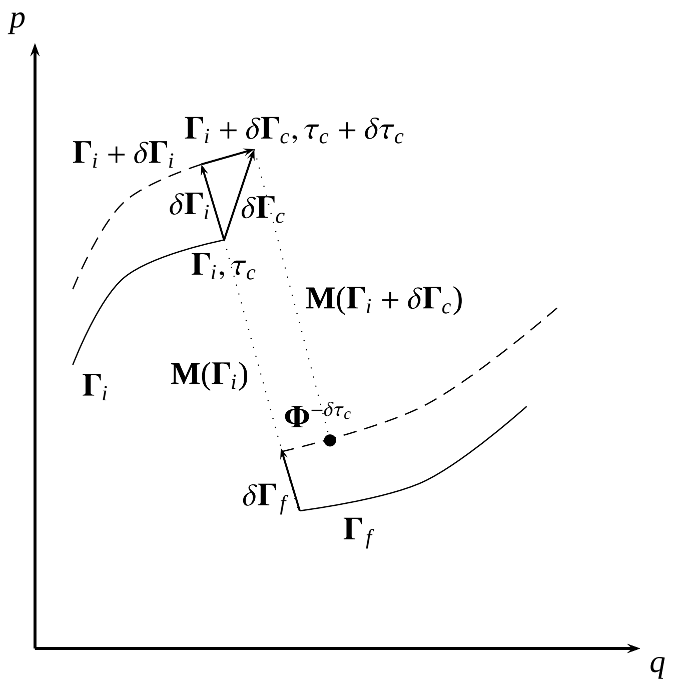

p
q
$\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c, \tau_c + \delta\tau_c$
$\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_i$
$\delta\boldsymbol{\Gamma}_i$
$\delta\boldsymbol{\Gamma}_c$
$\boldsymbol{\Gamma}_i, \tau_c$
$\mathbf{M}(\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c)$
$\boldsymbol{\Gamma}_i$
$\mathbf{M}(\boldsymbol{\Gamma}_i)$
$\boldsymbol{\Phi}^{-\delta\tau_c}$
$\delta\boldsymbol{\Gamma}_f$
$\boldsymbol{\Gamma}_f$

FIGURE 3.2. Effect of time offset on perturbation vectors. The time difference between the application of the discontinuous map **M** in the reference (unperturbed) and perturbed trajectories at the collision time in the reference trajectory (denoted τ_c) is $\delta\tau_c$. $\delta\boldsymbol{\Gamma}_f$ represents the offset vector immediately after collision in the reference trajectory. The state after collision in the perturbed trajectory, $\mathbf{M}(\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c)$, is propagated forwards or backwards in time (depending on the sign of $\delta\tau_c$) to τ_c using the propagator $\boldsymbol{\Phi}^{-\delta\tau_c}$.

It is desired to construct a transformation **S** that updates a perturbation after a collision in the reference trajectory, as in the following:

$$\delta\boldsymbol{\Gamma}_f(\tau_c) = \mathbf{S}(\boldsymbol{\Gamma}_i(\tau_c))\delta\boldsymbol{\Gamma}_i + O\left(|\delta\boldsymbol{\Gamma}_i|^2\right).$$

From Figure 3.2 it is clear, from geometric considerations, that

$$\begin{aligned}\delta\boldsymbol{\Gamma}_f &= \boldsymbol{\Phi}^{-\delta\tau_c}(\mathbf{M}(\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c)) - \boldsymbol{\Gamma}_f, \\ &= \mathbf{M}(\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c) + \mathbf{F}(\mathbf{M}(\boldsymbol{\Gamma}_i + \delta\boldsymbol{\Gamma}_c))(-\delta\tau_c) - \mathbf{M}(\boldsymbol{\Gamma}_i),\end{aligned} \tag{3.3.5}$$

where Equation (3.1.2) is used for time-propagation. Equation (3.1.2) can again be used to evaluate $\delta\mathbf{\Gamma}_c$,

$$\begin{aligned}\mathbf{\Gamma}_i + \delta\mathbf{\Gamma}_c &= \mathbf{\Gamma}_i + \delta\mathbf{\Gamma}_i + \mathbf{F}(\mathbf{\Gamma}_i + \delta\mathbf{\Gamma}_i)\delta\tau_c \\ \delta\mathbf{\Gamma}_c &= \delta\mathbf{\Gamma}_i + \mathbf{F}(\mathbf{\Gamma}_i)\delta\tau_c + J_{\mathbf{F}}(\mathbf{\Gamma}_i)\delta\mathbf{\Gamma}_i\delta\tau_c + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big), \\ &= \delta\mathbf{\Gamma}_i + \mathbf{F}(\mathbf{\Gamma}_i)\delta\tau_c + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big).\end{aligned} \tag{3.3.6}$$

The linearization of $\mathbf{M}(\mathbf{\Gamma}_i + \delta\mathbf{\Gamma}_c)$ about $\mathbf{\Gamma}_i$ is

$$\begin{aligned}\mathbf{M}(\mathbf{\Gamma}_i + \delta\mathbf{\Gamma}_c) &= \mathbf{M}(\mathbf{\Gamma}_i) + J_{\mathbf{M}}(\mathbf{\Gamma}_i)\delta\mathbf{\Gamma}_c + O\Big(|\delta\mathbf{\Gamma}_c|^2\Big) \\ &= \mathbf{M}(\mathbf{\Gamma}_i) + J_{\mathbf{M}}(\mathbf{\Gamma}_i)\delta\mathbf{\Gamma}_c + O\left(|\delta\mathbf{\Gamma}_i|^2, \frac{(\delta\mathbf{Q}_i\cdot\mathbf{P}_i)}{m}\delta\tau_c, \frac{|\mathbf{P}_i|^2}{m^2}\delta\tau_c^2\right) \\ &= \mathbf{M}(\mathbf{\Gamma}_i) + J_{\mathbf{M}}(\mathbf{\Gamma}_i)\delta\mathbf{\Gamma}_c + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big),\end{aligned} \tag{3.3.7}$$

where $J_{\mathbf{M}}(\mathbf{\Gamma})$ is the Jacobian of $\mathbf{M}$ evaluated at $\mathbf{\Gamma}$. Combining Equations (3.3.5) and (3.3.7), and using the linearity of $\mathbf{F}$, the offset vector after perturbation is

$$\begin{aligned}\delta\mathbf{\Gamma}_f &= \mathbf{M}(\mathbf{\Gamma}_i) + J_{\mathbf{M}}(\mathbf{\Gamma}_i)(\delta\mathbf{\Gamma}_i + \mathbf{F}(\mathbf{\Gamma}_i)\delta\tau_c) + \mathbf{F}(\mathbf{M}(\mathbf{\Gamma}_i))(-\delta\tau_c) - \mathbf{M}(\mathbf{\Gamma}_i) + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big), \\ &= J_{\mathbf{M}}(\mathbf{\Gamma}_i)\delta\mathbf{\Gamma}_i + \Big[J_{\mathbf{M}}(\mathbf{\Gamma}_i)\mathbf{F}(\mathbf{\Gamma}_i) - \mathbf{F}(\mathbf{M}(\mathbf{\Gamma}_i))\Big]\delta\tau_c + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big) \\ &:= \mathbf{S}(\mathbf{\Gamma}_i(\tau_c))\delta\mathbf{\Gamma}_i + O\Big(|\delta\mathbf{\Gamma}_i|^2\Big).\end{aligned} \tag{3.3.8}$$

Putting the transformations together for the free flight and collision stages, and assuming that n collisions occur before time t at times $\tau_1, \cdots, \tau_n$, $\delta\mathbf{\Gamma}(t)$ is

$$\delta\mathbf{\Gamma}(t) = \mathbf{L}^{t-\tau_n}_{\delta\mathbf{\Gamma}(\tau_n)} \cdot \mathbf{S}(\mathbf{\Gamma}_i(\tau_n)) \cdot \mathbf{L}^{\tau_n-\tau_{n-1}}_{\delta\mathbf{\Gamma}(\tau_{n-1})} \cdot \cdots \cdot \mathbf{L}^{\tau_2-\tau_1}_{\delta\mathbf{\Gamma}(\tau_1)} \cdot \mathbf{S}(\mathbf{\Gamma}_i(\tau_1)) \cdot \mathbf{L}^{\tau_1}_{\delta\mathbf{\Gamma}(0)} \cdot \delta\mathbf{\Gamma}(0), \tag{3.3.9}$$

where $\delta\mathbf{\Gamma}(0)$ is the initial perturbation at $t = 0$.

3.3.3 Constructing the Transformation S

The discussion begins with the definition of some helpful notation.

Note. Let $\mathbf{u}$ be an m-dimensional vector, $\mathbf{v}$ be n-dimensional ($m, n \geq 2$ are arbitrary) and w be a scalar. Define the matrix of partial derivatives $\frac{\partial\mathbf{v}}{\partial\mathbf{u}}$ as

$$\frac{\partial\mathbf{v}}{\partial\mathbf{u}} := \begin{pmatrix} \frac{\partial v^1}{\partial u^1} & \cdots & \frac{\partial v^1}{\partial u^m} \\ \vdots & \ddots & \vdots \\ \frac{\partial v^n}{\partial u^1} & \cdots & \frac{\partial v^n}{\partial u^m} \end{pmatrix}. \tag{3.3.10}$$

Let the vector of partial derivatives $\frac{\partial w}{\partial \mathbf{u}}$ be

$$\frac{\partial w}{\partial \mathbf{u}} := \begin{pmatrix} \frac{\partial w}{\partial u^1} \\ \vdots \\ \frac{\partial w}{\partial u^m} \end{pmatrix}. \tag{3.3.11}$$

Finally, let $\mathbf{u} \otimes \mathbf{v}$ denote the $n \times m$ outer product matrix

$$\mathbf{u} \otimes \mathbf{v} := \begin{pmatrix} u^1 v^1 & \cdots & u^1 v^m \\ \vdots & \ddots & \vdots \\ u^n v^1 & \cdots & u^n v^m \end{pmatrix}. \tag{3.3.12}$$

♣

Assume again that the indices $1 \leq j < k \leq N$ represent the colliding particles. From Equation (3.1.7), $J_{\mathbf{M}}(\mathbf{\Gamma})$ can be written as

$$J_{\mathbf{M}}(\mathbf{\Gamma}) = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{C} & \mathbf{D} \end{pmatrix}, \tag{3.3.13}$$

where the $2N \times 2N$ matrices $\mathbf{I}$ and $\mathbf{0}$ are the identity and zero matrices, respectively. Applying Equations (3.3.8) and (3.3.13),

$$\begin{aligned} \delta\mathbf{Q}_f &= \mathbf{I}\delta\mathbf{Q}_i + \left(\mathbf{I}\frac{\mathbf{P}_i}{m} - \frac{\mathbf{P}_f}{m}\right)\delta\tau_c, \\ &= \delta\mathbf{Q}_i + \left(\frac{\mathbf{P}_i}{m} - \frac{\mathbf{P}_f}{m}\right)\delta\tau_c, \end{aligned} \tag{3.3.14}$$

$$\begin{aligned} \delta\mathbf{P}_f &= \mathbf{C}\delta\mathbf{Q}_i + \mathbf{D}\delta\mathbf{P}_i + \left[\mathbf{D}\frac{\mathbf{P}_i}{m} - \mathbf{0}_{2N\times 1}\right]\delta\tau_c \\ &= \mathbf{C}\left(\delta\mathbf{Q}_i + \frac{\mathbf{P}_i}{m}\delta\tau_c\right) + \mathbf{D}\delta\mathbf{P}_i, \\ &:= \mathbf{C}\delta\mathbf{Q}_c + \mathbf{D}\delta\mathbf{P}_i, \end{aligned} \tag{3.3.15}$$

where $\delta\mathbf{Q}_c := \delta\mathbf{Q}_i + \frac{\mathbf{P}_i}{m}\delta\tau_c$. For an arbitrary $2N$-dimensional vector $\mathbf{v}$ the product $\mathbf{v}\delta\tau_c$ can be rewritten as

$$\begin{aligned} \mathbf{v}\delta\tau_c &:= \frac{1}{\mathbf{p}/m \cdot \mathbf{q}}(\delta\mathbf{q} \cdot \mathbf{q})\mathbf{v} + O\left(|\delta\mathbf{q}|^2\right) \\ &= \frac{1}{\mathbf{p}/m \cdot \mathbf{q}}(\mathbf{v} \otimes \mathbf{q})\delta\mathbf{q} + O\left(|\delta\mathbf{q}|^2\right) \\ &:= \frac{1}{\mathbf{p}/m \cdot \mathbf{q}}(\mathbf{v} \otimes \mathbf{q})\mathbf{T}\delta\mathbf{Q}_i + O\left(|\delta\mathbf{q}|^2\right), \end{aligned} \tag{3.3.16}$$

where $\mathbf{T}$ is the $2 \times 2N$ linear transformation

$$\mathbf{T} := \begin{pmatrix} \mathbf{0}_{2\times2} & \cdots & -\mathbf{I}_{2\times2} & \mathbf{0}_{2\times2} & \cdots & \mathbf{0}_{2\times2} & \mathbf{I}_{2\times2} & \mathbf{0}_{2\times2} & \cdots \end{pmatrix}. \tag{3.3.17}$$

The 2×2 identity matrices, $\mathbf{I}_{2\times2}$, are applied to the perturbation elements $\delta\mathbf{q}_i^j, \delta\mathbf{q}_i^k$ with a negative and positive sign, respectively. From Equation (3.3.17), it is true that $\mathbf{T}\delta\mathbf{Q}_i = \delta\mathbf{q} := \delta\mathbf{q}_i^k - \delta\mathbf{q}_i^j$, as required. Combining Equations 3.3.14 and 3.3.15 with 3.3.16 gives the desired form for the linear transformation $\mathbf{S}$ from Equation 3.3.9,

$$\mathbf{S}(\mathbf{\Gamma}_i(\tau_c)) = \begin{pmatrix} \mathbf{I}_{2N\times2N} + \frac{(\mathbf{P}\otimes\mathbf{q})\mathbf{T}}{\mathbf{p}\cdot\mathbf{q}} & \mathbf{0}_{2N\times2N} \\ \mathbf{C}\left[\mathbf{I}_{2N\times2N} + \frac{(\mathbf{P}_i\otimes\mathbf{q})\mathbf{T}}{\mathbf{p}\cdot\mathbf{q}}\right] & \mathbf{D} \end{pmatrix}, \tag{3.3.18}$$

where $\mathbf{P} := \mathbf{P}_f - \mathbf{P}_i$.

The $2N \times 2N$ matrices $\mathbf{C}$ and $\mathbf{D}$ from Equation (3.3.18) are composed of the 2×2 submatrices of the form $\mathbf{c}_{mn}$ and $\mathbf{d}_{mn}$, where mn refers to the particle pair m, n. These submatrices are computed by taking the partial derivatives

$$\begin{aligned} \mathbf{c}_{mn} &:= \frac{\partial\mathbf{p}_f^m}{\partial\mathbf{q}_i^n}, \\ \mathbf{d}_{mn} &:= \frac{\partial\mathbf{p}_f^m}{\partial\mathbf{p}_i^n}. \end{aligned} \tag{3.3.19}$$

When $(m, n) \notin \{(j, j), (j, k), (k, j), (k, k)\}$, $\mathbf{c}_{mn} = \mathbf{0}_{2\times2}$, the 2×2 zero matrix, and $\mathbf{d}_{mn} = \mathbf{I}_{2\times2}$, the 2×2 identity matrix. For particles interacting via the collision rule in Equation (3.1.6), unlike the elastic collision rule, the chain rule must be applied to the offset term $\alpha d\mathbf{p}$,

$$\begin{aligned}
\mathbf{c}_{jj} = \frac{\partial(\alpha d\mathbf{p})}{\partial \mathbf{q}_i^j} &= \alpha \frac{\partial(d\mathbf{p})}{\partial \mathbf{q}_i^j} + d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{q}_i^j} \\
&= -\frac{\alpha}{\sigma^2}[\mathbf{q} \otimes \mathbf{p} + (\mathbf{q} \cdot \mathbf{p})\mathbf{I}_{2\times 2}] + d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{q}_i^j} = \mathbf{c}_{kk}, \\
\mathbf{c}_{jk} = -\frac{\partial(\alpha d\mathbf{p})}{\partial \mathbf{q}_i^k} &= -\left[\alpha \frac{\partial(d\mathbf{p})}{\partial \mathbf{q}_i^k} + d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{q}_i^k}\right] \\
&= \frac{\alpha}{\sigma^2}[\mathbf{q} \otimes \mathbf{p} + (\mathbf{q} \cdot \mathbf{p})\mathbf{I}_{2\times 2}] - d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{q}_i^k} = \mathbf{c}_{kj}, \\
\mathbf{d}_{jj} = \frac{\partial}{\partial \mathbf{p}_i^j}(\beta \mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p}) &= \beta \mathbf{I}_{2\times 2} - \frac{\alpha}{\sigma^2}\mathbf{q} \otimes \mathbf{q} - d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{p}_i^j} \\
&= \mathbf{d}_{kk}, \\
\mathbf{d}_{jk} = \frac{\partial}{\partial \mathbf{p}_i^k}(\beta \mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p}) &= (1-\beta)\mathbf{I}_{2\times 2} + \frac{\alpha}{\sigma^2}\mathbf{q} \otimes \mathbf{q} + d\mathbf{p} \otimes \frac{\partial \alpha}{\partial \mathbf{p}_i^j} \\
&= \mathbf{d}_{kj}.
\end{aligned}$$

The computation of partial derivatives involving α requires implicit differentiation, leading to the following relation, for $x \in \{q_i^{1j}, q_i^{2j}, q_i^{1k}, q_i^{2k}, p_i^{1j}, p_i^{2j}, p_i^{1k}, p_i^{2k}\}$:

$$\frac{\partial \alpha}{\partial x} = A\frac{\partial a_\alpha}{\partial x} + B\frac{\partial b_\alpha}{\partial x} + C\frac{\partial c_\alpha}{\partial x}, \tag{3.3.20}$$

where

$$\begin{aligned}
A &:= -\frac{\alpha^2}{b_\alpha + 2a_\alpha \alpha}, \\
B &:= -\frac{\alpha}{b_\alpha + 2a_\alpha \alpha}, \\
C &:= -\frac{1}{b_\alpha + 2a_\alpha \alpha}.
\end{aligned}$$

The partial derivatives in Equation (3.3.20) are

$$
\begin{aligned}
\frac{\partial a_\alpha}{\partial \mathbf{q}_i^j} &= -\frac{2(\mathbf{q}\cdot\mathbf{p})}{\sigma^2}\mathbf{p} = -\frac{\partial a_\alpha}{\partial \mathbf{q}_i^k}\\
\frac{\partial a_\alpha}{\partial \mathbf{p}_i^j} &= -\frac{2(\mathbf{q}\cdot\mathbf{p})}{\sigma^2}\mathbf{q} = -\frac{\partial a_\alpha}{\partial \mathbf{p}_i^k}\\
\frac{\partial b_\alpha}{\partial \mathbf{q}_i^j} &= \frac{2(2\beta-1)(\mathbf{q}\cdot\mathbf{p})}{\sigma^2}\mathbf{p} = -\frac{\partial b_\alpha}{\partial \mathbf{q}_i^k}\\
\frac{\partial b_\alpha}{\partial \mathbf{p}_i^j} &= \frac{2(2\beta-1)(\mathbf{q}\cdot\mathbf{p})}{\sigma^2}\mathbf{q} = -\frac{\partial b_\alpha}{\partial \mathbf{p}_i^k}\\
\frac{\partial c_\alpha}{\partial \mathbf{q}_i^j} &= \mathbf{0}_{2\times 1} = \frac{\partial c_\alpha}{\partial \mathbf{q}_i^k}\\
\frac{\partial c_\alpha}{\partial \mathbf{p}_i^j} &= -2\beta(\beta-1)\mathbf{p} = -\frac{\partial c_\alpha}{\partial \mathbf{p}_i^k}.
\end{aligned}
$$

Since particles j and k are the only particles undergoing collision, all other perturbation components remain unaffected by the impact ($l \neq j, k$):

$$
\begin{aligned}
\delta\mathbf{q}_f^l &= \delta\mathbf{q}_i^l,\\
\delta\mathbf{p}_f^l &= \delta\mathbf{p}_i^l.
\end{aligned}
$$

For particles j and k, the perturbation vectors are mapped after a collision as

$$
\begin{aligned}
\delta\mathbf{q}_f^j &= \delta\mathbf{q}_i^j + \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{q}_f^k &= \delta\mathbf{q}_i^k - \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{p}_f^j &= \beta\delta\mathbf{p}_i^j + (1-\beta)\delta\mathbf{p}_i^k + \frac{\alpha}{\sigma^2}\left(\frac{-b_\alpha}{b_\alpha + 2a_\alpha\alpha}(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c\right)\\
&\quad - \frac{2\beta(\beta-1)(\mathbf{q}\cdot\mathbf{p})}{\sigma^2(b_\alpha + 2a_\alpha\alpha)}(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},\\
\delta\mathbf{p}_f^k &= \beta\delta\mathbf{p}_i^j + (1-\beta)\delta\mathbf{p}_i^k - \frac{\alpha}{\sigma^2}\left(\frac{-b_\alpha}{b_\alpha + 2a_\alpha\alpha}(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c\right]\\
&\quad + \frac{2\beta(\beta-1)(\mathbf{q}\cdot\mathbf{p})}{\sigma^2(b_\alpha + 2a_\alpha\alpha)}(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},
\end{aligned}
\tag{3.3.21}
$$

where $\delta\mathbf{p} := \delta\mathbf{p}_i^k - \delta\mathbf{p}_i^j$. The final two terms in Equation (3.3.21) can be simplified by introducing the parameter κ

$$
\kappa := \frac{2\beta(\beta-1)}{\sigma^2(b_\alpha + 2a_\alpha\alpha)}. \tag{3.3.22}
$$

After some algebra, Equation (3.3.21) can be rewritten as

$$\begin{aligned}
\delta\mathbf{q}_f^j &= \delta\mathbf{q}_i^j + \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{q}_f^k &= \delta\mathbf{q}_i^k - \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{p}_f^j &= \beta\delta\mathbf{p}_i^j + (1-\beta)\delta\mathbf{p}_i^k + \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&\quad + \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c - \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},\\
\delta\mathbf{p}_f^k &= (1-\beta)\delta\mathbf{p}_i^j + \beta\delta\mathbf{p}_i^k - \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&\quad - \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c + \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q}.
\end{aligned} \tag{3.3.23}$$

For the elastic collision rule ($\beta = 1$), Equation (3.3.23) takes a much simpler form [16, 21, 31],

$$\begin{aligned}
\delta\mathbf{q}_f^j &= \delta\mathbf{q}_i^j + \frac{(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{q}_f^k &= \delta\mathbf{q}_i^k - \frac{(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{p}_f^j &= \delta\mathbf{p}_i^j + \frac{1}{\sigma^2}[(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c],\\
\delta\mathbf{p}_f^k &= \delta\mathbf{p}_i^k - \frac{1}{\sigma^2}[(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c].
\end{aligned} \tag{3.3.24}$$

The elastic collision offset mapping above is provided for reference. In the next chapter, the dynamical properties of particles undergoing elastic collisions is presented.

3.4 Concluding Remarks

The offset mapping from Equation (3.3.23) provides a way for evaluating (to first-order) the effect of an infinitesimal perturbation to a reference trajectory in a dynamical system undergoing collisions between periods of free flight. This mapping will prove to be very useful in the remainder of this work, as the composition of this map with the free-flight mapping from Equation (3.3.2) can be used to compute the Lyapunov exponents using Benettin's algorithm 3. The applicability of Benettin's algorithm follows directly from Oseledec's theorem applied to the composition map $\mathbf{M}\circ\mathbf{\Phi}$ [16, 21, 35]. The matrix, A, in the theorem is replaced by the matrix product $\mathbf{S} \cdot \mathbf{L}$. The remainder of this work will be focused on numerical investigations using the derivations from this chapter.

Chapter 4
Dynamical Properties of Particle Systems Undergoing Elastic Collisions

The dynamical properties of particle systems interacting via elastic collisions are discussed in this chapter. The physical setting established in Chapter 3 is again valid here; namely, particles experience free flight in-between collisions. The elastic collision rule given in Equations (3.1.14) and (3.3.24) is restated here for convenience:

$$\begin{aligned}
\mathbf{Q}_f &= \mathbf{Q}_i, \\
\mathbf{p}^l_f &= \mathbf{p}^l_i, \quad l \neq j, k, \\
\mathbf{p}^j_f &= \mathbf{p}^j_i + d\mathbf{p}, \\
\mathbf{p}^k_f &= \mathbf{p}^k_i - d\mathbf{p}, \\
\delta\mathbf{q}^l_f &= \delta\mathbf{q}^l_i, \quad l \neq j, k, \\
\delta\mathbf{p}^l_f &= \delta\mathbf{p}^l_i, \quad l \neq j, k, \\
\delta\mathbf{q}^j_f &= \delta\mathbf{q}^j_i + \frac{(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right), \\
\delta\mathbf{q}^k_f &= \delta\mathbf{q}^k_i - \frac{(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right), \\
\delta\mathbf{p}^j_f &= \delta\mathbf{p}^j_i + \frac{1}{\sigma^2}[(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c], \\
\delta\mathbf{p}^k_f &= \delta\mathbf{p}^k_i - \frac{1}{\sigma^2}[(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q} + (\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c].
\end{aligned}$$

There are two reasons to use the elastic collision dynamics as an introduction to those of generalized collisions. Firstly, the results will serve as a baseline for comparison when the β-term of Equation (3.1.6) is not equal to 1. Secondly, the methods employed for studying the elastic case apply well when studying the generalized collision rule for varying β. This chapter will present some of the findings from important contributions to the topic in two-dimensions (also called hard-disk systems) [8, 16, 21, 91].

To limit the number of free parameters in the simulation, all numerical results are presented in reduced units where the particle mass m, diameter σ and the Boltzmann constant are equal to 1. Using these reduced units, the natural unit of time is $t^* = (mN\sigma^2/K)^{1/2}$ [16]. The constant N corresponds to the total number of particles, while the constant K represents the total kinetic energy: $K = \sum_{i=1}^{N} \|\mathbf{p}^i\|^2/(2m)$. Using these reduced units, the spectrum of Lyapunov exponents is measured with units $1/t^*$, and so the exponents scale with $K^{1/2}$. Using this knowledge, the kinetic energy can be eliminated as a free parameter; the restriction of $K/N = 1$ is enforced. The only free parameters are the number of particles N, the area density ρ, and the aspect ratio of the two-dimensional simulation box $A := L_y/L_x$, where L_y and L_x are the y and x lengths of the box with periodic boundary conditions[1]. The area density, ρ, is the area of the simulation box occupied by particles divided by the total area of the box (4.0.1)[2]

$$\rho = \frac{\pi N \sigma^2}{4AL_x^2}. \tag{4.0.1}$$

For the particle initial conditions, the positions are chosen to lie on a regular triangular lattice as this represents the optimal packing configuration for particles in a two-dimensional box. The momenta are drawn from a zero-mean Gaussian distribution with standard deviation equal to $(K/N)^{1/2}$. The reason for choosing the Gaussian distribution is convenience, as any initial distribution of momenta will relax over time to a Gaussian distribution in each of the x and y axes. Applying a constant shift in momentum to each particle before collision leads to the same constant shift in the momentum after collision and so, to prevent global drift, the average momentum of each particle in each axis is initialized to zero. Due to the conservative nature of the dynamics, this zero average momentum is conserved. The momenta are then rescaled so that the total kinetic energy of the system is equal to N [16, 21, 30, 31]. During each simulation, no collision event is missed. For implementation details, see Subsection *hardStep* in Appendix B.6.2.

[1]To match conventions in the literature, a periodic box is considered rather than the 2-torus $\mathbb{T}^2$ presented in the last chapter. These notions are equivalent.

[2]This convention differs from that used in Dellago *et. al* [16]: $\rho = N/(L_xL_y)$.

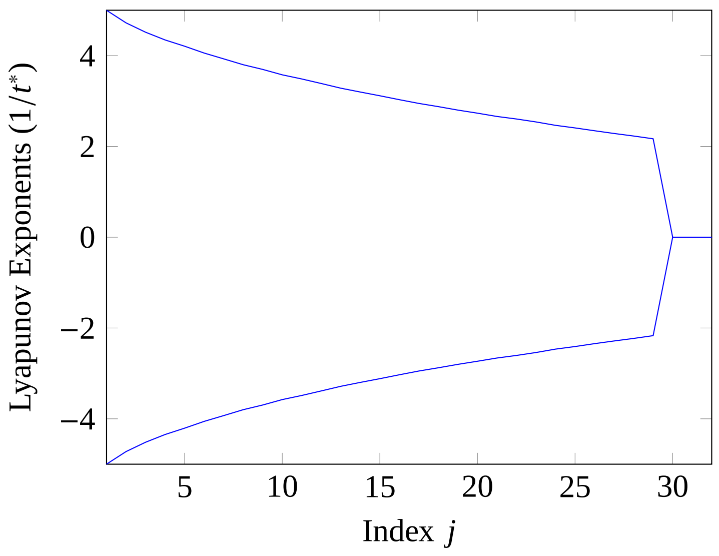


Figure 4.1. Plot of the Lyapunov spectrum of a hard disk system with $N = 16$, $A = \frac{\sqrt{3}}{2}$ and $\rho = 0.5498$. The results are obtained from a simulation run duration of $1000t^*$ time units; long enough to guarantee convergence of the spectrum to better than 1%. The spectrum can be broken up into two pieces by symmetry: the positive branch (indexed by $1 \le j \le 2N$) and the negative branch (indexed by $1 \le 4N - j + 1 \le 2N$). The plot demonstrates that the exponents come in pairs: $\lambda_j = -\lambda_{4N-j+1}$. This is known as *the conjugate-pairing rule*.

4.1 Lyapunov Exponents for Hard Disks

To evaluate the Lyapunov exponents for hard disk systems for a desired density, aspect ratio and particle number requires the use of data from every collision. If individual collisions are ignored, i.e. colliding particles are allowed to pass through one another rather than apply the collision map from Equation (3.1.6), the exponents would resemble those for a configuration with smaller density or number of particles or both. The computation of the Lyapunov exponents is insensitive to the reorthonormalization frequency, provided the time between successive reorthonormalizations is not so large as to result in the tangent vectors' collapse onto the maximally unstable direction. To avoid such collapse, in practice, reorthonormalization of tangent vectors is performed every $0.05t^* - 0.1t^*$ time steps.

Due to the *symplecticity*[3] of the tangent space dynamics for the hard disk system, each

[3]For arbitrary $m \in \mathbb{N}$, a $2m$-dimensional map $\mathbf{T}$ is symplectic if $\mathbf{T}^\top \mathbf{J} = -\mathbf{J}\mathbf{T}$, where the matrix $\mathbf{J}$ is given

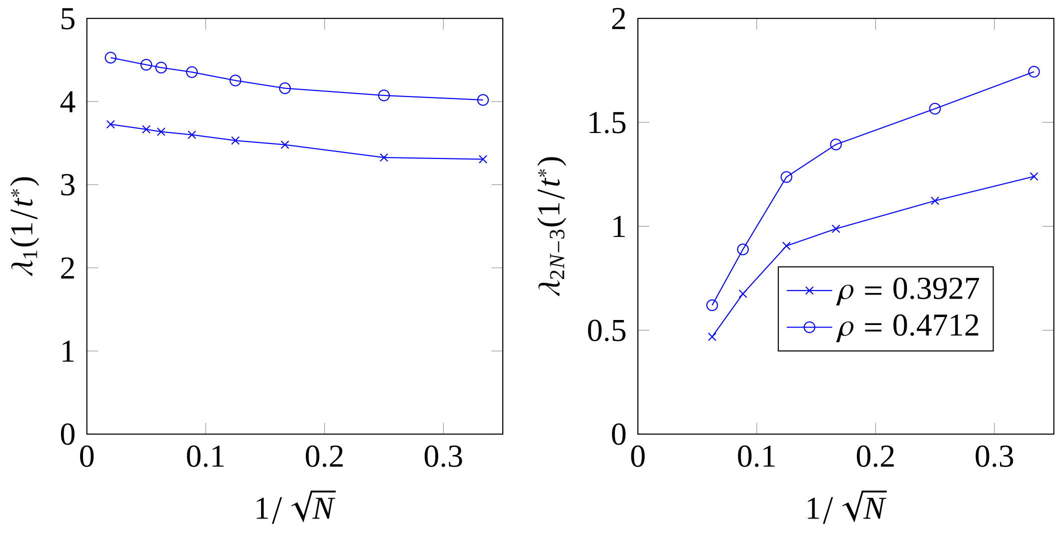


FIGURE 4.2. Plots showing dependence of the maximal exponent (λ_1) and the smallest positive exponent (λ_{2N-3}) as a function of $1/\sqrt{N}$ for two densities: $\rho = \{0.3927, 0.4712\}$, $A = \frac{\sqrt{3}}{2}$. The maximal exponent increases with increasing N, while the first step corresponding to λ_{2N-3} decreases with increasing N. The results are obtained from simulation run durations of $1000t^*$ time units; long enough to guarantee convergence of the spectrum to better than 1%.

positive exponent is paired with a negative exponent of the same absolute value [16, 17, 21]. In computing the exponents alone, only the positive half of the spectrum is computed to reduce computational effort. The symmetry of the Lyapunov spectrum is shown in Figure 4.1. Of particular note is the large jump in the spectrum at $j = 2N - 2$. This jump, and the origin of the six zero exponents present, will be discussed in greater detail in Sections 4.1.1 and 4.3.

A natural entry point into the study of hard disk systems is to identify how the maximal and smallest positive exponents are affected by the number of particles N and the density ρ. Some results showing these relationships are in Figure 4.2. For each additional particle, the

as

$$\mathbf{J} := \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & -\mathbf{I} \end{pmatrix}.$$

The matrices $\mathbf{I}$ and $\mathbf{0}$ are the $m \times m$ identity and zero matrices, respectively [17, 27].

number of degrees-of-freedom increases by 4, therefore it is not surprising that the maximal exponent λ_1 increases with increasing N[4]. The maximal exponent increases also with increasing density, as larger density means higher frequency of collisions [16]. It is surprising, however, that the smallest positive exponent λ_{2N-3} decreases with increasing N. This result is quite remarkable, as it corresponds with the creation of *steps* containing degenerate exponents near the center index $j = 2N$ as N increases (see Figure 4.5). The number of steps increases with N. Another quantity of interest is the rate of information production, which is quantified by the Kolmogorov-Sinai entropy. By information production, it is meant that initially indistinguishable trajectories in phase space become resolvable (i.e. information is produced) after some amount of time has elapsed. This is due to the exponential separation of nearby trajectories along the unstable directions in tangent space [18]. When certain conditions on the invariant measure μ are upheld[5], the Kolmogorov-Sinai entropy is simply the sum of all positive Lyapunov exponents [102, 103]

$$h_{KS} = \sum_{\{j:\lambda_j>0\}} \lambda_j. \tag{4.1.2}$$

For large N, the average rate of entropy production *per particle*, as quantified by the Kolmogorov-Sinai entropy from Equation (4.1.2), does not change with increasing particle number N at fixed density ρ (see Figure 4.3). Again, due to increased collision frequencies at higher densities, the entropy production per particle increases with increasing ρ [16].

[4]This is true up to the thermodynamic limit, where N is sufficiently large that no significant information about the hard disk system is gained by adding more particles [66, 68].

[5]From Young [103]: The necessary condition for equality in Pesin's theorem [73] is when the ergodic measure μ (from Theorem 1) is equivalent to the Lebesgue measure. This is true since the Liouville measure

$$d\mu_L := \frac{\prod_{j=1}^{N} dq^{1j}dq^{2j}dp^{1j}dp^{2j}}{\int_X \prod_{j=1}^{N} dq^{1j}dq^{2j}dp^{1j}dp^{2j}} \tag{4.1.1}$$

is an ergodic invariant measure on the submanifold defined by the conserved quantities for the hard disk system with elastic collisions. It is assumed for the analysis in Chapter 5 that this property persists for the generalized collision rule of Equation (3.1.6).

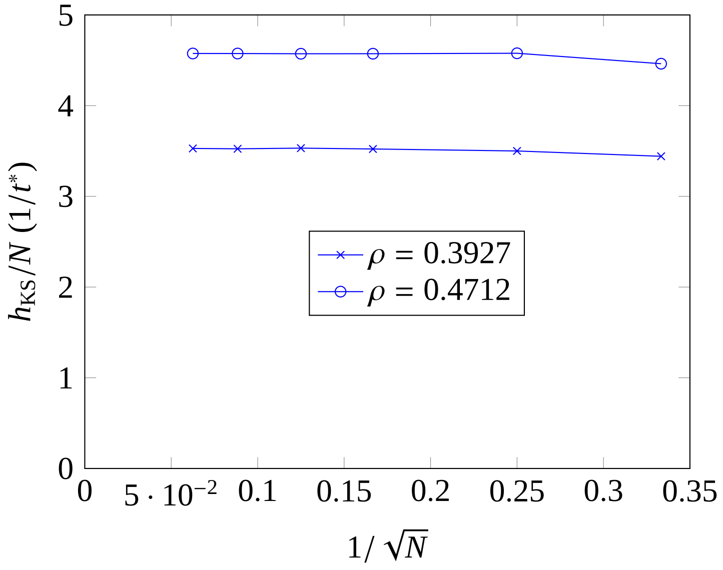


FIGURE 4.3. Plot showing that the average entropy production per particle, as quantified by Equation (4.1.2), is unaffected by the number of particles for large enough N, $A = \frac{\sqrt{3}}{2}$. The per-particle entropy production increases with increasing density ρ. The results are obtained from simulation run durations of $1000t^*$ time units; long enough to guarantee convergence of the spectrum to better than 1%.

4.1.1 Zero Exponents

As shown in Figures 4.1 and 4.5, there exist $2d + 2$ zero exponents independent of particle number N. For hard-disk systems, the number of spatial dimensions of the simulation box is $d = 2$. The origin of these zero exponents are the quantities left invariant by the dynamics. These quantities are invariance under global position shift (contributes one zero exponent for each spatial dimension, d total, corresponding to perturbation vectors $\mathbf{e}_1, \mathbf{e}_2$ in Equation (4.1.3)), invariance under global momentum shift (contributes d zero exponents corresponding to perturbation vectors $\mathbf{e}_3, \mathbf{e}_4$), conservation of kinetic energy (contributes one zero exponent corresponding to perturbation vector $\mathbf{e}_5$) and one additional zero exponent accounting for non-exponential growth tangent to the evolution of the system (along $\mathbf{e}_6$) [21, 31]. The neutral directions $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4, \mathbf{e}_5, \mathbf{e}_6\}$ corresponding to the conserved quantities are

$$\begin{aligned}
\mathbf{e}_1 &= \frac{1}{N}\begin{pmatrix}1 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0\end{pmatrix}^\top, \\
\mathbf{e}_2 &= \frac{1}{N}\begin{pmatrix}0 & 1 & 0 & 1 & \cdots & 0 & 0 & 0 & 0\end{pmatrix}^\top, \\
\mathbf{e}_3 &= \frac{1}{N}\begin{pmatrix}0 & 0 & 0 & 0 & \cdots & 1 & 0 & 1 & 0\end{pmatrix}^\top, \\
\mathbf{e}_4 &= \frac{1}{N}\begin{pmatrix}0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 1\end{pmatrix}^\top, \\
\mathbf{e}_5 &= \frac{1}{||\mathbf{P}||}\begin{pmatrix}0 & 0 & 0 & 0 & \cdots & p^{11} & p^{21} & \cdots & p^{1N} & p^{2N}\end{pmatrix}^\top, \\
\mathbf{e}_6 &= \frac{1}{||\mathbf{P}||}\begin{pmatrix}p^{11} & p^{21} & \cdots & p^{1N} & p^{2N} & 0 & 0 & \cdots & 0 & 0\end{pmatrix}^\top,
\end{aligned} \tag{4.1.3}$$

where $\mathbf{P} := (p^{11}\ p^{21}\ p^{12}\ p^{22}\ \cdots\ p^{1N}\ p^{2N})^\top$ is the vector containing all individual particle momenta. The subspace $\mathcal{N}(\mathbf{\Gamma}) := \langle \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \mathbf{e}_4, \mathbf{e}_5, \mathbf{e}_6 \rangle$ is known as the *center subspace* because elements of $\mathcal{N}(\mathbf{\Gamma})$ do not grow (or shrink) exponentially.

4.2 Measured Lyapunov Vectors for Hard Disk Systems

Benettin's algorithm computes a QR-factorization of the tangent space at each orthonormalization step (see Algorithm 3). The columns of the Q matrix are referred to as *Lyapunov* *vectors*. These measured Lyapunov vectors (also referred to as *modes*) come in two distinct classes. To the largest positive (negative, in absolute value) exponents are associated Lyapunov vectors which are highly localized in time. The structure of these vectors is highly dependent upon the reorthonormalization procedure and the collision geometries in between orthonormalization steps [92]. For the smallest positive (or least negative) exponents, the situation is much more interesting and will be discussed in Section 4.3.

4.2.1 Localization

The origin of the instability of trajectories in hard disk systems are collisions between particles. Therefore, it is expected that the most unstable directions should be determined by a small number of particles at any given time [21, 31, 91]. Taniguchi and Morriss proposed a localization measure for identifying the number of particles actively contributing to the unstable (and stable) directions in phase space [91]. Let the contribution of each particle j to the squared norm of the i^{th} Lyapunov vector ($1 \le i \le 2N$), $\delta\mathbf{\Gamma}_i$, be defined as

$$\gamma_j^{(i)} := \delta\mathbf{q}_j^{(i)} \cdot \delta\mathbf{q}_j^{(i)} + \delta\mathbf{p}_j^{(i)} \cdot \delta\mathbf{p}_j^{(i)}, \tag{4.2.1}$$

where $\delta\mathbf{q}_j^{(i)}, \delta\mathbf{p}_j^{(i)}$ are perturbations to the position and momentum components of particle j in the reference trajectory $\mathbf{\Gamma}$ corresponding to the mode i. An entropy-like quantity $S^{(i)}$ can be defined to identify the time-averaged number of particles contributing to the squared norm of the i^{th} Lyapunov vector, where

$$S^{(i)} := -\sum_{j=1}^{N} \langle \gamma_j^{(i)} \log \gamma_j^{(i)} \rangle. \tag{4.2.2}$$

In Equation (4.2.2), $\langle \cdot \rangle$ denotes the time average. The quantity $S^{(i)}$ has a maximum value of $\log N$ when each disk on average contributes equally to the norm of $\delta\mathbf{\Gamma}_i$. The maximum value occurs when $\gamma_j^{(i)} = 1/N$ for each particle j. Rather than work with $S^{(i)}$, it is more convenient to deal with the *localization width*

$$\mathcal{W}^{(i)} := \frac{1}{N} \exp[S^{(i)}]. \tag{4.2.3}$$

There are nice properties to Equation (4.2.3), namely that $1/N \leq \mathcal{W}^{(i)} \leq 1$ and that $\mathcal{W}^{(i)}$ is minimal when very few particles contribute to the norm of $\delta\boldsymbol{\Gamma}_i$ and maximal when each particle contributes equally to it. Figure 4.4 shows the average localization properties of Lyapunov vectors for hard disk systems. Reflecting the symmetry in the Lyapunov spectrum, the localization width is also symmetric about the center index. The vectors corresponding to the largest exponents are, on average, the most localized with respect to $\mathcal{W}^{(i)}$, as expected. Near the center index, larger localization widths indicate more particles actively contributing to the vectors' norms. This fact reaffirms that there are directions in phase space that are insensitive to collision. This is to be expected, since the zero exponents correspond to directions in tangent space associated with conserved quantities of the global dynamics. Particularly noteworthy is the fact that the comb-like structure observed in Figure 4.4 extends beyond the vectors with zero exponent. This means that there are additional vectors near the center index that have a global delocalized structure. This structure reflects the existence of global modes corresponding to nearly equal contribution of each particle to the Lyapunov vector norm, and hence, to the Lyapunov exponent magnitude. The qualitative features of the comb structure in Figure 4.4 are seen for any particle number N.

4.3 Coherent Vector Structure Near the Center Index

In Section 2.3, the Lyapunov vectors measured through the Modified Gram-Schmidt orthonormalization procedure were shown to be related to the backward Oseledec matrix (recall Equation (2.3.5)). Due to the symplectic nature of the tangent space dynamics, the subspace $\mathcal{N}(\boldsymbol{\Gamma})$ coincides directly with the element $E^{(m)}(\boldsymbol{\Gamma})$ of the Oseledec splitting associated with the zero exponent[6] [21]. Because of this coincidence, the center subspace $\mathcal{N}(\boldsymbol{\Gamma})$ is covariant. As previously mentioned, for large enough N, a step discontinuity appears in the Lyapunov spectrum near the center index (see Figure 4.5). These steps come with

[6]The index m denotes the index of the non-degenerate zero exponent.

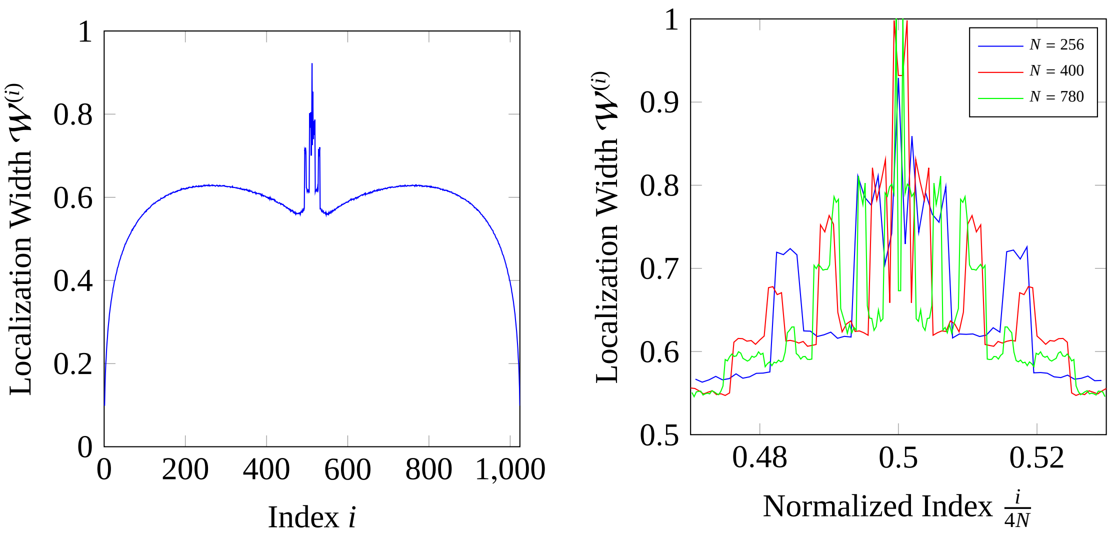


FIGURE 4.4. Left: Plot of the localization width $\mathcal{W}^{(i)}$ from Equation (4.2.3) for $N = 256$, $A = 1$ and $\rho = 0.3927$. The results are averaged over 900 samples each separated in time by t^*. The vectors associated with the largest exponents have small $\mathcal{W}^{(i)}$, corresponding to few particles contributing to its norm. The comb-like structure about the center index is due to coherent, delocalized structures in these Lyapunov vectors. Right: Plot of the localization width $\mathcal{W}^{(i)}$ for multiple N near the center index. The comb-like structure exists for large enough particle number N. The number of steps (or "teeth") in the structure grows with N.

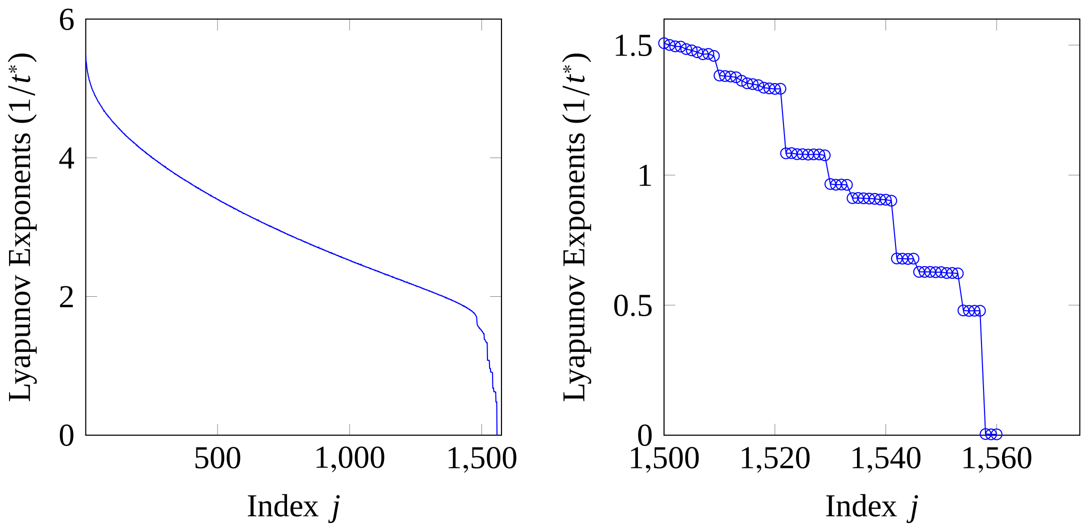


FIGURE 4.5. Plots of the Lyapunov spectrum of a hard disk system with $N = 780$, $A = 1$ and $\rho = 0.5498$. The results are obtained from simulation run durations of $1000t^*$ time units; long enough to guarantee convergence of the spectrum to better than 1%. Left: The positive branch of the Lyapunov spectrum. Right: The zoomed-in plot of the spectrum showing the step structure near the center index $j = 2N$. The exponents in the step structure are repeated with multiplicities of $2, 4$ or 8; depending upon wavenumber $\mathbf{k_n}$ (see Equation (4.3.3) for definition). The multiplicities associated with each step are dependent upon the number of particles N, the aspect ratio A and the density ρ.

different multiplicities; depending upon the number of particles N, the aspect ratio A and the density ρ. Before discussing the multiplicities, an important simplification for viewing the modes is discussed.

A special property of the near-zero Lyapunov vectors for the hard disk system is observed experimentally (see Figure 4.6). The Lyapunov vectors associated with the degenerate near-zero exponents are such that the angles between the position perturbation component $\delta\mathbf{Q}$ and the momentum perturbation component $\delta\mathbf{P}$ are nearly aligned [21, 31]. This alignment is measured using the law of cosines:

$$\langle \cos\Theta^{(i)} \rangle = \left\langle \frac{\sum_{j=1}^{N} \delta\mathbf{q}_j^{(i)} \cdot \delta\mathbf{p}_j^{(i)}}{||\delta\mathbf{Q}^{(i)}||\, ||\delta\mathbf{P}^{(i)}||} \right\rangle, \tag{4.3.1}$$

where $\delta\mathbf{Q}^{(i)}$ and $\delta\mathbf{P}^{(i)}$ are the position and momentum perturbation components (see Section 3.3 for definition). It can then be concluded that the position and momentum components contain the same directional information, and therefore only the position (or momentum) components of a perturbation vector need to be considered.

4.3.1 Mode Classification

To motivate some of the results that follow, first consider Figure 4.7. Following Eckmann *et. al* [21], a simple way to view the mode structure is by considering the position perturbations associated with each particle as sampling a continuous vector field over the simulation box $[0, L_x) \times [0, L_y)$[7]. Figure 4.7 shows the structure for modes belonging to subspaces with different associated Lyapunov exponents (see Figure 4.5). The modes associated with λ_{1539} and λ_{1548} show irrotational wave-like structures, while the λ_{1528} and λ_{1556} modes show divergence-free rotating vector fields. Any claims regarding the structure of these, and all other delocalized modes, needs to account for the observed differences between modes associated to different exponents. Due to the periodicity observed in the λ_{1528} and λ_{1556}

[7]This construction only makes sense when the number of particles is sufficient to sample the vector field on a typical length scale of its variation. Due to the delocalized nature of the modes, this is an appropriate assumption.

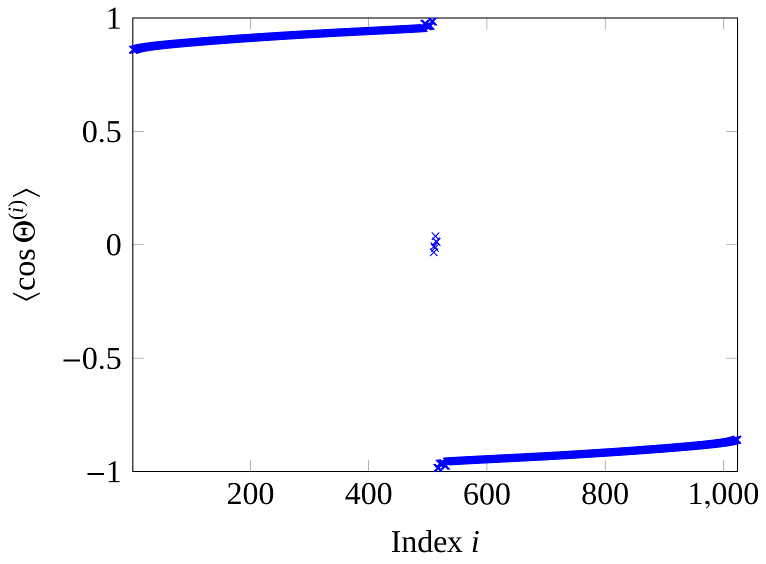


FIGURE 4.6. Plot of $\langle \cos \Theta^{(i)} \rangle$ from Equation (4.3.1) for $N = 256$, $A = 1$ and $\rho = 0.3927$. The results are averaged over 900 samples each separated in time by t^*. For vectors corresponding to degenerate exponents near the center index, the position perturbation $\delta\mathbf{Q}$ and momentum perturbation $\delta\mathbf{P}$ are nearly (anti-)parallel. This property makes visualizing the vectors easier.

modes, a good starting point would be to consider a description of the modes that utilizes a periodic vector field over the container [21]. It is well-known that a two-dimensional vector field can be uniquely split into a divergence-free component and an irrotational component via a generalized Helmholtz decomposition [65]. Armed with these observations, a formal classification of the mode structure for near-zero exponents follows from Eckmann *et. al* [21].

Due to the periodic character of the modes observed in Figure 4.7, it has been claimed that the mode structure is due to scalar modulations of the six vectors in Equation (4.1.3) corresponding to the continuous symmetries of the system. Scalar modulation means that, for vectors corresponding to exponents with index near the center index, the $\mathbf{e}_i$ from Equa-

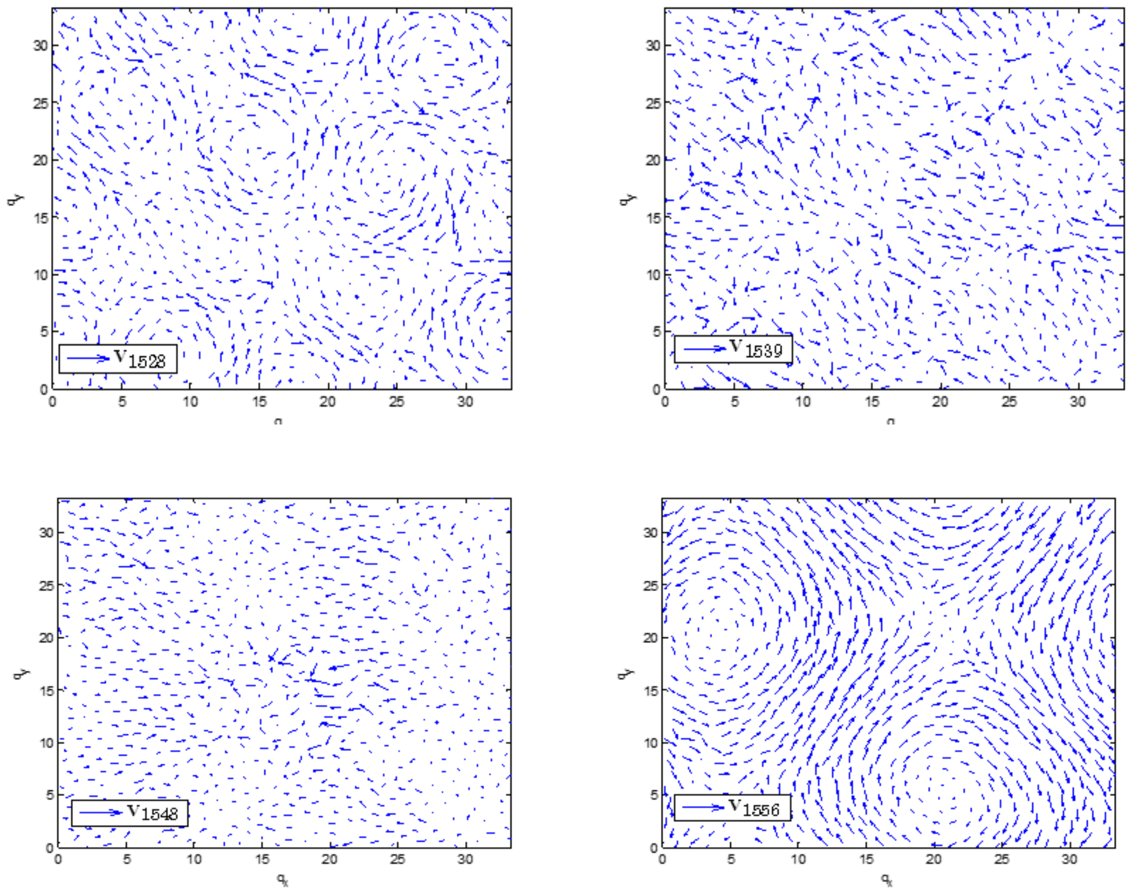


FIGURE 4.7. Observed delocalized (global) mode structure for exponents λ_{1528}(Top Left), λ_{1539}(Top Right), λ_{1548}(Bottom Left) and λ_{1556} (Bottom Right) from Figure 4.5 taken at time $1000t^*$ ($N = 780$, $A = 1$ and $\rho = 0.5498$). These structures are observed in the orthogonal Lyapunov vectors only, not in the full dynamics. This may be due to the difference in timescale between the fastest (localized) and slowest (large length-scale) modes. Top Left: Divergence-free periodic mode $\mathbf{v}_{1528}$ from the third transverse branch $T(3, 3)$. Top Right: Lyapunov mode $\mathbf{v}_{1539}$ from the second longitudinal branch $LP(2, 2)$. Bottom Left: Lyapunov mode $\mathbf{v}_{1548}$ from the first longitudinal branch $LP(1, 1)$. Bottom Right: Periodic divergence-free mode $\mathbf{v}_{1556}$ from the first transverse branch $T(1, 1)$. The features shown in both modes $\mathbf{v}_{1528}$ and $\mathbf{v}_{1556}$ show that some modes are purely periodic over the container $[0, L_x) \times [0, L_y)$ and non-propagating. The wavenumber associated with the spatial oscillations is increasing for increasing exponents. Other modes show irrotational behavior, with a non-uniform vector length, as in modes $\mathbf{v}_{1539}$ and $\mathbf{v}_{1548}$.

$\mathbf{n}$	Basis for A	Dim
$(n_x, 0)$	$\cos(n_x k_x x)$, $\sin(n_x k_x x)$	2
$(0, n_y)$	$\cos(n_y k_y y)$, $\sin(n_y k_y y)$	2
(n_x, n_y)	$\cos(n_x k_x x)\cos(n_y k_y y)$, $\sin(n_x k_x, x)\cos(n_y k_y y)$, $\cos(n_x k_x x)\sin(n_y k_y y)$, $\sin(n_x k_x x)\sin(n_y k_y y)$	4

TABLE 4.1. Table showing basis elements for a functional decomposition of A from Equation (4.3.2) for different $\mathbf{n}$.

tion (4.1.3)are replaced by $\mathbf{A}(\mathbf{Q})\mathbf{e}_i$, where the matrix $\mathbf{A}(\mathbf{Q})$ is

$$\begin{aligned}\mathbf{A}(\mathbf{Q}) &= \{a_{r,s}\}_{1\le r,s\le 4N},\\ a_{4(n-1)+1,4(n-1)+1}(\mathbf{Q}) &= a_{4(n-1)+1,4(n-1)+2}(\mathbf{Q}) = A(q^{1n}, q^{2n}) \quad 1 \le n \le N,\\ a_{r,s}(\mathbf{Q}) &= 0 \quad \text{for all other } r, s,\\ A(x,y) &:= \sum_{|l|=n_x,|m|=n_y} c_{l,m} \exp[i(lk_x x + mk_y y)],\end{aligned} \tag{4.3.2}$$

where n_x and k_x (n_y and k_y) represent the node number and wavenumber, respectively, in the x (y) direction

$$\begin{aligned}\mathbf{n} &:=(n_x, n_y) \in \mathbb{N}^2\\ \mathbf{k_n} &:=(n_x k_x, n_y k_y) = (\frac{2\pi n_x}{L_x}, \frac{2\pi n_y}{L_y}),\end{aligned} \tag{4.3.3}$$

of the periodic vector field A. For a given $\mathbf{k_n}$, the function A can be represented by basis functions shown in Table 4.1:

The fact that $\delta\mathbf{Q}$ is nearly (anti-)parallel to $\delta\mathbf{P}$ for modes with near-zero exponent can now be used; only modulations to the vectors $\mathbf{e}_1, \mathbf{e}_2$ and $\mathbf{e}_6$ from Equation (4.1.3) need to be considered [21]. Since each of these three perturbation vectors can be modulated by A, which has dimension four when $n_x, n_y \neq 0$ (dimension 2, when either n_x or $n_y = 0$), a twelve-fold (six-fold if either n_x or $n_y = 0$) degeneracy of each repeated exponent is expected. However, the Lyapunov exponents near zero are not dependent upon $\mathbf{k_n}$ only. The degeneracy is split up into two branches; one with eight-fold degeneracy (four-fold if either n_x or $n_y = 0$) and another with four-fold degeneracy (two-fold if either n_x or $n_y = 0$). The four(two)-fold degenerate branch is called *Transverse*, while the eight(four)-fold branch is

$\mathbf{n}$	Basis for $T(\mathbf{n})$
$(n_x, 0)$	$\begin{pmatrix} 0 \\ c_x \end{pmatrix}, \begin{pmatrix} 0 \\ s_x \end{pmatrix}$
$(0, n_y)$	$\begin{pmatrix} c_y \\ 0 \end{pmatrix}, \begin{pmatrix} s_y \\ 0 \end{pmatrix}$
(n_x, n_y)	$\begin{pmatrix} \frac{1}{n_y k_y} c_x s_y \\ \frac{1}{n_x k_x} s_x c_y \end{pmatrix}, \begin{pmatrix} \frac{1}{n_y k_y} c_x s_y \\ \frac{1}{n_x k_x} s_x c_y \end{pmatrix}$ $\begin{pmatrix} \frac{1}{n_y k_y} s_x s_y \\ -\frac{1}{n_x k_x} c_x c_y \end{pmatrix}, \begin{pmatrix} \frac{1}{n_y k_y} c_x c_y \\ -\frac{1}{n_x k_x} s_x s_y \end{pmatrix}$

TABLE 4.2. Table showing (unnormalized) basis elements for the transverse subspace $T(\mathbf{n})$. For notational convenience, the basis elements from Table 4.1 are abbreviated: c_x means $\cos(n_x k_x x)$ and c_y means $\cos(n_y k_y y)$. The definitions for the sine elements follow the same convention.

called *Longitudinal*.

Transverse Branch Transverse modes are combinations of modulations of the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ into a rotational (divergence-free) vector field. For a given node definition, $\mathbf{n}$, denote the measured modes in the associated transverse branch by $T(\mathbf{n})$. It follows that the two-dimensional curl of A

$$\nabla \wedge A := \begin{pmatrix} \partial_y A \\ -\partial_x A \end{pmatrix}, \tag{4.3.4}$$

is contained in the subspace $T(\mathbf{n})$. In Equation (4.3.4), $\partial_x := \partial/\partial x$ and $\partial_y := \partial/\partial y$ are partial differentiation with respect to the x and y coordinates. Basis elements for the $T(\mathbf{n})$ subspace can be obtained by finding functions whose curl returns the basis elements from Table 4.1 (see Equation (4.3.4)). These basis elements (without normalization) are shown in Table 4.2.

Longitudinal Branch The longitudinal branch consists of two separate subspaces, each of dimension four (again, two if either n_x or $n_y = 0$). The longitudinal modes, which span the subspace denoted $L(\mathbf{n})$, are irrotational vector fields obtained by combining modulations

n	Basis for $L(\mathbf{n})$	Basis for $P(\mathbf{n})$
n	Basis for $L(\mathbf{n})$	Basis for $P(\mathbf{n})$
$(n_x, 0)$	$\begin{pmatrix} c_x \\ 0 \end{pmatrix}, \begin{pmatrix} s_x \\ 0 \end{pmatrix}$	$\begin{pmatrix} p_x \\ p_y \end{pmatrix} c_x, \begin{pmatrix} p_x \\ p_y \end{pmatrix} s_x$
$(0, n_y)$	$\begin{pmatrix} 0 \\ c_y \end{pmatrix}, \begin{pmatrix} 0 \\ s_y \end{pmatrix}$	$\begin{pmatrix} p_x \\ p_y \end{pmatrix} c_y, \begin{pmatrix} p_x \\ p_y \end{pmatrix} s_y$
(n_x, n_y)	$\begin{pmatrix} \frac{1}{n_x k_x} c_x s_y \\ -\frac{1}{n_y k_y} s_x c_y \end{pmatrix}, \begin{pmatrix} \frac{1}{n_x k_x} c_x s_y \\ -\frac{1}{n_y k_y} s_x c_y \end{pmatrix}$	$\begin{pmatrix} p_x \\ p_y \end{pmatrix} c_x c_y, \begin{pmatrix} p_x \\ p_y \end{pmatrix} s_x c_y$
	$\begin{pmatrix} \frac{1}{n_x k_x} s_x s_y \\ -\frac{1}{n_y k_y} c_x c_y \end{pmatrix}, \begin{pmatrix} \frac{1}{n_x k_x} c_x c_y \\ -\frac{1}{n_y k_y} s_x s_y \end{pmatrix}$	$\begin{pmatrix} p_x \\ p_y \end{pmatrix} c_x s_y, \begin{pmatrix} p_x \\ p_y \end{pmatrix} s_x s_y$

TABLE 4.3. Table showing (unnormalized) basis elements for the longitudinal subspaces $L(\mathbf{n})$ and $P(\mathbf{n})$. For notational convenience, the basis elements from Table 4.1 are again abbreviated as in Table 4.2.

by A of the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$. This vector field can be written as the gradient of A

$$\nabla A := \begin{pmatrix} \partial_x A \\ \partial_y A \end{pmatrix}.$$

These modes are paired together with modes from the four(two)-dimensional subspace denoted $P(\mathbf{n})$, which are known simply as *P-modes*. P-modes are obtained by combining modulations by A of the vector $\mathbf{e}_6$. Measured modes from longitudinal branches are composed of elements with non-zero projection on both the $L(\mathbf{n})$ and $P(\mathbf{n})$ subspaces [21]. For this reason, the subspace $LP(\mathbf{n}) := L(\mathbf{n}) \oplus P(\mathbf{n})$ is referenced instead of the individual subspaces. Unlike modes from $T(\mathbf{n})$ or $L(\mathbf{n})$, P-modes are dependent upon the values of the momentum state $\mathbf{P}$ at each time instance. Basis elements for the L and P subspaces are shown in Table 4.3. Because of the pairing of the $L(\mathbf{n})$ and $P(\mathbf{n})$ subspaces, and the dependence of P-modes on each particle's momentum, observing structure in the LP modes is difficult, however methods to observe the periodic behavior do exist [21, 67, 68].

The Lyapunov modes associated with the degenerate exponents near the center index are oftentimes referred to as *hydrodynamic Lyapunov modes* because of their connection to the hydrodynamic modes in fluid dynamics that arise from conservation of mass, momentum and energy in particle-particle interactions [13, 61, 62]. As demonstrated above, the hydrodynamic Lyapunov modes can be thought of as consequences of the continuous symmetries in Equation (4.1.3), which are left invariant under collision.

4.4 Covariant Lyapunov Vectors

In general, the structure observed in the Lyapunov vectors output from Benettin's algorithm are insufficient for evaluating the structure of the tangent space decomposition along typical trajectories. To obtain this information, Ginelli's method for computing covariant Lyapunov vectors is used (see Section 2.3.2). The procedure for computing the covariant Lyapunov vectors of hard disk systems is computationally intensive, therefore systems of small particle number N are considered [8, 69, 68]. For the analysis of this section, the case where $N = 100$ in a square box ($A = 1$) with density $\rho = 0.7$ is considered. This number of particles is sufficient to observe a single degeneracy (belonging to the transverse branch $T(1,1)$) near zero in the Lyapunov spectrum (see Figure 4.8). This section compares the quantitative measures addressed in the previous sections for the orthonormal Lyapunov vectors from the Modified Gram-Schmidt process to the vectors computed using Ginelli's method.

The discussion begins with the Lyapunov exponents approximated using both methods; Figure 4.8 shows this comparison. The symmetry about the center index ($i = 200$) is seen with one important difference. Due to the fact that Ginelli's method performs a backward iteration (see Algorithm 4), the sign of the Lyapunov exponents is reversed from what is computed using Benettin's algorithm on the forward iterates.

4.4.1 Localization and Mode Classification

A natural question to ask is whether or not the properties observed for orthogonal Lyapunov vectors (e.g. localization of perturbation vectors and the parallelism of position and momentum components of perturbation vectors with near-zero exponents) are again observed.

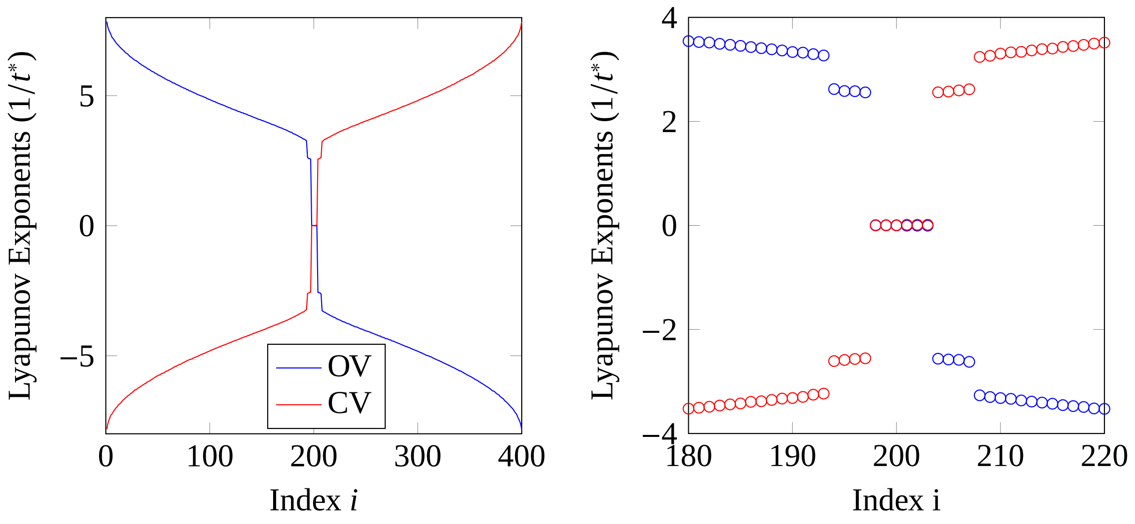


FIGURE 4.8. Plots of the Lyapunov spectrum of a hard disk system with $N = 100$, $A = 1$ and $\rho = 0.7$ using Benettin (OV) and Ginelli (CV) Methods. The results are obtained from simulation run durations of $500t^*$ time units forward (backward) for the Benettin (Ginelli) algorithms. Left: The full Lyapunov spectrum shows agreement in the two methods' calculation of the exponents (up to sign-reversal). The sign-reversal symmetry is a consequence of the backward evolution required for computing the covariant Lyapunov vectors. Right: The zoomed-in plot of the spectrum showing the single step near the center index $j = 2N$.

The answer to this question can be drawn from the output of Figure 4.9. The results for the localization $\mathcal{W}^{(i)}$ show that the covariant Lyapunov vectors are more localized. This means that, corresponding to exponents from the non-degenerate (continuous) portion of the spectrum in Figure 4.8, on average fewer particles contribute to the vector norm of each CV than to the orthogonal Lyapunov vector with the same index. The covariant Lyapunov vectors do not have to satisfy an orthogonality constraint, therefore this increased localization is expected [8]. Again, due to the symplectic nature of the dynamics, the symmetry in the localization width $\mathcal{W}^{(i)}$ about the center index $j = 2N$ is observed for both orthogonal and covariant Lyapunov vectors. In both cases, the vectors corresponding to the most unstable (stable) directions in forward-time are highly localized due to the fact that binary collisions are the source of the instability of typical trajectories. Perhaps most interesting is the monotonicity of the localization width of the covariant Lyapunov vectors up to the center index. This indicates that, in forward-time, the system has memory of past collisions, and that this memory is longer in time for smaller positive exponents.

As demonstrated in the right-half of Figure 4.9, the covariant Lyapunov vectors corresponding to exponents near-zero no longer have the property that the position and momentum components are parallel. Therefore, visualization of covariant Lyapunov vectors as vector fields over the simulation box does not capture the same periodic structure as seen in the orthogonal Lyapunov vectors (recall Figure 4.7) [8]. A reasonable question to ask is whether or not hydrodynamic Lyapunov modes can be observed in the covariant vectors [101]. The localization observed in the left-half of Figure 4.9 seems to indicate that hydrodynamic Lyapunov modes are present, as there are vectors that are left invariant under collision. Though they will not be discussed here, methods exist for explaining the structure, and observing the spatial periodicity, of covariant Lyapunov vectors corresponding to near-zero exponents (see [8, 68, 69]). The main point to emphasize here is that the constraint of orthogonality present in orthogonal Lyapunov vectors actually makes visualizing periodic structure in hydrodynamic Lyapunov modes easier than for covariant Lyapunov vectors.

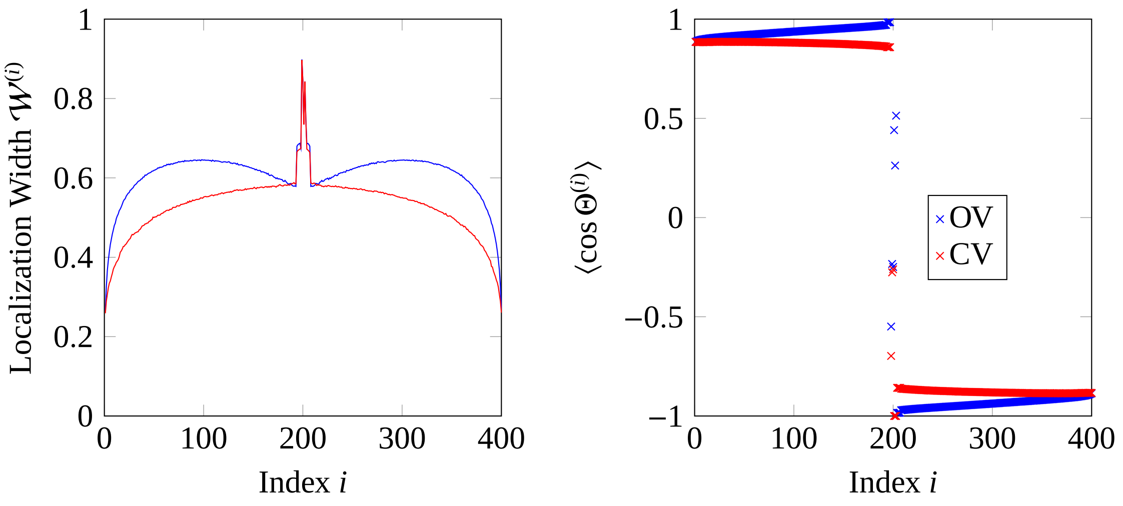


FIGURE 4.9. Plots of quantitative measures for localization ($\mathcal{W}^{(i)}$ defined in Equation (4.2.3)) and tangency of position and momentum components ($\langle \cos \Theta^{(i)} \rangle$ defined in Equation (4.3.1)) of perturbation vectors for the hard disk system with $N = 100$, $A = 1$ and $\rho = 0.7$ using the methods of Benettin (OV) and Ginelli (CV). The results are obtained from computing averages over $500t^*$ time units forward (backward) for the Benettin (Ginelli) algorithms. Left: The localization width as a function of index i, $\mathcal{W}^{(i)}$, shows that the covariant Lyapunov vectors are more highly localized than the orthogonal Lyapunov vectors. Right: For vectors corresponding to degenerate exponents near the center index, the position perturbation $\delta\mathbf{Q}$ and momentum perturbation $\delta\mathbf{P}$ are nearly (anti-)parallel for orthogonal Lyapunov vectors but not for covariant Lyapunov vectors.

4.4.2 Hyperbolicity

As discussed in Section 2.3.2, the orthogonal Lyapunov vectors that are measured do not, in general, provide a detailed description of the Oseledec splitting of the tangent space at each point along typical trajectories. Using Ginelli's method (see Algorithm 4) [36, 37], the structure of the Oseledec subspaces is revealed through the computed covariant Lyapunov vectors. Knowledge of this structure provides great insight into the *hyperbolicity* of hard disk systems. In the context of dynamical systems, hyperbolicity means that the unstable and stable subspaces of the system at an arbitrary point in phase space are separated by an angle bounded away from zero, which thus indicates that the stable and unstable manifolds are nowhere tangent.

For continuity of subsequent arguments, a quick redefinition is most useful. Redefine the matrix $\mathbf{W}_k$, which is the matrix containing as its columns the covariant Lyapunov vectors computed at timestep $n + \omega k$, from Equation (2.3.14) as $\mathbf{W}(\mathbf{\Gamma}_{n+\omega k})$. Let the covariant vectors $\mathbf{w}_j(\mathbf{\Gamma})$, where $1 \leq j \leq 4N$, be the columns of the matrix $\mathbf{W}(\mathbf{\Gamma})$. For hard-disk systems, let the unstable and stable subspaces at a point $\mathbf{\Gamma}$ along a typical trajectory be defined as $E^{(u)}(\mathbf{\Gamma})$ and $E^{(s)}(\mathbf{\Gamma})$. These spaces can be written in terms of the $\mathbf{w}_j$,

$$\begin{aligned} E^{(u)}(\mathbf{\Gamma}) &= \mathbf{w}_1 \oplus \mathbf{w}_2 \oplus \cdots \oplus \mathbf{w}_{2N-3}(\mathbf{\Gamma}) \\ E^{(s)}(\mathbf{\Gamma}) &= \mathbf{w}_{2N+4} \oplus \mathbf{w}_2 \oplus \cdots \oplus \mathbf{w}_{4N}(\mathbf{\Gamma}). \end{aligned} \tag{4.4.1}$$

The full Oseledec splitting from Equation (1.2.12) can now be written as the direct sum of the unstable, stable and neutral subspaces

$$\mathbb{R}^{4N} = E^{(u)}(\mathbf{\Gamma}) \oplus \mathcal{N}(\mathbf{\Gamma}) \oplus E^{(s)}(\mathbf{\Gamma}). \tag{4.4.2}$$

Equipped with the covariant splitting provided by the matrix $\mathbf{W}(\mathbf{\Gamma})$, the question of whether or not hard-disk systems are hyperbolic can now be addressed. To compute the minimum angle Ψ between vectors spanning $E^{(u)}$ and $E^{(s)}$, the law of cosines can be used:

$$\begin{aligned} \mathbf{M}(\mathbf{\Gamma}) &:= (E^{(u)}(\mathbf{\Gamma}))^\top E^{(s)}(\mathbf{\Gamma}), \\ \Psi &= \min\Big(\arccos(|\mathbf{M}(\mathbf{\Gamma})|)\Big), \end{aligned} \tag{4.4.3}$$

where min(·) and $|\cdot|$ in Equation (4.4.3) are the minimum over all elements and the absolute value of the argument matrix, respectively. The results for a hard-disk system with $N = 100$, $A = 1$ and $\rho = 0.7$ are shown in Figure 4.10. The rationale in Equation (4.4.3) can be used for any two subspaces. In addition to $E^{(u)}$ and $E^{(s)}$, the subspaces $E^{(T)}$ and $E^{(T*)}$ corresponding to the step observed in the Lyapunov spectrum (see Figure 4.8) can be shown to be separated in angle as well:

$$\begin{aligned} E^{(T)}(\mathbf{\Gamma}) &:= \mathbf{w}_{2N-6} \oplus \mathbf{w}_{2N-4} \oplus \mathbf{w}_{2N-5} \oplus \mathbf{w}_{2N-3}(\mathbf{\Gamma}) \\ E^{(T*)}(\mathbf{\Gamma}) &:= \mathbf{w}_{2N+4} \oplus \mathbf{w}_{2N+5} \oplus \mathbf{w}_{2N+6} \oplus \mathbf{w}_{2N+7}(\mathbf{\Gamma}). \end{aligned} \tag{4.4.4}$$

The separation in angle (represented by Φ) is shown in Figure 4.10. The results show that along typical trajectories, the angle Ψ separating the unstable and stable subspaces is bounded away from zero, and therefore the system appears to be *partially* hyperbolic [6]; total hyperbolicity cannot be claimed due to the existence of the center subspace $\mathcal{N}$.

4.5 Further Reading

The study of the dynamical properties of hard-disk systems and billiards has been a very active research area for several decades, due to numerous applications to physical systems. Throughout this time, many nice theoretical results have been developed. For nice treatments, see Gaspard [35] or Szász [88]. The genesis of much of this research came from the studies of Sinai *et. al* [10, 83, 84] and the application by many of dynamical systems methods to the Lorentz gas [11, 16, 88].

The questions relating to the theoretical origin of the structure seen in the Lyapunov spectrum (i.e. the degenerate exponents near the center index etc...) remain largely unanswered. Attempts have been made to show the connections of the Lyapunov exponents to random matrices [22, 43, 44, 90], periodic orbit expansion [89] and kinetic theory [61, 62, 68, 69], however many open questions remain. Recently, attempts have been made to qualitatively understand the time evolution of both the orthogonal and covariant Lyapunov vectors [67, 68, 69]. Studies have also been conducted showing the effect of

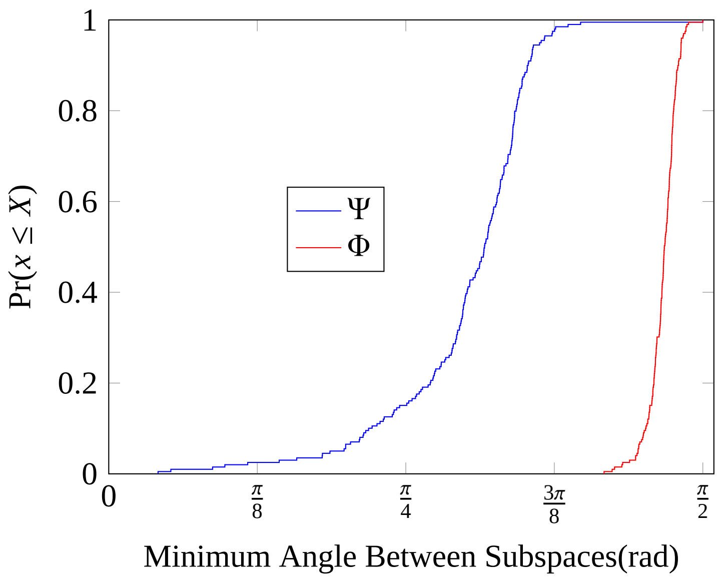


FIGURE 4.10. Plot of the empirical cumulative distribution function (eCDF) of angle separations for stable and unstable subspaces for the hard disk system with $N = 100$, $A = 1$ and $\rho = 0.7$. The results are obtained from simulation runs executed over $500t^*$ time units. The angle Ψ separating the spaces $E^{(u)}$ and $E^{(s)}$ is shown in blue, while the angle Φ separating the spaces $E^{(T)}$ and $E^{(T*)}$ is shown in red. The data for Ψ shows that the angle separating the stable and unstable subspaces along trajectories is bounded away from zero almost surely.

different boundary conditions on the study of the structure of the hydrodynamic Lyapunov modes [21].

Since the construction of covariant vectors has become computationally feasible due to the work of Ginelli *et. al* [36, 37], many new possibilities regarding a quantitative assessment of the hyperbolicity of large-dimensional dynamical systems exist and have been explored [8, 68]. Whether or not the systems can definitively be classified as (partially) hyperbolic remains an open question.

Chapter 5
Dynamical Properties of Generalized Collisions

The dynamical properties of particle systems interacting via generalized collisions, as derived in Chapter 3, are discussed in this chapter. This chapter moves in parallel with Chapter 4, except that the focus is now the new, generalized collision rule. Recall Equations (3.1.6) and (3.3.23):

$$
\begin{aligned}
\mathbf{Q}_f &= \mathbf{Q}_i,\\
\mathbf{p}_f^l &= \mathbf{p}_i^l, \quad l \neq j,k,\\
\mathbf{p}_f^j &= \beta \mathbf{p}_i^j + (1-\beta)\mathbf{p}_i^k + \alpha d\mathbf{p},\\
\mathbf{p}_f^k &= (1-\beta)\mathbf{p}_i^j + \beta \mathbf{p}_i^k - \alpha d\mathbf{p},\\
\delta\mathbf{q}_f^l &= \delta\mathbf{q}_i^l, \quad l \neq j,k,\\
\delta\mathbf{p}_f^l &= \delta\mathbf{p}_i^l, \quad l \neq j,k,\\
\delta\mathbf{q}_f^j &= \delta\mathbf{q}_i^j + \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{q}_f^k &= \delta\mathbf{q}_i^k - \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{p}_f^j &= \beta\delta\mathbf{p}_i^j + (1-\beta)\delta\mathbf{p}_i^k + \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&\quad + \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c - \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},\\
\delta\mathbf{p}_f^k &= (1-\beta)\delta\mathbf{p}_i^j + \beta\delta\mathbf{p}_i^k - \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&\quad - \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c + \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},\\
\kappa &:= \frac{2\beta(\beta-1)}{\sigma^2(b_\alpha + 2a_\alpha\alpha)}
\end{aligned}
$$

The goal of the construction of the collision rule in Equation (3.1.6) is to reduce the chaos, as quantified by the Lyapunov exponents and corresponding quantities such as the Kolmogorov-Sinai entropy h_{KS}, observed in the system . The origin of chaos in systems of impulsively interacting particles is the convexity of the particles; due to small changes in particle geometry at collision, trajectories diverge exponentially over time. The motivation for the construction of this rule is to determine if, by taking weighted averages of the colliding particles' momenta at collision (weighted by a single term β), the impulsive force applied along the collision center-of-mass vector (see Figure 3.1) could be reduced. It is this impulsive force, which is dependent upon the unit vector connecting the colliding particles' centers, that is the origin of the dynamic instability of trajectories. Figure 5.1 demonstrates that, for increasing β, the average value of the collisional parameter α is monotonically increasing over the interval $[0, 0.94]$. This addresses the notion that decreasing the weight β applied to the own particle's momentum at collision decreases the impulsive force applied along the collision center-of-mass vector. Now, the question of whether or not reducing $\langle\alpha\rangle$ actually reduces the chaotic properties of colliding disk systems can be addressed.

All technical details are the same as discussed in Chapter 4, with one important exception. For elastic collisions, there were three free parameters: N, A and ρ. For generalized collisions, β is an additional free parameter. The simulation unit for time is again $t^* = (mN\sigma^2/K)^{1/2}$, and all of the following are normalized to unity: mass m, diameter σ and the average kinetic energy per particle. The particle positions are again chosen to lie initially on a regular triangular lattice. The momenta are drawn from a zero-mean Gaussian distribution with standard deviation equal to $(K/N)^{1/2} = 1$. The momenta are then translated so that the center-of-mass of the particle system has zero momentum and rescaled so that the total kinetic energy of the system is equal to N

5.1 Lyapunov Exponents

The first questions one can address regarding the dynamical instabilities involve coarse quantities such as the maximum Lyapunov exponent λ_1, the minimum positive exponent $\max(0, \lambda_{2N-3})$[1] and the Kolmogorov-Sinai entropy per particle h_{KS}/N. These measures for

[1]For small enough β, the number of positive and negative exponents is no longer the same; the number of negative exponents increases. For these β, λ_{2N-3} would be negative. For quantifying the character of the spectrum near the zero exponents for the case of these small β, the metric $\max(0, \lambda_{2N-3})$ is used.

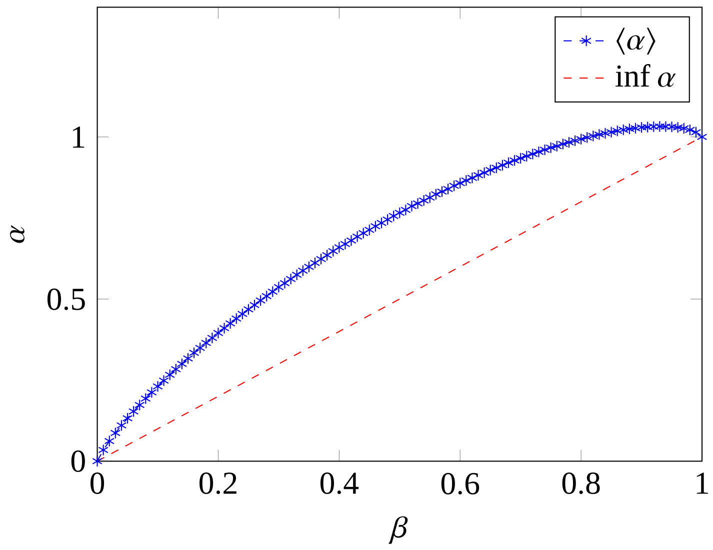


FIGURE 5.1. Plots showing the average of α for multiple β with $N = 256$, $A = 1$ and $\rho = 0.7$. The results are obtained from computing averages over $750t^*$ time units. The blue series indicates the mean of α, while the red series shows the greatest lower bound, or infimum, of α: $\inf \alpha = \beta$ (see Appendix A.2.2 for justification of this claim).

multiple β are shown in Figure 5.2. For lower densities, the maximal exponent λ_1 increases monotonically with β. This demonstrates a nice correspondence with the hypothesis of decreasing β meaning decreasing instability. Above a density threshold ρ_{pt}, the situation becomes more interesting. Similar to what was shown in Dellago *et. al* [16], at a density $0.5 < \rho_{pt} < 0.6$ a phase transition occurs resulting in a change in the character observed for both the maximal exponent and the KS entropy. For $\rho > \rho_{pt}$, decreasing β actually indicates an *increasing* measure of the chaos present; while the KS entropy shows increasing entropy production per particle for increasing β. Somewhat interestingly, the measure $\max(0, \lambda_{2N-3})$ provides a rough idea of what can be expected for the step in spectra near the center index $i = 2N$ corresponding to the zero exponents. For small ρ the magnitude of the first step is small and increases only slightly above some threshold $\beta_c^{(\rho)} \approx 0.15$. The step characteristic does not begin to appear until some larger threshold ($\beta > \beta_c^{(\rho)}$ above). Above this critical β, the change in the exponent λ_{2N-3} vs. β becomes larger for increasing β.

After looking at some comparisons of the narrower measures, a broader assessment

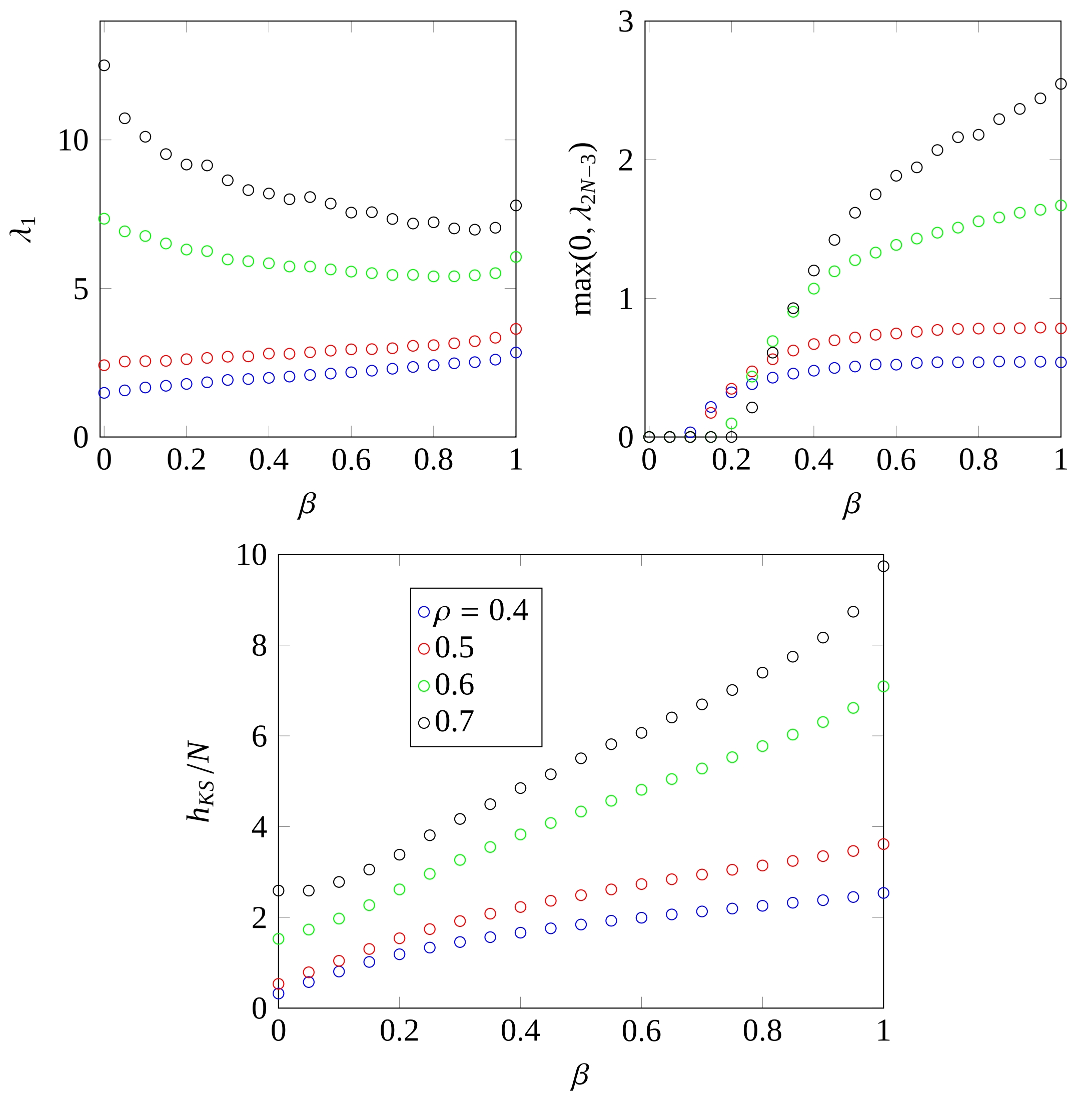


FIGURE 5.2. Plots showing λ_1, $\max(0, \lambda_{2N-3})$ and h_{KS}/N for multiple β with $N = 100$, $A = 1$ and $\rho = \{0.4, 0.5, 0.6, 0.7\}$. The results are obtained from computing averages over $900t^*$ time units forward. Top left: The maximum Lyapunov exponent vs. β and ρ. Top right: Plot of the minimum positive Lyapunov exponent $\max(0, \lambda_{2N-3})$ showing that for β small enough, there is no step in spectra near the center index $i = 2N$. Bottom: Kolmogorov-Sinai entropy shows evidence of a phase transition between $\rho = 0.5$ and 0.6.

of the instability of trajectories can now be addressed. In Figure 5.3, the full Lyapunov spectrum is shown for multiple β. The spectrum for $\beta = 0$ (velocity swap between particles, $\alpha = 0$ uniformly) shows that, due to the presence of $2N$ additional conserved quantities (the momenta), there are now $2N + 6$ zero exponents. The consequence of time-reversibility (recall Section 3.2), which is true only for $\beta = 0$ and 1, is that the Lyapunov spectrum is symmetric about the center index; volumes in phase space, on average, neither grow nor contract (see Figure 5.4). The figure shows the desired trend: for decreasing β, the values of the continuous portion of the spectrum (before the onset of steps) are reduced. The negative half of the spectrum is larger than its positive counterpart for each $\beta \neq 0, 1$. This is a natural consequence, given the structure of the inverse collision rule of Equation (3.2.4). For the reverse-time collision, the impulsive force applied along the collision center of mass is larger than in forward-time, therefore the observation of larger magnitude exponents in the negative branch is to be expected. As β increases, the appearance of the familiar steps in the spectrum near the center index $i = 2N$ become apparent (see the plot at right in Figure 5.3). Whether or not steps appear is fundamentally important to the observation of hydrodynamic Lyapunov modes. This will be discussed further in Section 5.2.2.

5.2 Properties of Orthogonal Vectors (Modes)

The spectra in Figure 5.3 indicate that properties of the orthogonal Lyapunov vectors for generalized collisions may be similar to those of elastic collisions. Due to the lower KS entropy, and the quantitatively different spectra observed for different β, it seems reasonable that the localization of the orthogonal Lyapunov vectors would be smaller for most indices, due to the smaller impact of individual collisions on determining the unstable directions. The results for localization will be discussed first.

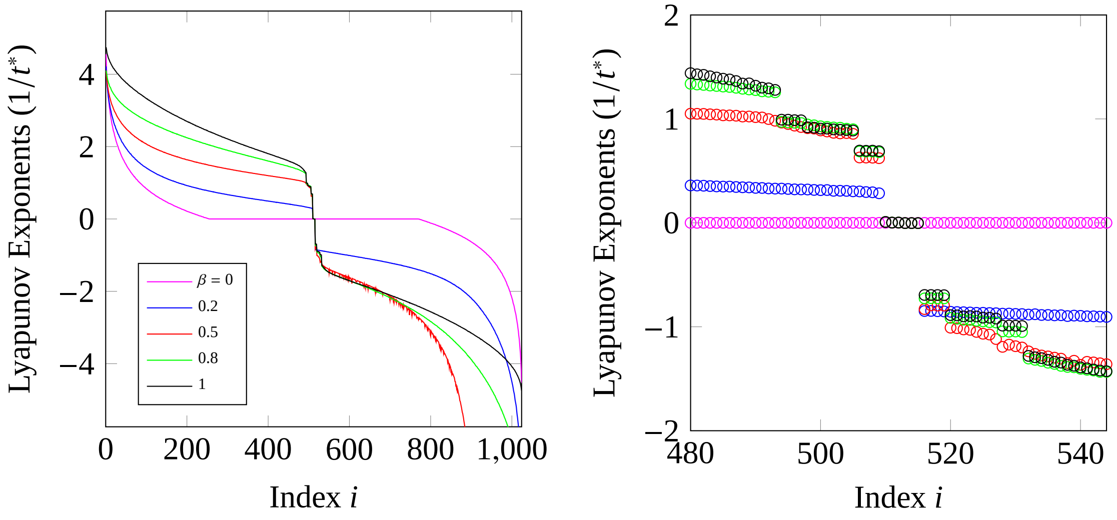


FIGURE 5.3. Plots showing the Lyapunov spectra for multiple β with $N = 256$, $A = 1$ and $\rho = 0.5$. The results are obtained from computing averages over $900t^*$ time units forward. Left: The full Lyapunov spectrum shows that for decreasing β the continuous portion of the positive (negative) Lyapunov spectra is smaller (larger) in magnitude. Right: Zoomed-in plot of the spectra at indices near the center index $i = 2N$ shows the removal of steps as β decreases.

5.2.1 Localization

Recall the localization measure $\mathcal{W}^{(i)}$ of Equation (4.2.3):

$$\mathcal{W}^{(i)} := \frac{1}{N}\exp[S^{(i)}].$$

This measure attains a maximum (of 1) when all particles contribute equally to the norm of the i^{th} orthogonal Lyapunov vector and a minimum when only a small number of particles contributes. The results for multiple β are shown in Figure 5.5. The most striking thing about this figure is the localization of orthogonal Lyapunov vectors associated with large negative exponents. Recall that at $\beta = 0.5$, the reverse-time collision rule is undefined, and therefore coefficients relating the forward-time β, α to the reverse-time counterparts are unbounded. This fact is reflected in the numerics; both in the localization width $\mathcal{W}^{(i)}$ and the (very large) most negative Lyapunov exponents (near $i = 4N$). The figure at top of Figure 5.6 shows the localization of the positive-half of the spectra. For $i > 50$, the

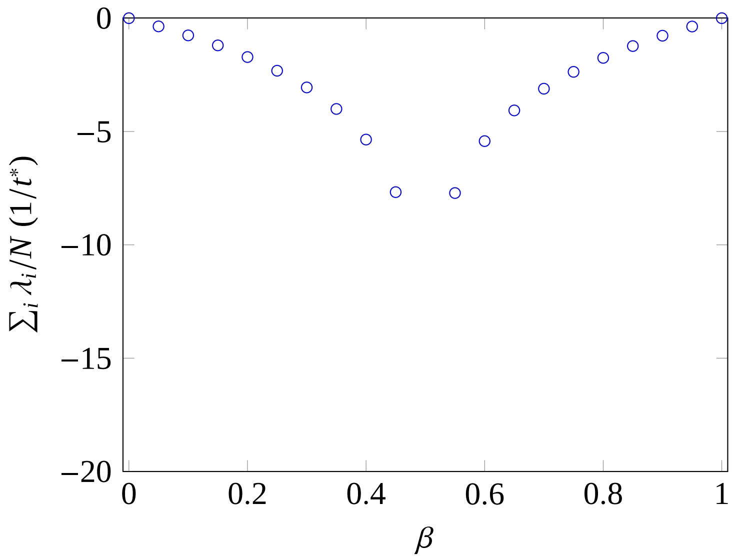


FIGURE 5.4. Plot showing the average per particle phase space contraction rate vs. β with $N = 100$, $A = 1$ and $\rho = 0.5$. The results are obtained from computing averages over $900t^*$ time units forward. Despite the asymmetry of the spectrum for $\beta \neq 0, 1$, the sum of all Lyapunov exponents is symmetric about $\beta = 0.5$, where the inverse collision rule is undefined.

localization of orthogonal Lyapunov vectors of index i decreases ($\mathcal{W}^{(i)}$ is larger). This indicates that the effect of individual collisions on these vectors decreases with decreasing β. For $i \leq 50$, due to the increased change (steeper slope) in λ_i vs. i, the effect of individual collisions is more pronounced for smaller β. The plot at bottom right of Figure 5.6 again indicates the presence of delocalized steps for large enough β.

5.2.2 Observing Coherent Structures

It is expected that hydrodynamic Lyapunov modes will be observed for large enough β due to the similarity of the localization width (see Figure 5.6) and the appearance of steps in the Lyapunov spectrum (see Figure 5.3) to the elastic case. But, what is the best way to quantify the observation of these hydrodynamic Lyapunov modes? A logical starting point would be to look at the modes themselves, as shown in Figure 5.7. This figure demonstrates the broader trend that, for decreasing β, the observation of modes near-zero become less

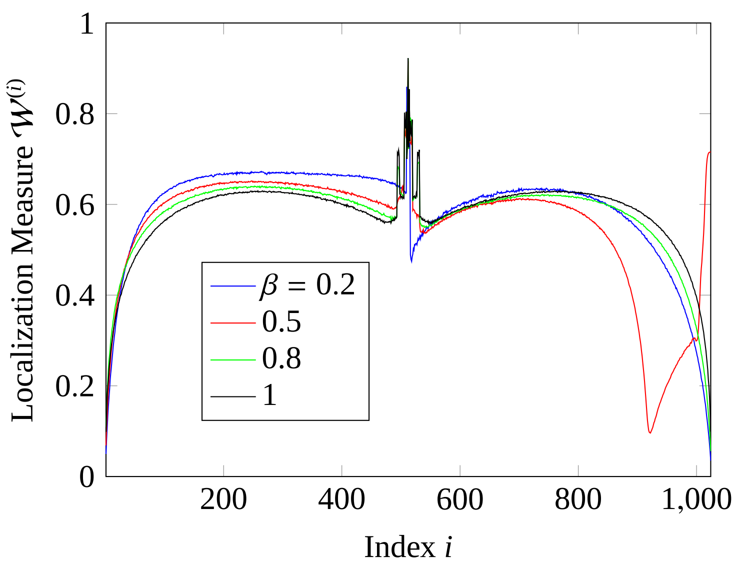


Figure 5.5. Plot showing the localization measure $\mathcal{W}^{(i)}$ for multiple β with $N = 256$, $A = 1$ and $\rho = 0.5$ for all indices $1 \leq i \leq 4N$. The results are obtained from computing averages over $900t^*$ time units forward. The localization of orthogonal Lyapunov vectors associated with negative exponents are, in general, more localized than the orthogonal Lyapunov vectors of conjugate-indexed exponents. (Recall: λ_i and λ_{4N-i+1} are conjugate-indexed)

structured. However, the viewing method chosen uses the vector field convention from Eckmann *et. al* [21], and developed in Section 4.3, which is only valid when the position and momentum components of orthogonal Lyapunov vectors near the center index are nearly (anti-)parallel. This property is not true, in general, for generalized collisions. Therefore, another method for identifying whether or not coherent structures exist is needed.

The structures that are sought from the near-zero vectors are believed to have a periodic character, as was discussed in Section 4.3. Therefore it makes sense that performing a spatial Fourier transform of the orthogonal Lyapunov vector data should reveal periodicity of the vectors with near-zero exponent. In fact, such an approach has been taken already with soft-disk potentials [31] with some success. A more fundamental observation concerning the presence of hydrodynamic Lyapunov modes follows Yang and Radons [100, 101]. This approach is concerned with the fluctuations of the *local* Lyapunov exponents $\bar{\lambda}_i(\mathbf{\Gamma}(t))$, which track the average growth (or contraction) of directions in phase space along typical

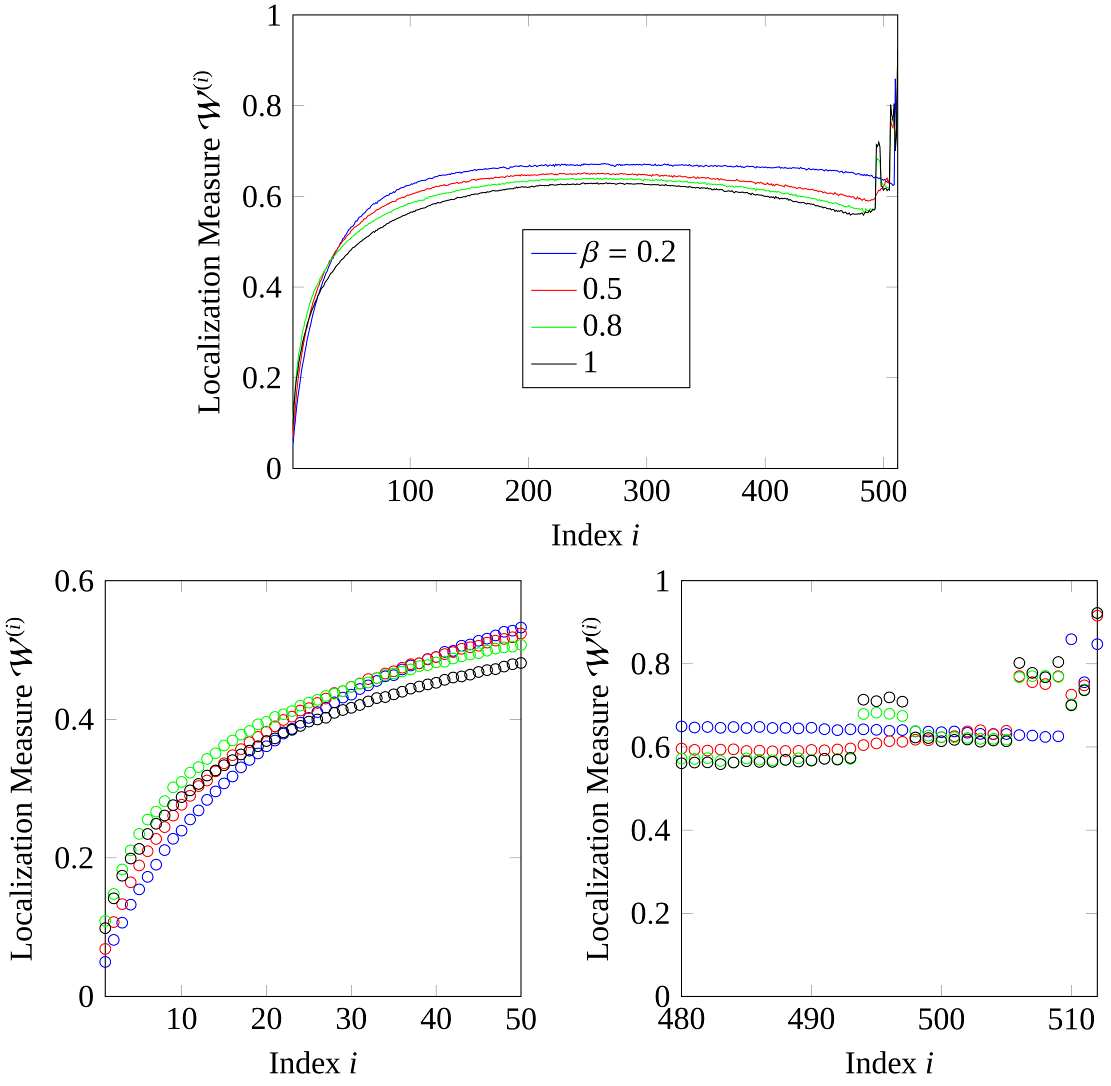


FIGURE 5.6. Plots showing the localization measure $\mathcal{W}^{(i)}$ for multiple β with $N = 256$, $A = 1$ and $\rho = 0.5$. The results are obtained from computing averages over $900t^*$ time units forward. Top: The localization measure for the positive spectra shows that Lyapunov vectors are more delocalized for smaller β. Bottom left: Zoomed-in plot of the localization measure for the positive spectra indicates that the largest exponents are more localized for smaller β. Bottom right: Zoomed-in plot near the center index $i = 2N$ shows that there is reason to suspect hydrodynamic Lyapunov modes for some β given the steps in the localization measure.

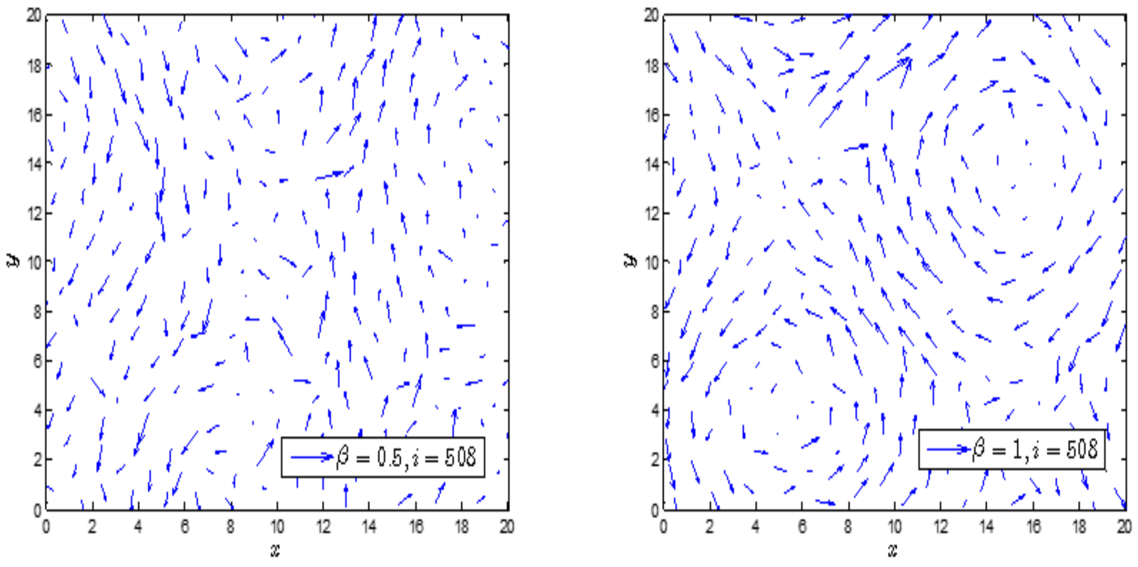


FIGURE 5.7. Plot showing the decay of hydrodynamic Lyapunov modes for small enough β with $N = 256$, $A = 1$ and $\rho = 0.5$. The results are obtained after simulating over $900t^*$ time units forward. Left: Although the hydrodynamic Lyapunov mode associated with the $T(1,1)$ transverse branch of the spectrum is present somewhat at $\beta = 0.5$, the structure is damaged. Right: The $T(1,1)$ branch for the elastic collision $\beta = 1$ case shows a near completely periodic character.

trajectories over short time periods. The local Lyapunov exponents can be computed as part of the usual Benettin scheme, however rather than taking the average over all time steps, only values over a smaller subset in time are considered[2]. Figure 5.8 shows this variation for $\beta = \{0.2, 0.5, 0.8, 1\}$. For smaller β the variation in exponents associated with steps causes the corresponding subspaces to be numerically unresolvable. Yang and Radons argue that the *domination* of the Oseledec splitting, meaning that the fluctuations observed for neighboring exponents are small relative to the difference between them, is necessary for the observation of structured hydrodynamic Lyapunov modes [100]. Applying this result to the problem at hand, a measure p_i^{split} is developed here to quantify the degree of numerical resolution of subspaces associated with adjacent Lyapunov exponents[3]:

$$\begin{aligned} p_i^{\text{split}} =& \Pr(p_i(t) > 1), \\ p_i(t) :=& \max_{|i-j|=1} \begin{cases} \bar{\lambda}_j(\mathbf{\Gamma}(t))/\bar{\lambda}_i(\mathbf{\Gamma}(t)) & \text{if } j > i \\ \bar{\lambda}_i(\mathbf{\Gamma}(t))/\bar{\lambda}_j(\mathbf{\Gamma}(t)) & \text{if } i > j \end{cases} \end{aligned} \tag{5.2.1}$$

The measure p_i^{split} is a means of quantifying the fraction of time that adjacent Lyapunov exponents are numerically indistinguishable. When p_i^{split} is small, it indicates that the subspace corresponding to index i is dynamically isolated from adjacent subspaces, and therefore the numerical procedure for computing Lyapunov exponents (i.e. Benettin's or Ginelli's algorithms) resolves the coherent structure. Large fluctuations in the local Lyapunov exponents near the center index $i = 2N$ correspond to a high probability of subspaces corresponding to adjacent exponents being numerically indistinguishable, as measured by p_i^{split} (see Figure 5.9). For increasing β, the probability of subspaces associated with near-zero exponents being numerically resolvable is very large, which provides evidence for coherent hydrodynamic Lyapunov modes for large enough β.

[2]The average is taken over a time on the order of the mean free time, t_{MFT}, which is the average time in between successive collisions for a single particle of arbitrary index.

[3]The integer i indexes non-degenerate exponents.

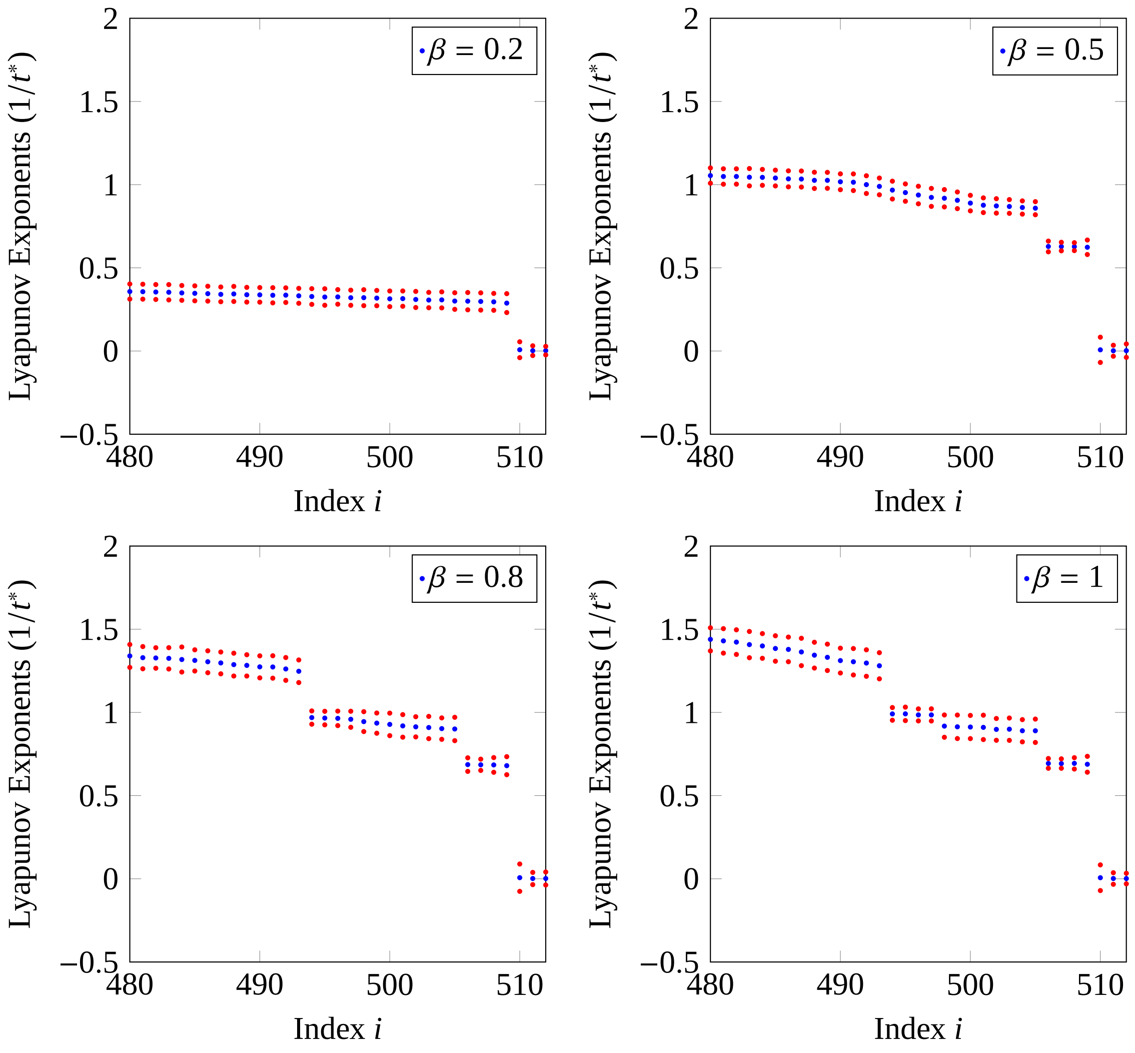


FIGURE 5.8. Plots showing the fluctuation in the local Lyapunov exponents for various β, $N = 256$, $A = 1$ and $\rho = 0.5$ for indices near the center index $i = 2N$. The results are obtained from data sampled over $900t^*$ time units. The blue markers indicate the long-time average of the local Lyapunov exponents at each index, while the red markers indicate one standard deviation about the long-time average. The inability to numerically distinguish subspaces corresponding to exponents near-zero becomes clearer for decreasing β.

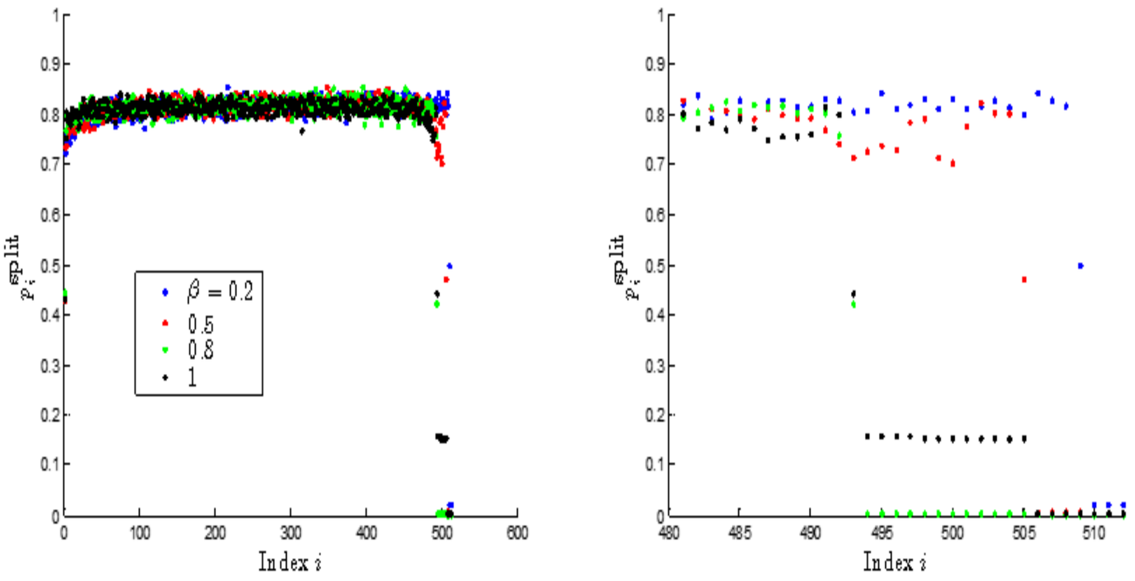


FIGURE 5.9. Plot showing the measure of the domination of the Oseledec splitting for multiple β with $N = 256$, $A = 1$ and $\rho = 0.5$. The results are obtained from computing averages over $900t^*$ time units forward. The splitting strength measure p_i^{split} for all indices $1 \leq i \leq 2N$ shows that fluctuations for near-zero exponents are small enough to observe good hydrodynamic Lyapunov modes for certain β.

5.3 Hyperbolicity

The notion of subspace distinction is most important when dealing with the issue of hyperbolicity. When fluctuations in local Lyapunov exponents are large, and the magnitude of the separation between the smallest positive and least negative exponents is small, distinguishing the stable and unstable subspaces from one another is difficult. To address hyperbolicity, the covariant vectors (CVs) computed using Ginelli's algorithm (see Algorithm 4) are used. Recall the definition of the unstable and stable subspaces at phase point $\mathbf{\Gamma}$ from Equation (4.4.1):

$$E^{(u)}(\mathbf{\Gamma}) = \mathbf{w}_1 \oplus \mathbf{w}_2 \oplus \cdots \oplus \mathbf{w}_{2N-3}(\mathbf{\Gamma})$$
$$E^{(s)}(\mathbf{\Gamma}) = \mathbf{w}_{2N+4} \oplus \mathbf{w}_2 \oplus \cdots \oplus \mathbf{w}_{4N}(\mathbf{\Gamma}),$$

where the $\mathbf{w}_i$ are the CVs. As β decreases, the probability of observing a near-zero minimum angle (recall Equation (4.4.3)) between the stable and unstable subspaces increases. Moreover, the average angle, and the variation about the mean angle, between $E^{(u)}$ and $E^{(s)}$ decreases with β (see Figure 5.10). This indicates a closer alignment between the unstable and stable manifolds for smaller β (i.e. a higher probability that the two subspaces cannot be differentiated). This close alignment is to be expected given the fluctuations present in the local Lyapunov exponents.

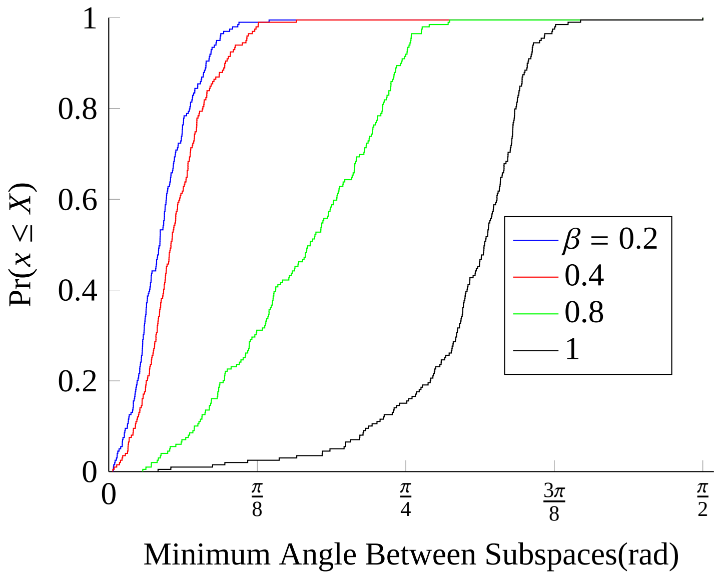


FIGURE 5.10. Plot showing hyperbolic properties of the generalized collision rule for multiple β with $N = 100$, $A = 1$ and $\rho = 0.7$. The results are obtained simulation runs of $500t^*$ time units backward. The minimum angle observed decreases with decreasing β.

Conclusions and Future Work

The novel contributions of this thesis were the construction of a new, generalized impulsive collision rule for studying the dynamics of multi-particle systems and the application, and development, of analytical methods for assessing qualitative features of the tangent space dynamics of these systems. These contributions are important because they help to advance the understanding of large degree-of-freedom systems and the understanding of the chaotic properties of such systems. The choice of these particle systems was deliberate. Because of exactness of the dynamics (i.e. there is no truncation error due to numerical integration and a closed-form solution for evaluating the time of each collision exists), hard disk systems provide a very illustrative example of chaos in non-smooth dynamical systems. Therefore, substantive statements regarding the character of chaos in non-smooth dynamical systems can be made when studying these systems.

This work was intended to provide a thorough theoretical background for evaluating properties, both quantitative and qualitative, of chaotic dynamical systems with many degrees-of-freedom. The material is interesting in its own right, however the motivation for the presentation was to provide the necessary tools for addressing quantitative (and qualitative) measures of chaos in these complex systems. Once established, these measures were used to answer the simple question that was the genesis of this work: Can a generalized impulsive collision rule be developed for particle systems that systemically reduces the chaos present?

The work of Chapter 5 demonstrated that the novel collision rule constructed in Chapter 3, for certain ρ and β, reduces measures of instability observed (Lyapunov exponents and Kolmogorov-Sinai entropy) in particle systems. These measures provided justification for the simple intuitive hypothesis that decreasing the average of the weight α (recall Equations (3.1.6) and (3.3.23)) applied to the impulsive force term applied along the collision center-of-mass vector should in fact *decrease* the disorder (chaos) observed. Along with these observations, nice numerical results followed that were used to bolster this intuition. Reducing the chaos was shown to come with a price: namely that decreasing β resulted in decreased numerical resolution of subspaces corresponding to adjacent Lyapunov exponents and decreased angular separation of stable and unstable subspaces (manifolds). Several interesting global properties emerged as well. As a consequence of time-irreversibility for $\beta \neq 0, 1$, the negative Lyapunov exponents were larger in magnitude than the conjugate-indexed positive exponents. No conjugate-pairing rule between these exponents was observed, however there was a rather interesting symmetry observed for the sum of all Lyapunov exponents, which quantifies the average growth (contraction) of phase space volumes co-moving along typical trajectories (see Figure 5.4). This symmetry was to be expected given the symmetric nature of the collision rule (namely the invariance in the form of the collision rule under interchange of particle index). The localization of Lyapunov vectors showed that for decreasing β the orthonormalization procedure preserved the effect of individual collisions longer, on average, as indicated by the increasing values of $\mathcal{W}^{(i)}$ observed in Figures 5.5 and 5.6. Although much was presented, many open questions remain.

The most pressing open questions have to deal with macroscopic quantities such as transport coefficients [18, 95], velocity-velocity autocorrelations [76] and position-velocity correlation measures [79, 80] for these generalized collisions. So far, all that has been addressed has to deal with microscopic measures of instability. No connection has yet been established between the measures of microscopic chaos (e.g. Lyapunov exponents) and the measures of macroscopic order; though some preliminary work in this direction has been done [14, 50]. The connection between the global structure observed in the orthogonal Lyapunov vectors computed using Benettin's algorithm and the emergence of order in the full nonlinear dynamics remains unclear. It is suspected that the timescale associated with the fastest growing subspaces is too short relative to that of the slowest growing subspaces for observing such coherent structures. Lega [59] demonstrated the emergence of collective structures in the full nonlinear dynamics of particle systems with a collision rule very close to the one developed in Chapter 3 with $\beta = 0.5$. The rule from Lega was dissipitative, and random tumbling was enabled to prevent relaxation to an equilibrium of zero energy. Given the observation of coherent structures in cases such as these, it is uncertain whether or not the Lyapunov exponents would indicate anything significant regarding the presence (or absence) of chaos in such systems. Additionally, from a purely statistical mechanics perspective, a Boltzmann-like equation (see Dorfman Chapter 2 [18]) can be derived for this new collision kernel.

Appendices

Appendix A
Justification for Important Results

A.1 The Multiplicative Ergodic Theorem

The goal of this appendix is to prove the Multiplicative Ergodic Theorem of Oseledec [70] (MET) first discussed in Chapter 1 Section 1.2.3. The proof expands upon the approach from Choe [12], which uses the Subadditive Ergodic Theorem (SET) of Kingman [54, 86]. The methods used in proving the SET do little to build intuition regarding the proof of the MET, therefore the interested reader is referred to external sources for its proof (e.g. Pitman [75] or Steele [86]). The strategy for the proof is to focus on geometric methods that provide motivation for the numerical algorithms presented in Chapter 2; the reader is referred to external sources for the technical details required from measure theory (e.g. Ruelle [77]).

The MET is stated and proved for transformations which are not necessarily invertible. This theorem is all that is required for proving the existence of Lyapunov exponents under very general circumstances for (nonlinear) differentiable maps which are iterated forward in time. The statement and proof of the theorem is done with discrete-time dynamics in mind. Recall from the arguments of Section 1.2.1 that arguments for discrete-time dynamics can be extended to continuous-time using the time-one map. Now, the statement and proof of the MET.

Theorem 2 (The Multiplicative Ergodic Theorem for Not-Necessarily Invertible Transformations [12, 70, 77])**.** Let $\mathbf{M}$ be an N-dimensional (not necessarily invertible) measure-preserving transformation on a probability space (X, μ). Let $A : X \to GL(N, \mathbb{R})$ be measurable with

$$\int_X \log^+(\|A\|)d\mu < \infty,$$
$$\log^+(\cdot) := \max(0, \log(\cdot)).$$

Require further that $A^{-1}(\mathbf{\Gamma})$ is almost surely (a.s.) invertible. Define

$$A_n(\mathbf{\Gamma}) := A(\mathbf{M}^{n-1}\mathbf{\Gamma}) \cdots A(\mathbf{M\Gamma})A(\mathbf{\Gamma}).$$

For $\mathbf{\Gamma} \in X$ there a.s. exist numbers (known as the *Lyapunov exponents*)

$$L^{(1)}(\mathbf{\Gamma}) > \cdots > L^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}), \tag{A.1.1}$$

and a sequence of subspaces (known as the *Oseledec filtration*)

$$\mathbb{R}^N = V_+^{(1)}(\mathbf{\Gamma}) \supsetneq \cdots \supsetneq V_+^{(r(\mathbf{\Gamma}))}(\mathbf{\Gamma}) \supsetneq V_+^{(r(\mathbf{\Gamma})+1)}(\mathbf{\Gamma}) = \{\emptyset\}, \tag{A.1.2}$$

such that

(i) $\lim_{n\to\infty} \frac{1}{n} \log \|A_n(\mathbf{\Gamma})\mathbf{v}\|$ exists and is equal to $L^{(i)}(\mathbf{\Gamma})$ when $\mathbf{v} \in V_+^{(i)}(\mathbf{\Gamma}) \setminus V_+^{(i+1)}(\mathbf{\Gamma})$.

(ii) $A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) = V_+^{(i)}(\mathbf{M\Gamma})$.

(iii) $r(\mathbf{\Gamma}), L^{(i)}(\mathbf{\Gamma})$ and $\dim V_+^{(i)}(\mathbf{\Gamma})$ are measurable.

(iv) $r(\mathbf{\Gamma}), L^{(i)}(\mathbf{\Gamma})$ and $\dim V_+^{(i)}(\mathbf{\Gamma})$ are constant along orbits of $\mathbf{M}$.

(v) $\lim_{n\to\infty} \log|\det A_n(\mathbf{\Gamma})| = \sum_{i=1}^{r(\mathbf{\Gamma})} L^{(i)}(\mathbf{\Gamma})(\dim V_+^{(i)}(\mathbf{\Gamma}) - \dim V_+^{(i+1)}(\mathbf{\Gamma}))$.

Proof. Crucial to the beginning of the proof is the following statement (modified only slightly from Section 1.2.3)

> For practical purposes, the growth rates of individual directions are not usually considered, but rather the growth rates of volumes of subspaces spanned by

perturbation vectors. Assume n is large, and let $\sigma_n^{(1)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(r)}(\mathbf{\Gamma})$ be the $r(\mathbf{\Gamma})$ distinct singular values of $A_n(\mathbf{\Gamma})$. Define

$$\phi_n^{(k)}(\mathbf{\Gamma}) := \log\Big(\sup_{\dim W=k}\Big|\det(A_n(\mathbf{\Gamma})|_W)\Big|\Big). \tag{A.1.3}$$

In (1.2.8), $\sup_{\dim W=k} |\det(A_n(\mathbf{\Gamma})|_W)|$ is interpreted as the supremum of volume elements of k-dimensional subspaces W spanned by columns of $A_n(\mathbf{\Gamma})$. The supremum is obtained when W is spanned by the eigenvectors of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ corresponding to the k largest singular values of $A_n(\mathbf{\Gamma})$ [12]. This step requires some justification. From Equation (1.2.6) in the last section 1.2.2, $A_n(\mathbf{\Gamma})$ has a singular value decompositionfrom which it follows that [1]

$$\begin{aligned}
|\det A_n(\mathbf{\Gamma})| &= |\det U(\mathbf{\Gamma}) \det \Sigma(\mathbf{\Gamma}) \det V(\mathbf{\Gamma})^\top| \\
&= |\det \Sigma(\mathbf{\Gamma})| \\
&= (\sigma_n^{(1)}(\mathbf{\Gamma}))^{m(\sigma_n^{(1)}(\mathbf{\Gamma}))} \cdots (\sigma_n^{(r)}(\mathbf{\Gamma}))^{m(\sigma_n^{(r)}(\mathbf{\Gamma}))}.
\end{aligned}$$

The multiplicity of each singular value, $\sigma_n^{(i)}(\mathbf{\Gamma})$, is represented by $m(\sigma_n^{(i)}(\mathbf{\Gamma}))$. The determinant of the restriction of $A_n(\mathbf{\Gamma})$ to k-dimensional subspaces is then the product of k singular values. This quantity is maximized when the space spanned by the columns of $V(\mathbf{\Gamma})$ associated with the k largest singular values of $A_n(\mathbf{\Gamma})$ is considered [2].

The k largest (distinct) singular values of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ are $\sigma_n^{(1)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(j)}(\mathbf{\Gamma})$, where j is chosen such that $\sum_{i=1}^{j} m(\sigma_n^{(i)}(\mathbf{\Gamma})) = k$. From the previous

[1]The singular values used in subsequent arguments are such that $\sigma_n^{(1)}(\mathbf{\Gamma}) > \sigma_n^{(2)}(\mathbf{\Gamma}) > \cdots > \sigma_n^{(r)}(\mathbf{\Gamma})$.

[2]Due to multiplicity, k can only take values from the restricted set

$$k \in \Big\{m(\sigma_n^{(1)}(\mathbf{\Gamma})), m(\sigma_n^{(1)}(\mathbf{\Gamma})) + m(\sigma_n^{(2)}(\mathbf{\Gamma})), \cdots, \sum_{i=1}^{r} m(\sigma_n^{(i)}(\mathbf{\Gamma}))\Big\}. \tag{A.1.4}$$

In other words, only volumes spanned by all eigenvectors of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ with degenerate eigenvalue are well-defined [21] arguments, $\phi_n^{(k)}(\mathbf{\Gamma})$ is reduced to (A.1.5)

$$\begin{aligned} \phi_n^{(k)}(\mathbf{\Gamma}) &= \log((\sigma_n^{(1)}(\mathbf{\Gamma}))^{m(\sigma_n^{(1)}(\mathbf{\Gamma}))} \cdots (\sigma_n^{(j)}(\mathbf{\Gamma}))^{m(\sigma_n^{(j)}(\mathbf{\Gamma}))}) \\ &= \sum_{i=1}^{j} m(\sigma_n^{(i)}(\mathbf{\Gamma})) \log \sigma_n^{(i)}(\mathbf{\Gamma}), \end{aligned} \tag{A.1.5}$$

It is desired that the average growth rate of $\phi_n^{(k)}(\mathbf{\Gamma})$ is well-defined as $n \to \infty$; that is,

$$\lim_{n\to\infty} \frac{1}{n} \phi_n^{(k)}(\mathbf{\Gamma}) = \phi_*^k(\mathbf{\Gamma}), \tag{A.1.6}$$

for some measurable $\phi_*^k(\mathbf{\Gamma})$.

The subspaces W from Equation (A.1.3) are intersections of the elements of the filtration from Equation (A.1.2). When computing the largest singular value of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$, $\sigma_n^(1)(\mathbf{\Gamma})$, $W = V_+^{(1)} \cap V_+^{(2)}$. When W is taken to be $V_+^{(1)} \cap V_+^{(j)}$, where $1 < j \leq r(\mathbf{\Gamma})$, the sum of the first j distinct singular values of $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ (weighted by their respective multiplicities) is obtained from Equation (A.1.3). In the long time limit as $n \to \infty$, the determinant in Equation (A.1.3) can become unbounded; therefore the average growth *rate* as quantified by the long time average in Equation (A.1.6) is of greater interest. The term $\phi_*^k(\mathbf{\Gamma})$, if it can be proven to exist, would determine the sum of the first k Lyapunov exponents of the map $\mathbf{M}$ from Equation (A.1.1). Proving this fact requires a corollary to Kingman's Subadditive Ergodic Theorem [54, 86].

Theorem 3 (Subadditive Ergodic Theorem [54])**.** Let $\mathbf{M}$ be a measure-preserving transformation on a probability space (X, μ). For $n \geq 1$ suppose that $g_n : X \to \mathbb{R}$ is a measurable function such that for all m and $n \geq 1$.

$$\int_X (g_1)^+ d\mu := \int_X \max(0, g_1) d\mu < \infty \tag{A.1.7}$$

$$g_{m+n} \leq g_m + g_n \circ T^m \ \mu\text{-a.e.} \tag{A.1.8}$$

Then there exists a measurable function $g : X \to \mathbb{R} \cup \{-\infty\}$ such that

(a) $g \circ T = g$ (T-invariance)

(b) $\int_X g^+ d\mu < \infty$

(c) $\lim_{n\to\infty} g_n = g$ μ-a.e.

Proof. Proofs of this theorem use elementary methods of analysis and probability theory. An easy-to-follow proof based on Steele [86] can be found in the very illustrative lectures of Pitman [75]. □

A corollary concerning the existence of the limit in Equation (A.1.6) immediately follows from the SET.

Corollary 4. Let (X, μ) be a probability space, $\mathbf{M}$ an N-dimensional measure-preserving transformation on X and $A : X \to GL(N, \mathbb{R})$ a measurable transformation. For some $\mathbf{\Gamma} \in X$, let $\phi_n^{(k)}(\mathbf{\Gamma})$ be as defined in Equation (A.1.6). The following statements are true:

(a) $\phi_n^{(k)}(\mathbf{\Gamma})$ is measurable for all k, n.

(b) $\int_X (\phi_1^{(k)}(\mathbf{\Gamma}))^+ d\mu < \infty$ for all k.

(c) The sequence $\{\phi_n^{(k)}(\mathbf{\Gamma})\}_{1\le k\le r(\mathbf{\Gamma})}$ is subadditive.

Therefore, a measurable function $\phi_*^k(\mathbf{\Gamma})$ exists such that $\lim_{n\to\infty} \frac{1}{n}\phi_n^{(k)}(\mathbf{\Gamma}) = \phi_*^k(\mathbf{\Gamma})$.

Proof. For (a), it is assumed that A is measurable on X, which means that entries of A are themselves measurable functions on X. For any n, the entries of $A_n(\mathbf{\Gamma})$ will be countable sums of countable products of measurable functions on X, so these entries are measurable as well [29]. This implies that $|\det A_n(\mathbf{\Gamma})|$ and its restriction $\left|\det A_n(\mathbf{\Gamma})|_W\right|$ are measurable on X, since the determinant is a countable set of operations acting on measurable functions on X. Therefore the suprema of $\left|\det A_n(\mathbf{\Gamma})|_W\right|$ over k-dimensional subspaces W are also measurable. It follows that $\phi_n^{(k)}$ is measurable for all n, k.

For (b), recall that the $\{\sigma_1^{(i)}(\mathbf{\Gamma})\}$, $1 \leq i \leq r(\mathbf{\Gamma})$, are the singular values of $A(\mathbf{\Gamma})$. From Equation (A.1.5), it follows that

$$\begin{aligned}
\phi_1^{(k)}(\mathbf{\Gamma}) &= \log((\sigma_1^{(1)}(\mathbf{\Gamma}))^{m(\sigma_1^{(1)}(\mathbf{\Gamma}))} \cdots (\sigma_1^{(j)})^{m(\sigma_1^{(j)}(\mathbf{\Gamma}))}) \\
&= \sum_{i=1}^{j} m(\sigma_1^{(i)}(\mathbf{\Gamma})) \log \sigma_1^{(i)}(\mathbf{\Gamma}) \\
&\leq k \log \sigma_1^{(1)}(\mathbf{\Gamma}).
\end{aligned}$$

From the assumptions in the statement of the theorem, it is true that

$$\begin{aligned}
\int_X (\phi_1^{(k)}(\mathbf{\Gamma}))^+ d\mu &\leq k \int_X \log^+ \sigma_1^{(1)}(\mathbf{\Gamma}) d\mu \\
&= k \int_X \log^+ \|A\| d\mu \\
&< \infty.
\end{aligned}$$

To show (c), recall that by definition $A_{l+n}(\mathbf{\Gamma}) := A_n(\mathbf{M}^l\mathbf{\Gamma})A_l(\mathbf{\Gamma})$, and so

$$\left|\det A_{l+n}(\mathbf{\Gamma})|_W\right| = \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_{A_l(\mathbf{\Gamma})W} \det A_l(\mathbf{\Gamma})|_W\right|. \tag{A.1.9}$$

It follows that

$$\begin{aligned}
\sup_{\dim W=k} \left|\det A_{l+n}(\mathbf{\Gamma})|_W\right| &= \sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_{A_l(\mathbf{\Gamma})W} \det A_l(\mathbf{\Gamma})|_W\right| \\
&= \sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_{A_l(\mathbf{\Gamma})W}\right| \sup_{\dim W=k} \left|\det A_l(\mathbf{\Gamma})|_W\right| \\
&\leq \sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_W\right| \sup_{\dim W=k} \left|\det A_l(\mathbf{\Gamma})|_W\right|,
\end{aligned}$$

since $\sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_{A_l(\mathbf{\Gamma})W}\right| \leq \sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_W\right|$. Taking the log,

$$\begin{aligned}
\phi_{l+n}^{(k)}(\mathbf{\Gamma}) &:= \log\Big(\sup_{\dim W=k} \left|\det A_{l+n}(\mathbf{\Gamma})|_W\right|\Big) \\
&\leq \log\Big(\sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_W\right| \sup_{\dim W=k} \left|\det A_l(\mathbf{\Gamma})|_W\right|\Big) \\
&= \log\Big(\sup_{\dim W=k} \left|\det A_l(\mathbf{\Gamma})|_W\right|\Big) + \log\Big(\sup_{\dim W=k} \left|\det A_n(\mathbf{M}^l\mathbf{\Gamma})|_W\right|\Big) \\
&= \phi_l^{(k)}(\mathbf{\Gamma}) + \phi_n^{(k)}(\mathbf{M}^l\mathbf{\Gamma}).
\end{aligned}$$

It is therefore true that $\phi^{(k)}_{l+n}(\mathbf{\Gamma}) \leq \phi^{(k)}_{l}(\mathbf{\Gamma}) + \phi^{(k)}_{n}(\mathbf{M}^l\mathbf{\Gamma})$, which proves subadditivity. Therefore, all conditions of the SET 3 are met and so a measurable $\phi^k_*(\mathbf{\Gamma})$ exists such that $\lim_{n\to\infty} \frac{1}{n}\phi^{(k)}_n(\mathbf{\Gamma}) = \phi^k_*(\mathbf{\Gamma})$. □

From Corollary 4, the following limit exists

$$\lim_{n\to\infty} \frac{1}{n}\phi^{(k)}_n(\mathbf{\Gamma}) \quad = \quad \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{k} \log \sigma^{(i)}_n(\mathbf{\Gamma}).$$

This can only be true if all of the terms converge; that is $\frac{1}{n}\log \sigma^{(i)}_n(\mathbf{\Gamma})$ converges for every i. Since $A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma})$ is symmetric, it is orthogonally diagonalizable:

$$A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma}) = C_n(\mathbf{\Gamma})D_n(\mathbf{\Gamma})C_n(\mathbf{\Gamma})^{-1},$$

where $C_n(\mathbf{\Gamma})$ is an orthogonal matrix and $D_n(\mathbf{\Gamma})$ is the diagonal matrix with entries $(\sigma^{(i)}_n(\mathbf{\Gamma}))^2$. Then for any n, it is true that

$$(A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma}))^{\frac{1}{2n}} = C_n(\mathbf{\Gamma})D_n(\mathbf{\Gamma})^{\frac{1}{2n}}C_n(\mathbf{\Gamma})^{-1}.$$

Since $\frac{1}{n}\log \sigma^{(i)}_n(\mathbf{\Gamma})$ converges as $n \to \infty$, $D_n(\mathbf{\Gamma})^{\frac{1}{2n}}$ and $(A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma}))^{\frac{1}{2n}}$ converge as well. Define

$$\mathbf{\Lambda}(\mathbf{\Gamma}) := \lim_{n\to\infty}(A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma}))^{\frac{1}{2n}}. \tag{A.1.10}$$

Let $\sigma^{(i)}(\mathbf{\Gamma})$ be the ith eigenvalue of $\Lambda(\mathbf{\Gamma})$, where $\sigma^{(i)}(\mathbf{\Gamma}) := \lim_{n\to\infty} \sigma^{(i)}_n(\mathbf{\Gamma})^{\frac{1}{n}}$, with corresponding eigenspace $U^{(i)}(\mathbf{\Gamma})$ [3]. There exists a sequence of integers $\{J_i(\mathbf{\Gamma})\}$

$$0 = J_{r(\mathbf{\Gamma})+1}(\mathbf{\Gamma}) < J_{r(\mathbf{\Gamma})}(\mathbf{\Gamma}) < \cdots < J_1 = N$$

such that

$$L^{(i)}(\mathbf{\Gamma}) := \log \sigma^{(i)}(\mathbf{\Gamma}) = \lim_{n\to\infty} \frac{1}{n} \log \sigma_{n,j}(\mathbf{\Gamma})$$

[3]Since $\mathbf{\Lambda}(\mathbf{\Gamma})$ is orthogonally-diagonalizable, it is true that

$$U(\mathbf{\Gamma})\Sigma(\mathbf{\Gamma})U^{-1}(\mathbf{\Gamma}),$$

where $U(\mathbf{\Gamma})$ and $\Sigma(\mathbf{\Gamma})$ are $N \times N$ orthonormal and diagonal matrices, respectively. The matrix $\Sigma(\mathbf{\Gamma})$ contains the eigenvalues $\sigma^{(j)}(\mathbf{\Gamma})$, $1 \leq j \leq r(\mathbf{\Gamma})$, of $\mathbf{\Lambda}(\mathbf{\Gamma})$. For each distinct eigenvalue $\sigma^{(j)}(\mathbf{\Gamma})$, the eigenspace $U^{(j)}(\mathbf{\Gamma})$ is simply the columns of $U(\mathbf{\Gamma})$ corresponding to $\sigma^{(j)}(\mathbf{\Gamma})$ in $\Sigma(\mathbf{\Gamma})$.

whenever $J_{i+1}(\mathbf{\Gamma}) < j \leq J_i(\mathbf{\Gamma})$, $1 \leq i \leq r(\mathbf{\Gamma})$ [4]. Observe that the sequence $\{J_i(\mathbf{\Gamma})\}$ is related to the multiplicity $m(\sigma^{(i)}(\mathbf{\Gamma}))$ according to

$$m(\sigma^{(i)}(\mathbf{\Gamma})) = J_i(\mathbf{\Gamma}) - J_{i+1}(\mathbf{\Gamma}).$$

Since $\mathbf{\Lambda}(\mathbf{\Gamma})$ is obtained from a limit, it is clearly $\mathbf{M}$-invariant and so the following is true:

1. $L^{(i)}(\mathbf{M\Gamma}) = L^{(i)}(\mathbf{\Gamma})$ a.s.

2. $r(\mathbf{M\Gamma}) = r(\mathbf{\Gamma})$ a.s.

This partially proves Part (iv).

To prove Part (i), let $V_+^{(r(\mathbf{\Gamma})+1)}(\mathbf{\Gamma}) := \{\emptyset\}$ and

$$V_+^{(i)}(\mathbf{\Gamma}) := U^{(1)}(\mathbf{\Gamma}) \oplus \cdots \oplus U^{(i)}(\mathbf{\Gamma}), \tag{A.1.11}$$

where $1 \leq i \leq r(\mathbf{\Gamma})$. Take $\mathbf{v} \in V_i(\mathbf{\Gamma}) \setminus V_{i+1}(\mathbf{\Gamma})$. Clearly $\mathbf{v}$ will have non-zero projection onto $U^{(j)}(\mathbf{\Gamma})$ for some $J_{i+1}(\mathbf{\Gamma}) < j \leq J_i(\mathbf{\Gamma})$; call this projection $\mathbf{u}$. The length of this projection grows exponentially, so for large enough n, the norm of $||A_n(\mathbf{\Gamma})\mathbf{v}||$ will be indistinguishable from $||A_n(\mathbf{\Gamma})\mathbf{u}||$. Fix $\varepsilon_1 > 0$ such that for all $n \geq N_1$

$$||A_n(\mathbf{\Gamma})\mathbf{u}|| - \varepsilon_1 < ||A_n(\mathbf{\Gamma})\mathbf{v}|| < ||A_n(\mathbf{\Gamma})\mathbf{u}|| + \varepsilon_1.$$

Similarly, fix $\varepsilon_2 > 0$ such that for all $n \geq N_2$

$$\sigma_i^n(\mathbf{\Gamma})||\mathbf{u}|| - \varepsilon_2 < ||A_n(\mathbf{\Gamma})\mathbf{u}|| < \sigma_i^n(\mathbf{\Gamma})||\mathbf{u}|| + \varepsilon_2.$$

Combining these two statements, and taking $\varepsilon := \varepsilon_1 + \varepsilon_2$ and $n \geq N := \max(N_1, N_2)$

$$\sigma_i^n(\mathbf{\Gamma})||\mathbf{u}|| - \varepsilon < ||A_n(\mathbf{\Gamma})\mathbf{v}|| < \sigma_i^n(\mathbf{\Gamma})||\mathbf{u}|| + \varepsilon.$$

This relationship must hold for all ε, so it can be concluded that for all $n \geq N$

$$\begin{aligned} \frac{1}{n}\log||A_n(\mathbf{\Gamma})\mathbf{v}|| &= \frac{1}{n}\log(\sigma_i(\mathbf{\Gamma})^n||\mathbf{u}||) \\ &= \log\sigma_i(\mathbf{\Gamma}) + \frac{1}{n}\log||\mathbf{u}||. \end{aligned}$$

[4] Recall the convention that subscripts denote quantities (potentially) repeated by multiplicity greater than one.

The second term in the right-hand side (RHS) $\frac{1}{n}\log\|\mathbf{u}\|$ goes to 0 in the limit, therefore

$$L^{(i)}(\mathbf{\Gamma}) := \lim_{n\to\infty}\frac{1}{n}\log\|A_n(\mathbf{\Gamma})\mathbf{v}\| = \log\sigma^{(i)}(\mathbf{\Gamma}).$$

This completes the proof of Part (i).

For Part (ii), containment must be shown both ways.

- ($\subseteq$) Define

  $$L_+(\mathbf{\Gamma},\mathbf{v}) := \lim_{n\to\infty}\frac{1}{n}\|A_n(\mathbf{\Gamma})\mathbf{v}\| = L^{(i)}(\mathbf{\Gamma}).$$

  It follows that

  $$\begin{aligned} L_+(\mathbf{M\Gamma},A(\mathbf{\Gamma})\mathbf{v}) &:= \lim_{n\to\infty}\frac{1}{n}\log\|A_n(\mathbf{M\Gamma})A(\mathbf{\Gamma})\mathbf{v}\| \\ &= \lim_{n\to\infty}\frac{1}{n}\log\|A_{n+1}(\mathbf{\Gamma})\mathbf{v}\| \\ &= \lim_{n\to\infty}\frac{1}{n}\log\|A_n(\mathbf{\Gamma})\mathbf{v}\| \\ &= L_+(\mathbf{\Gamma},\mathbf{v}). \end{aligned}$$

  Therefore, it can be concluded from the arguments for Part (i) that the vector $A(\mathbf{\Gamma})\mathbf{v} \in V_+^{(i)}(\mathbf{M\Gamma}) \setminus V_+^{(i+1)}(\mathbf{M\Gamma})$ for each $\mathbf{v} \in V_+^{(i)}(\mathbf{\Gamma}) \setminus V_+^{(i+1)}(\mathbf{\Gamma})$. This proves the containment $A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) \subseteq V_+^{(i)}(\mathbf{M\Gamma})$.

- ($\supseteq$) The fact that A is invertible applies here. Take $\mathbf{v} \in V_i(\mathbf{M\Gamma}) \setminus V_{i+1}(\mathbf{M\Gamma})$. It follows that

  $$\begin{aligned} L_+(\mathbf{M\Gamma},\mathbf{v}) &:= \lim_{n\to\infty}\frac{1}{n}\log\|A_n(\mathbf{M\Gamma})\mathbf{v}\| \\ &= \lim_{n\to\infty}\frac{1}{n}\log\|A_{n-1}(\mathbf{M\Gamma})A(\mathbf{\Gamma})\mathbf{v}\| \\ &= \lim_{n\to\infty}\frac{1}{n}\log\|A_n(\mathbf{M\Gamma})A(\mathbf{\Gamma})\mathbf{v}\| \\ &= L_+(\mathbf{M\Gamma},A(\mathbf{\Gamma})\mathbf{v}). \end{aligned}$$

  Again, it can be concluded from the arguments for Part (i) that the vector $\mathbf{v} \in A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) \setminus A(\mathbf{\Gamma})V_+^{(i+1)}(\mathbf{\Gamma})$. This proves the containment $A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) \subseteq V_+^{(i)}(\mathbf{M\Gamma})$.

Since containment was proven in both directions, Part (ii) is true [5].

For (iii), measurability follows from the fact that the objects $r(\mathbf{\Gamma})$, $L^{(i)}(\mathbf{\Gamma})$ and $\dim V_+^{(i)}(\mathbf{\Gamma})$ are obtained by a countable set operations on measurable sets. For the technical details from measure theory, see Ruelle [77] or Folland [29].

For the remainder of (iv), recall that $A(\mathbf{\Gamma})^{-1}$ exists, and is therefore full rank, by assumption. Since Part (ii) has been proven, it is true that

$$\dim V_+^{(i)}(\mathbf{M\Gamma}) = \dim A(\mathbf{\Gamma})V_+^{(i)}(\mathbf{\Gamma}) = \dim V_+^{(i)}(\mathbf{\Gamma}).$$

It was shown above that $L^{(i)}(\mathbf{M\Gamma}) = L^{(i)}(\mathbf{\Gamma})$ and $r(\mathbf{M\Gamma}) = r(\mathbf{\Gamma})$, and now it has been proven that $\dim V_+^{(i)}(\mathbf{M\Gamma}) = \dim V_+^{(i)}(\mathbf{\Gamma})$. Therefore, Part (iv) is true in its entirety.

For Part (v), begin by noting that

$$\begin{aligned}\det \Lambda(\mathbf{\Gamma}) &= \lim_{n\to\infty}\left[\det(A_n(\mathbf{\Gamma})^\top A_n(\mathbf{\Gamma}))\right]^{\frac{1}{2n}} \\ &= \lim_{n\to\infty} \det(A_n(\mathbf{\Gamma}))^{\frac{1}{n}}.\end{aligned}$$

It follows that

$$\begin{aligned}\lim_{n\to\infty}\frac{1}{n}\log\left|\det A_n(\mathbf{\Gamma})\right| &= \sum_{i=1}^{r(\mathbf{\Gamma})} \log\sigma_i(\mathbf{\Gamma}) \dim U^{(i)}(\mathbf{\Gamma}) \\ &= \sum_{i=1}^{r(\mathbf{\Gamma})} \log\sigma^{(i)}(\mathbf{\Gamma})(\dim V_+^{(i)}(\mathbf{\Gamma}) - \dim V_+^{(i+1)}(\mathbf{\Gamma})), \\ &:= \sum_{i=1}^{r(\mathbf{\Gamma})} \log\sigma^{(i)}(\mathbf{\Gamma}) m(\sigma^{(i)}(\mathbf{\Gamma})).\end{aligned}$$

Therefore, Part (v) is proven true.

As all Parts (i)-(v) have been proven true, this completes the proof. □

A.2 Support for Claims in Chapter 3

A.2.1 Proof that $\beta \in [0, 1]$

In Section 3.1.2, the following claim was made:

[5] In general, it is not true that $A(\mathbf{\Gamma})U^{(i)}(\mathbf{\Gamma}) = U^{(i)}(\mathbf{M\Gamma})$

Claim 1. The β of Equation (3.1.6) is such that $\beta \in [0, 1]$.

Proof. The parameter β must be chosen so that the momentum of each particle after collision is real-valued. This constraint forces the discriminant in Equation (3.1.11) to be greater than or equal to zero:

$$b_\alpha^2 - 4a_\alpha c_\alpha \geq 0, \tag{A.2.1}$$

where

$$\begin{aligned} a_\alpha &:= \frac{(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ b_\alpha &:= -\frac{(2\beta-1)(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ c_\alpha &:= \beta(\beta-1)|\mathbf{p}|^2. \end{aligned}$$

Expanding out all of the terms in Equation (A.2.1) gives the condition

$$(2\beta-1)^2(\mathbf{q}\cdot\mathbf{p})^2 - 4\beta(\beta-1)||\mathbf{p}||^2 \geq 0.$$

Now, invoke the law of cosines ($\mathbf{q}\cdot\mathbf{p} = ||\mathbf{q}||||\mathbf{p}||\cos\theta$). This results in

$$(2\beta-1)^2||\mathbf{q}||^2||\mathbf{p}||^2\cos^2\theta - 4\beta(\beta-1)||\mathbf{p}||^2 \geq 0,$$

where $\theta = \cos^{-1}(\frac{\mathbf{q}\cdot\mathbf{p}}{||\mathbf{q}||||\mathbf{p}||})$ is the angle between the relative position and momentum vectors immediately before collision. Just before collision, the particles are in contact, so $||\mathbf{q}|| = \sigma$, where σ is the diameter of each particle. Additionally, the term $||\mathbf{p}||^2$ in both terms on the left-hand side is positive definite[6], so it can be cancelled through. This leaves the inequality

$$(2\beta-1)^2\sigma^2\cos^2\theta - 4\beta(\beta-1) \geq 0.$$

The left hand side of the above equation after expansion and combination of like terms is

$$4\beta(\beta-1)(\sigma^2\cos^2\theta - 1) + \sigma^2\cos^2\theta \geq 0.$$

Moving the $\sigma^2\cos^2\theta$ term to the right hand side of the expression and multiplying through by the negative (taking care to flip the sign of the inequality) gives the following

$$4\beta(\beta-1)(1-\sigma^2\cos^2\theta) \leq \sigma^2\cos^2\theta.$$

[6]The term $||\mathbf{p}||$ must be non-zero for collision to occur.

Dividing both sides by $1 - \sigma^2 \cos^2\theta$ yields the final inequality

$$4\beta(\beta - 1) \leq \frac{\sigma^2 \cos^2\theta}{1 - \sigma^2\cos^2\theta}. \tag{A.2.2}$$

The right hand side of Equation (A.2.2) can be arbitrarily small; e.g. imagine a glancing collision where the relative momentum and position vectors at collision are nearly orthogonal, resulting in $\cos\theta \approx 0$. For the discriminant to be real-valued it is necessary that

$$4\beta(\beta - 1) \leq 0,$$

and this occurs if, and only if, $\beta \in [0, 1]$. □

A.2.2 Proof that $\inf \alpha = \beta$

In the caption of Figure 5.1, it was claimed that the infimum, or greatest lower bound, of α was equal to β.

Claim 2. The greatest lower bound of α, $\inf \alpha$, equals β.

Proof. First, expand all of the terms in Equation (3.1.11) and reduce:

$$\alpha = \frac{2\beta - 1}{2} + \frac{\sqrt{4\beta(\beta-1)((\mathbf{q}\cdot\mathbf{p})^2 - \mathbf{p}\cdot\mathbf{p}) + (\mathbf{q}\cdot\mathbf{p})^2}}{2|\mathbf{q}\cdot\mathbf{p}|}. \tag{A.2.3}$$

The strategy here is simple: it need only be shown that $4\beta(\beta-1)((\mathbf{q}\cdot\mathbf{p})^2 - \mathbf{p}\cdot\mathbf{p}) \geq 0$. Let $\mathbf{q} := (q_1\ q_2)^\top$ and $\mathbf{p} := (p_1\ p_2)^\top$. Expanding $(\mathbf{q}\cdot\mathbf{p})^2$ gives

$$\begin{aligned}(\mathbf{q}\cdot\mathbf{p})^2 &= (q_1p_1 + q_2p_2)^2 \\ &= (q_1^2 + q_2^2)(p_1^2 + p_2^2) - (q_1^2p_2^2 - 2q_1q_2p_1p_2 + q_2^2p_1^2) \\ &= \|\mathbf{q}\|^2\|\mathbf{p}\|^2 - (\mathbf{q}\wedge\mathbf{p})^2,\end{aligned} \tag{A.2.4}$$

where $\mathbf{q}\wedge\mathbf{p} := q_2p_1 - q_1p_2$. Immediately before collision, the particles are separated by one particle diameter σ (= 1 by construction for the simulation results presented in Chapters 4 and 5). Therefore, from Equation (A.2.4), the following is true

$$\begin{aligned}(\mathbf{q}\cdot\mathbf{p})^2 - \mathbf{p}\cdot\mathbf{p} &= \|\mathbf{q}\|^2\|\mathbf{p}\|^2 - (\mathbf{q}\wedge\mathbf{p})^2 - \|\mathbf{p}\|^2 \\ &= \|\mathbf{p}\|^2 - (\mathbf{q}\wedge\mathbf{p})^2 - \|\mathbf{p}\|^2 \\ &= -(\mathbf{q}\wedge\mathbf{p})^2.\end{aligned} \tag{A.2.5}$$

The term $(\mathbf{q} \wedge \mathbf{p})^2$ is positive semidefinite, therefore $(\mathbf{q} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p} \leq 0$. In Appendix A.2.1, it was shown that $4\beta(\beta - 1) \leq 0$. Therefore, it is true that

$$4\beta(\beta - 1)((\mathbf{q} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p}) \geq 0.$$

The above inequality can be applied to Equation (A.2.3) to get the following inequality for α:

$$\begin{aligned}
\alpha &= \frac{2\beta - 1}{2} + \frac{\sqrt{4\beta(\beta - 1)((\mathbf{q} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p}) + (\mathbf{q} \cdot \mathbf{p})^2}}{2|\mathbf{q} \cdot \mathbf{p}|} \\
&\geq \frac{2\beta - 1}{2} + \frac{\sqrt{(\mathbf{q} \cdot \mathbf{p})^2}}{2|\mathbf{q} \cdot \mathbf{p}|} \\
&= \frac{2\beta - 1}{2} + \frac{1}{2} \\
&= \beta.
\end{aligned} \tag{A.2.6}$$

Therefore, $\alpha \geq \beta$ and so $\inf \alpha = \beta$. $\square$

Appendix B
Computational Code

B.1 Code Availability

All source code discussed in this appendix is available at this link. You are free to use the code, but please cite accordingly.

B.2 Source Code Listings for Chapter 1

This section gives the Matlab® source code for recreating the data from Figures 1.2. This example uses the Hybrid Systems Simulation Solver [81]. The time histories are contained in the array **y**, where the indexing of the output is the same as in $\mathbf{\Gamma} := (x_\mathrm{b}\ v_\mathrm{b}\ x_\mathrm{p}\ v_\mathrm{p})^\top$.

```
%function run
global A w R;

% gr = 9.81;
% A = .1;
% R = 1;
% w = 4;
% h0 = -A; %period 1
% v0 = gr*pi/w; %period 1
% %v0 = -gr*pi/w;
% ht = -A;
% vt = 0;
% y0 = [h0;v0;ht;vt];
% tstop = 10;
% p = '1';

gr = 9.81;
A = .1;
R = 1;
w = 4;
h0 = .592; %period 2;
```

```
v0 = 0; %period 2;
%v0 = -gr*pi/w;
ht = -A;
vt = 0;
y0 = [h0;v0;ht;vt];
tstop = 10;
p = '2';

% gr = 9.81;
% A = .1;
% R = 1;
% w = 4;
% h0 = .697; %period 4!;
% v0 = 0; %period 4!;
% %v0 = -gr*pi/w;
% ht = -A;
% vt = 0;
% y0 = [h0;v0;ht;vt];
% tstop = 50;
% p = '4';

gr = 9.81;
A = .1;
R = 1;
w = 4;
h0 = 1.0; %period chaos!;
v0 = 0; %period chaos!;
%v0 = -gr*pi/w;
ht = -A;
vt = 0;
y0 = [h0;v0;ht;vt];
tstop = 15;
p = 'Chaos'

% simulation horizon
TSPAN=[0 tstop];
JSPAN = [0 2000];
% rule for jumps
% rule = 1 -> priority for jumps
% rule = 2 -> priority for flows
rule = 1;

%options = odeset('RelTol',1e-9,'MaxStep',.1);

```

```
% simulate
[t y j] = hybridsolver( @f,@g,@C,@D,y0,TSPAN,JSPAN,rule);
```

B.3 Source Code Listings for Chapter 2

B.3.1 Lorenz Example

The following source code examples recreate the data from Figure 2.1. The main driver is **repeatFroyland84.m**:

```
%script: repeatFroyland84.m
close all; clear all; clc
global rho;

rho_arr = .5:.5:500;
lyap = zeros(3,length(rho_arr));
for i = 1:length(rho_arr)
    rho = rho_arr(i);
    [Y lp] = liapunov(1000,[10 -20 14],20,.5);
    lyap(:,i) = lp(:,end);
end
save froyland1984
```

The function call to **liapunov()** handles the time integration and reorthonormalization of Benettin's algorithm (see Algorithm alg:ben):

```
% function liapunov.m
function [Ycalc lparr] = liapunov(tStop,ystart,nSteps,h)
global rho;
%find number of data columns
n      = length(ystart(1,:));
n1     = n;
n_exp  = n1;
n2     = n1*(n_exp+1);
y   = zeros(n2,1);
y0 = y;
cum=zeros(n2,1);
gsc=cum;
```

```
znorm=cum;
tstart=0;
%initialize integrator with desired tolerances
options = odeset('RelTol', 1e-8 , 'AbsTol', 1e-8, 'MaxStep', ↩
    2.5, ...
     'Refine',0,'InitialStep',0.001);
%initialize values
y(1:n) = ystart(:);
for i = 1:n_exp y((n1+1)*i)=1.0; end;

% debug step 1: pass
count = 0;
counter=0;
time = 0;
Ycalc(1,:) = ystart;
while (time < tStop)

    count = count + 1;

    [t1 Y] = ode45(@lrnz,[time time+h],y,options);
    y      = Y(size(Y,1) ,:);
    time = t1(size(Y,1));
    Ycalc(count+1,1:3) = y(1:3);
    yeig(count,1:12) = y(1:12);
    %
    %        construct new orthonormal basis by gram-schmidt
    %
    if (time > tstart)
    znorm(1)=0.0;
    for j=1:n1 znorm(1)=znorm(1)+y(n1+j)^2; end;

    znorm(1)=sqrt(znorm(1));

    for j=1:n1 y(n1+j)=y(n1+j)/znorm(1); end;

    for j=2:n_exp
        for k=1:(j-1)
            gsc(k)=0.0;
            for l=1:n1 gsc(k)=gsc(k)+y(n1*j+l)*y(n1*k+l); end;
        end;

        for k=1:n1
            for l=1:(j-1)
                y(n1*j+k)=y(n1*j+k)-gsc(l)*y(n1*l+k);
            end;
        end;

        znorm(j)=0.0;
        for k=1:n1 znorm(j)=znorm(j)+y(n1*j+k)^2; end;
        znorm(j)=sqrt(znorm(j));

        for k=1:n1 y(n1*j+k)=y(n1*j+k)/znorm(j); end;
    end;

    %
    %       update running vector magnitudes
    %

    for k=1:n_exp cum(k)=cum(k)+log(znorm(k)); end;
    %debug pt 6: result
    for k=1:n_exp lp(k)=cum(k)/(time-tstart); end;
    %debug pt 7: result
    if (mod(count,nSteps) == 0)
        counter = counter + 1;
        lparr(:,counter) = lp;
        tm(counter) = time;
    end
    end
end
```

The time histories of the phase space trajectories (as plotted in Figure 2.2) can be obtained by setting the variable **rho** in **repeatFroyland84.m** to 8, 28 or 355 and plotting the time history of the output vector in **Y**.

B.3.2 Bouncing Ball-Platform Example

The file needed for recreating the data of Figure 2.3 is **holmes.m**:

```
% script: holmes.m
close all; clear all; clc
w_loop = 0:.01:5;
b_loop = 1;

alpha = .4;
```

```
pnts = zeros(length(w_loop),201);
lyapHolmes = zeros(length(w_loop),2);

for ii = 1:length(w_loop)
    for jj = 1:1

        w = w_loop(ii) %0.3,0.8,1,1.2,1.5 2
        beta = b_loop(jj);
        %beta = 10;
        %w = 2;
        g = 9.81;
        gamma(ii) = 2*w^2*(1+alpha)*beta/g;
        n_exp = 2;
        n1 = n_exp;
        iter = 1001;
        n2 = n1*(n_exp+1);
        cum=zeros(n2,1);
        gsc=cum;
        znorm=cum;
        cnt = 0;
        dy(1) = 0;
        dy(2) = pi/2;

        %initialization for Lyapunov exponent computation
        dy(3)  = 1;
        dy(4)  = 0;
        dy(5)  = 0;
        dy(6)  = 1;

        for i = 1:iter
            %update reference trajectory via relation in G&H pg.↩
                103
            dyold = dy;
            dy(1) = mod(dyold(1) + dyold(2),2*pi);
            dy(2) = alpha*dyold(2) - gamma(ii)*cos(dy(1));

            %update tangent space dynamics via relation in G&H ↩
               pg. 103
            dy(3) = dyold(3) + dyold(4);
            dy(4) = gamma(ii)*sin(dy(1))*dyold(3) + (alpha + ↩
               gamma(ii)*sin(dy(1)))*dyold(4);
            dy(5) = dyold(5) + dyold(6);
            dy(6) = gamma(ii)*sin(dy(1))*dyold(5) + (alpha + ↩
               gamma(ii)*sin(dy(1)))*dyold(6);

            phi(i)= dy(1);
            v(i) = dy(2);

            if (mod(i,1) == 0)
                cnt = cnt + 1;
                % perform MGSR
                znorm(1)=0.0;
                for j=1:n1 znorm(1)=znorm(1)+dy(n1+j)^2; end;

                znorm(1)=sqrt(znorm(1));

                for j=1:n1 dy(n1+j)=dy(n1+j)/znorm(1); end;

                for j=2:n_exp
                    for k=1:(j-1)
                        gsc(k)=0.0;
                        for l=1:n1 gsc(k)=gsc(k)+dy(n1*j+l)*dy(n1*k+l); end;
                    end;

                    for k=1:n1
                        for l=1:(j-1)
                            dy(n1*j+k)=dy(n1*j+k)-gsc(l)*dy(n1*l+k);
                        end;
                    end;

                    znorm(j)=0.0;
                    for k=1:n1 znorm(j)=znorm(j)+dy(n1*j+k)^2; end;
                    znorm(j)=sqrt(znorm(j));

                    for k=1:n1 dy(n1*j+k)=dy(n1*j+k)/znorm(j); end;
                end;

                %
                %       update running vector magnitudes
                %

                for k=1:n_exp cum(k)=cum(k)+log(znorm(k)); end;
                %debug pt 6: result
                for k=1:n_exp lp(k)=cum(k)/i; end;
                lpArr(:,cnt) = lp;
            end

        end
        lyapHolmes(ii,:) = lpArr(:,end)';
        pnts(ii,:) = phi(end-200:end);
    end

end

figure(3);
subplot(2,1,1); hold on;
for ii = 1:length(pnts(:,1))
    plot(gamma(ii),pnts(ii,:),'k')
end
ylabel('Phase angle $$\phi$$','Interpreter','latex')
figure(3);hold off;

figure(3); subplot(2,1,2); hold on;
plot(gamma,squeeze(lyapHolmes(:,1)),'b',gamma,squeeze(lyapHolmes↩
    (:,2)),'r--');
xlabel('$$\mu$$ parameter for the Bouncing Ball-Platform System'↩
    ,'Interpreter','latex')
ylabel('$$L_i$$, $$i=1,2$$','Interpreter','latex')
h1 = legend('$$L_1$$','$$L_2$$');
set(h1,'Interpreter','latex');
figure(3); hold off;
```

B.3.3 Hénon Example

The driver routine for generating the data of Figures 2.4 and 2.5 is **testGinelli.m**:

```
% script: testGinelli.m
close all; clear; clc

global a b dim

format long g;

dim = 2;
a = 1.4;
```

```
b = 0.3;

% evolve far enough to ensure "well-relaxed" Lyapunov vectors

x0 = 0.5;
y0 = 0.4;

% full vectors includes nexp initially orthonormal eigenvectors
xold = [x0;y0;1;0;0;1];
x(:,1) = xold;

% fixed points p and q
xp = (b-1+sqrt( (b-1)^2 + 4*a ))/(2*a);
yp = b*xp;
xq = (b-1-sqrt( (b-1)^2 + 4*a ))/(2*a);
yq = b*xq;

p = [xp;yp];
q = [xq;yq];

numStepsToRelax = 2000;
%numStepsToRelax = 564;

for i = 1:numStepsToRelax
    x(:,i+1) = hmapFULL( squeeze(x(:,i)) );

    if ( mod(i,5) == 4 || i == numStepsToRelax )
        for j = 1:dim
            Gn(:,j) = x(dim*j+1:dim*j+2,i+1);
        end
        [Q,R] = qr(Gn); % note: this step performs the orthonormalization
        for j = 1:dim
            x(dim*j+1:dim*j+2,i+1) = Q(j,:);
        end
    end
end

figure(1); hold on;
scatter(x(1,:),x(2,:),'.')
plot(p(1),p(2),'kd','MarkerSize',14)
plot(q(1),q(2),'kd','MarkerSize',14)

%p = squeeze(x(:,end));
```

```
Jp      = [-2*a*p(1)  1;b  0];
[Vp Dp] = eig(Jp);
% plot(p(1),p(2),'kd','MarkerSize',14)
% perturb slightly off of fixed point p in the direction of the ↩
    stable
% subspace
xseg = (p(1)-7e-1):3e-5:(p(1)+7e-1);
ms   = Vp(2,2)/Vp(1,2);
yseg = ms*(xseg-p(1))+p(2);
seg(:,1) = xseg;
seg(:,2) = yseg;
%xstab = [xseg;yseg];
xstab = [];
for i = 1:length(seg(:,1))

    xold = seg(i,:)';
    for j = 1:4
        xnew = hinvmap(xold)';
        xold = xnew;
        xstab = [xstab,xnew];
    end
    %scatter(xstab(1,:),xstab(2,:),'r.');
end
scatter(xstab(1,:),xstab(2,:),'r.');
axis([-2.5 2.5 -2.5 2.5])
%figure(1); hold off;
clear R;

% evolve forward for a "sufficiently" long time
numStepsForward = 1000;
x1old = squeeze(x(:,end));
x1(:,1) = x1old;
tau = 1;

clear x;

n = 0;
cum = zeros(dim,1);
for i = 1:numStepsForward
    x1(:,i+1) = hmapFULL( squeeze(x1(:,i)) );

    if ( mod(i,tau) == 0 )
        n = n + 1;
        for j = 1:dim
```

```
            Gnbar(:,j) = x1(dim*j+1:dim*j+2,i+1);
        end
        [Gn,R1] = qr(Gnbar);
        for j = 1:dim
            x1(dim*j+1:dim*j+2,i+1) = Gn(j,:);
        end
        cum = cum + log(diag(abs(R1)));
        exp(:,n) = cum / (i+1);
        R(n).mat = R1;
        G(n).mat = Gn;
        x(n).state = x1(1:dim,i+1);
    end
end

% construct matrix Vn such that the column spans are the same as↩
    the final
% GS matrix (Gn from above): try just using Gn itself
C(n).mat = eye(2);
V(n).mat = Gn;
cumback = zeros(2,1);
count = 0;
for i = n:-1:2
    C(i-1).mat = R(i).mat\C(i).mat;
    for j = 1:dim
        nrm = norm(C(i-1).mat(:,j));
        cumback(j) = cumback(j) + log(nrm);
        C(i-1).mat(:,j) = C(i-1).mat(:,j) / nrm;
    end
    V(i-1).mat = G(i-1).mat*C(i-1).mat;

    count = count + 1;
    expBack(:,count) = cumback / (count * tau);
end

% test covariance of vectors
for i = 1:n
    J = eye(2);
    for j = 1:tau
        J = [-2*a*x(i).state(1) 1;b 0]*J;
    end
    vn = J*V(i).mat;
    for j = 1:dim
        vn(:,j) = vn(:,j) / norm(vn(:,j));
        diff(j,i) = norm( vn(:,j) - V(i).mat(:,j) );
    end

end
ms = V(1).mat(2,2) / V(1).mat(1,2);
mu = V(1).mat(2,1) / V(1).mat(1,1);
plot(x(1).state(1),x(1).state(2),'ko','MarkerSize',14);
xs = x(1).state(1) -.2:.01:x(1).state(1)+.2;
ys = ms*(xs-x(1).state(1))+x(1).state(2);
yu = mu*(xs-x(1).state(1))+x(1).state(2);
plot(xs,ys,'r','LineWidth',2)
plot(xs,yu,'k','LineWidth',2)
xlabel('x')
ylabel('y')
title('Covariant Vectors are Tangent to Stable (Red) and ↩
   Unstable (Blue) Manifolds')
plot(p(1),p(2),'kd','MarkerSize',14)

figure(1); hold off;
```

The above script **testGinelli.m** calls the function **hmapFull()** to compute the next iterate (for both the trajectory and linearized trajectory):

```
% function: hmapFull.m
function [ xnew ] = hmapFULL( xold )

global a b dim
xnew = zeros(dim*(dim+1),1);

% full nonlinear dynamics
xnew(1) = -a*xold(1)^2 + xold(2) + 1;
xnew(2) = b*xold(1);

% dim linear copies
for i = 1:dim
    xnew(dim*i+1) = -2*a*xold(1)*xold(dim*i+1) + xold(dim*i+2);
    xnew(dim*i+2) = b*xold(dim*i+1);
end
end
```

B.4 Source Code Listings for Chapters 4 & 5

This appendix is setup to explain the process for executing the code written to evaluate the Lyapunov spectrum for disk simulations using a generalized collision rule. Comments in the code are copious, however this document serves to provide theoretical background and references for further investigation. This section is structured in the following format:

1. **Global Variable Description-** presents global variable declarations from **utilities.h** along with a description of each variable.
2. **Detailed Module Description-** presents a description of what each individual function inside of the two source files **main.cpp** and **utilities.cpp** does.
3. **File I/O Description-** presents the proper structure (with examples) for formatting input files and reading output files (using Matlab).

We begin by discussing building and running the source files and follow with a module-by-module description of the code.

Note (IMPORTANT)**.** This code utilizes the fact that an $m \times n$ dimensional matrix can be represented as an mn dimensional vector. As C++ provides little capability out of the box to dynamically size multidimensional arrays, I decided to construct everything as a vector rather than a matrix. For instance, the vector **y** stores the $4N$ (N is the number of disks) values for the reference trajectory plus all **nlya** (see subsequent usage for more details) $4N$ offset vectors. Each disk has a 2-dimensional position and a 2-dimensional momentum component. Each $4N$ section of **y** is ordered as follows:

- The first $2N$ components pertain to the position (superscript (j) denotes the disk number):

$$\begin{pmatrix} q_x^{(1)} \\ q_y^{(1)} \\ q_x^{(2)} \\ q_y^{(2)} \\ \vdots \\ q_x^{(N)} \\ q_y^{(N)} \end{pmatrix}$$

In the case of a tangent vector, replace q with δq above.

- The remaining $2N$ components pertain to the momentum:

$$\begin{pmatrix} p_x^{(1)} \\ p_y^{(1)} \\ p_x^{(2)} \\ p_y^{(2)} \\ \vdots \\ p_x^{(N)} \\ p_y^{(N)} \end{pmatrix}$$

In the case of a tangent vector, replace p with δp above. The **nlya** tangent vectors are stacked sequentially into $\mathbf{y}$.

Unless stated otherwise, all matrices are represented by packing column-wise; that is the (i, j) element $a_{i,j}$ of an $m \times n$ matrix $\mathbf{A}$ is represented by the value $v(j * m + i)$ of an mn-dimensional vector $\mathbf{v}$.

B.4.1 Building the Code

In building/managing the code base, I have chosen to use development tools such as XCode on Mac OS X and Visual Studio on Windows OS. However, there is nothing OS-specific about any of the routines; everything is IEEE compliant. Therefore, from a Linux terminal, the command

```
g++ main.cpp utilities.cpp -o diskDynamics
```

will work.

B.4.2 Running the Code

To run the code from the command line, type

```
diskDynamics sampleInput.dat
```

Here, *sampleInput.dat* is the name of the input file specifying all of the system setup parameters. The code will produce erroneous data if either (1) the file is in the wrong format or (2) no input file is specified. See Section B.7.1 for proper input file formats.

B.5 Global Variable Description

B.5.1 utilities.h

This section presents the global variables used in the **diskDynamics** simulation; next to each variable declaration is the variable's description.

```
/* BEGIN GLOBAL VARIABLE DECLARATIONS
   NOTE ON GLOBAL DECLARATIONS: The statement (variable made ↩
       global for memory allocation convenience) is made to ↩
       indicate a global variable that is declared for convenient↩
        memory allocation ONLY.  These variables need not be ↩
       global, as they are not used in multiple functions, ↩
       however they are made global to prevent the re-allocation ↩
       of memory each time functions are called that use these ↩
       variables */

extern int DIM;                    // spatial dimension of the ↩
    configuration space

extern int nlya;                   // number of Lyapunov ↩
    exponents to compute

extern int nDisks;                 // number of disks

extern int phaseDim;               // dimension of total phase ↩
    space (=2*DIM*nDisks)

extern int noColl;                 // collision partner index ↩
    when no collision is possible (cannot be an integer
                                   // in [0,nDisks) )

extern int maxFlight;              // collision partner index ↩
    when the disk would travel a distance defined by a
                                   // distance constraint before ↩
                                      any possible collisions (↩
                                      cannot be an integer in
                                   // [0,nDisks) )

extern int iSeed;                  // value of random seed to ↩
    initialize random number generator

extern int tmbl;                   // flag to determine whether ↩
    or not to enable tumbling

extern int i_coll;                 // counter for number of ↩
    collisions that have occurred

extern int nOrtho;                 // number of simulation steps ↩
    in between successive applications of the Gram-
                                   // Schmidt orthnormalization ↩
                                      process to tangent vectors

extern int countCLV;               // counter for storing CLV ↩
    vectors

extern int modCLV;                 // counter for storing CLV ↩
    vectors

extern double *y;                  // array containing all of the↩
     state values (indices 0-(4N-1) ) and values for
                                   // the nlya tangent vectors (i↩
                                       = 1,nlya: indices (i*(4N)↩
                                      +0-(i+1)*(4N)-1) )

extern double *boxSize;            // size of the simulation box

extern double *maxFlightDblArray; // distance condition for ↩
```

```
   declaring maxFlight as a disk's collision partner

extern double *cum;               // vector storing accumulated ↩
   logarithms of tangent vector stretching factors

extern double dtTmbl;             // interval length for uniform↩
    random number draw on time between successive
                                  // tumbles of a disk;

extern double aspRatio;           // aspect ratio of the box (A ↩
   = Ly/Lx)

extern double density;            // volume density (packing ↩
   fraction) of disks in the simulation box

extern double beta;               // input value for weighting ↩
   factor beta in generalized collision rule

extern double alpha;              // computed value of alpha ↩
   required for conservation of kinetic energy in
                                  // generalized collision rule

extern double a_alpha;            // alpha^2 coefficient in ↩
   quadratic expression for alpha computation

extern double b_alpha;            // alpha coefficient in ↩
   quadratic expression for alpha computation

extern double c_alpha;            // constant coefficient in ↩
   quadratic expression for alpha computation

extern double stepSize;           // simulation time step

extern COLL   *collArray;         // array of structures storing↩
    own index, collision partner index and
                                  // associated collision time ↩
                                     for each disk (relative to ↩
                                     current simulation time
                                  // )

extern TUMBLE *tumbleArray;       // array of structures storing↩
    own index and time of next tumble (relative to
                                  // current simulation time)

```

```
extern double bigTime;              // (large) time to reinitialize disk next collision time after a collision
                                    // involving the disk has occurred

extern double Time;                 // simulation time

extern double TimeStartCollection; // time to start collecting data for statistical quantities (localization
                                    // width and cosine of angle between position and momentum perturbations in
                                    // Lyapunov vectors)

extern double vq;                   // dot product of relative momentum vector with relative position vector of
                                    // disks undergoing collision

extern double vv;                   // dot product of relative momentum vector with itself of disks undergoing
                                    // collision

extern double *dq1;                 // relative position vector connecting centers of disks undergoing collision

extern double *lSpec;               // spectrum of Lyapunov exponents

extern double *norm;                // tangent vector stretching factors

extern double *yi;                  // position vector of ith disk (variable made global for memory allocation
                                    // convenience)

extern double *yj;                  // position vector of jth disk (variable made global for memory allocation
                                    // convenience)

extern double *vi;                  // momentum vector of ith disk (variable made global for memory allocation
                                    // convenience)

extern double *vj;                  // momentum vector of jth
```

```
    disk (variable made global for memory allocation
                                    // convenience)

extern double *v;                   // relative momentum vector ↩
    between disks undergoing colliskion (variable made
                                    // global for memory ↩
                                       allocation convenience)

extern double *dq;                  // relative tangent position ↩
    vector between disks undergoing collision
                                    // (variable made global for ↩
                                       memory allocation ↩
                                       convenience)

extern double *dv;                  // relative tangent momentum ↩
    vector between disks undergoing collision
                                    // (variable made global for ↩
                                       memory allocation ↩
                                       convenience)

extern double *dqc;                 // difference in relative ↩
    position vectors between collision points in
                                    // reference (at q) and ↩
                                       perturbed (at q+dqc) ↩
                                       trajectories (variable ↩
                                       made
                                    // global for memory ↩
                                       allocation convenience)

extern double *qi;                  // orthogonal projector ↩
    column vector for modified Gram-Schmidt process
                                    // (variable made global for ↩
                                       memory allocation ↩
                                       convenience)

extern double *mean;                // computed mean of disk ↩
    momenta after initialization by random draw (variable
                                    // made global for memory ↩
                                       allocation convenience)

extern double *cumBatch;            // same as cum, except this ↩
    variable is used as an accumulation of the log of
                                    // stretching factors over ↩
                                       shorter periods of time ↩
```

```
                                      for computing batch
                                   // averages

extern double *avgCumBatch;        // same as lSpec, except this↩
     variable is used as an average over shorter
                                   // periods of time for ↩
                                      computing batch averages

extern STATS  stats;               // stats structure stores ↩
    statistical quantities: localization width and
                                   // cosine of angle between ↩
                                      position and momentum ↩
                                      perturbations of tangent
                                   // vectors

extern STATS  clvStats;            // stats structure stores ↩
    statistical quantities for CLVs: localization width and
                                   // cosine of angle between ↩
                                      position and momentum ↩
                                      perturbations of tangent
                                   // vectors

extern int lastPass;               //

extern int countMbatch;            // counter for batch average ↩
    processing

extern int numMbatch;              // size of batch average ↩
    processing interval (in number of counts)

extern int computeBatchAvg;        // flag to determine whether ↩
    or not to enable batch average collection

extern int writeAlpha;             // flag to determine whether ↩
    or not to write value of alpha (and dot(v,q) and
                                   // dot(v,v)) to output file ↩
                                      after each collision

extern int maxAlphaCount;          // maximum number of values ↩
    of alpha (and dot(v,q) and dot(v,v)) to record

extern int writeVACF;              // flag to determine whether ↩
    or not to write velocity autocorrelation function
                                   // values (and associated ↩
                                    times) to an output file

extern int iVacf;                  // counter for velocity autocorrelation function data collection

extern int processCLVs;            // flag to determine whether or not to compute covariant vectors (CLVs)

extern ofstream batchData;                   // file handle for batch average output file

extern ofstream alphaData;                   // file handle for alpha output file

extern ofstream vacfData;                    // file handle for velocity autocorrelation function output file

extern ofstream clvData;                     // file handle for covariant vector output file

extern vector< vector<double> > velocities; // vector holding time history of velocities from state vector for
                                             // velocity autocorrelation processing

extern vector< vector<double> > Rinv;        // vector holding time history of Gram-Schmidt scaling factors for
                                             // CLV computation

extern vector< vector<double> > GS;          // vector holding time history of Gram-Schmidt orthonormal bases
                                             // for CLV computation

extern vector< vector<double> > traj;        // vector holding time history of phase-space trajectories
                                             // for CLV computation

extern double vacfSettleTime;                // time before recording data for velocity autocorrelation function
                                             // processing

```

```
extern double vacfCaptureTime;              // amount of time to collect velocity data for velocity
                                            // autocorrelation processing

extern double clvSettleTime;                // amount of time before before beginning Ginelli method computations

extern int    modVacf;                      // number of time steps in-between successive collections of velocity
                                            // data for velocity autocorrelation function processing

extern ifstream  initialData;               // input file stream name holding initial state data
/* END GLOBAL VARIABLE DECLARATIONS */
```

B.6 Detailed Module Description

B.6.1 main.cpp

```
//
//  FUNCTION:  main
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function is the main executive routine for generalized collision simulations.
//
//  INPUTS:
//   argc - standard form for C/C++ main routine for integer input (currently unused)
//   argv - standard form for C/C++ main routine for character array input (used for input filename to read initialization data from)
//
//  OUTPUTS:
//   0 - normal execution
//   1 - clean exit due to user-defined exception declared
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   (none)
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   tmbl                (value declared)
//   dtTmbl              (value declared)
//   beta                (value declared)
//   writeAlpha          (value declared)
//   stepSize            (value declared)
//   nOrtho              (value declared)
//   nDisks              (value declared)
//   density             (value declared)
//   nlya                (value declared)
//   aspRatio            (value declared)
//   computeBatchAvg     (value declared)
//   numMbatch           (value declared)
//   writeVACF           (value declared)
//   vacfSettleTime      (value declared)
//   modVacf             (value declared)
//   vacfCaptureTime     (value declared)
//   batchData           (value declared and file opened for writing↩
    )
//   alphaData           (value declared and file opened for writing↩
    )
//   DIM                 (value declared)
//   phaseDim            (value declared)
//   iSeed               (initialized)
//   noColl              (value declared)
//   maxFlight           (value declared)
//   bigTime             (value declared)
//   maxAlphaCount       (value declared)
//   TimeStartCollection (value declared)
//   i_coll              (initialized)
//   Time                (initialized and incremented)
//   iVacf               (initialized)
//   countMbatch         (initialized)
//   y                   (memory allocated and specified states ↩
    written to file)
//   lSpec               (memory allocated)
//   collArray           (memory allocated)
//   tumbleArray         (memory allocated)
//   boxSize             (memory allocated and values declared)
//   maxFlightDblArray (memory allocated and values declared)
```

```
55 //   norm              (memory allocated)
56 //   cumBatch          (memory allocated)
57 //   avgCumBatch       (memory allocated)
58 //   qi                (memory allocated)
59 //   yi                (memory allocated)
60 //   yj                (memory allocated)
61 //   vi                (memory allocated)
62 //   vj                (memory allocated)
63 //   v                 (memory allocated)
64 //   dq                (memory allocated)
65 //   dv                (memory allocated)
66 //   dqc               (memory allocated)
67 //   vacfData          (value declared, file opened for writing, ↩
      and output
68 //                      written to file)
69 //   velocities        (vector is cleared after velocity ↩
      autocorrelation data has
70 //                      been processed)
71 //   modCLV            (initialized)
72 //   lastPass          (initialized to false)
73 //   clvData           (value declared, file opened for writing)
74 //
75 //   REVISION HISTORY:
76 //   Dinius, J.        Created                                ↩
      10/08/11
77 //   Dinius, J.        Comments added for v1.0 release        ↩
      05/27/13
78 //   Dinius, J.        Cleaned up for dissertation release    ↩
      11/25/13
79 //
80 int main (int argc, const char * argv[])
81 {
82     //OPEN INPUT FILE
83   inData.open(argv[1]);
84
85   if (!inData){
86     // INPUT FILE COULD NOT BE OPENED, SO EXIT
87     cerr << "Error: input file could not be opened.  Exiting..."↩
         << endl;
88     exit(1);
89   } // END if (!inData)
90   %char *initialFile = new char [256];
```

Line 80 above opens the input file declared as the first argument to the command line.

Lines 85 checks to make sure that the file can be opened; if it cannot, an error message is reported and the executable returns 1. In the case of normal executable operation, the program returns 0 at the end of the simulation.

In what follows, the source lines:

```
 while (dummyStr.compare(str1) != 0) {
         inData >> dummyStr;
     }
```

pull in the string values after each number or character array and moves the current point in the file to the string ***/**, *i.e.* the end of each line. This is a convention taken by convenience given the choice of C++ as the programming language. Note: when modifying input files, you will need to preserve the file structure AS IS, otherwise the program will not run properly; only numbers and output filename can be changed.

```
    // READ IN DATA FROM INPUT FILE AND OUTPUT VALUES TO SCREEN
    inData >> tmbl;
    cout << "Tumble (1) or not (0)?  Tumble =  " << tmbl << endl↩
       ;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the integer flag **tumble** that determines if tumbling should be turned on (1) or off (0).

```
    if (tmbl != 0){
        // IF TUMBLE HAS BEEN SELECTED, READ IN THE TIME ↩
           INTERVAL TO DRAW THE NEXT TUMBLE TIME FROM
        inData >> dtTmbl;
        inData >> dummyStr;
        cout << "\tMax time between tumble events for single ↩
           disk (uniformly dist) = " << dtTmbl << endl;
        while (dummyStr.compare(str1) != 0){
            inData >> dummyStr;
        } // END while (dummyStr.compare(str1) != 0)

    } // END if (tmbl != 0)
```

If tumbling has been enabled, the program will read in (and output to the screen) the next value in the input file which corresponds to the upper bound for a uniform distribution, **dtTmbl**. This value represents the maximum amount of time between tumbles for individual disks (see *tumble* section below).

```
    inData >> beta;
    inData >> dummyStr;
    cout << "beta = " << beta << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the weighting factor **beta** (β). See Subsection *collision* below.

```
    inData >> writeAlpha;
    inData >> dummyStr;
    cout << "Write alpha to file? " << writeAlpha << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the integer flag to output α for each collision after computation (1) or not (0). See Subsections *collision* and *computeAlpha*).

```
    inData >> stepSize;
    inData >> dummyStr;
    cout << "Stepsize = " << stepSize << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the time step **stepSize** (*dt*) to be used for forward time evolution of the system states.

```
    inData >> nSteps;
    inData >> dummyStr;
    cout << "No. of Steps = " << nSteps << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the number of time steps to execute the simulation for.

```
    inData >> nOrtho;
    inData >> dummyStr;
    cout << "No. of Steps between Orthonormalization Operations ↩
       = " << nOrtho << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the number of time steps between successive iterations of the MGSR process.

```
    inData >> nOutput;
    inData >> dummyStr;
    cout << "No. of Steps between File Output = " << nOutput << ↩
       endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the number of time steps between writing current system states to output file. As will be shown later, this functionality is currently commented-out. Only output at the end of the simulation is recorded. However, in the future an additional input flag can be used to turn intermediate output on/off.

```
   inData >> nDisks;
    inData >> dummyStr;
    cout << "No. of Disks = " << nDisks << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the desired number of disks **nDisks** (N).

```
    inData >> density;
    inData >> dummyStr;
    cout << "Density of disks in box = " << density << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the desired disk density **density** (ρ).

```
    inData >> nlya;
    inData >> dummyStr;
    cout << "No. of Lyapunov Exponents to Calculate = " << nlya ↩
       << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the number of Lyapunov exponents to calculate **nlya** (and hence the number of orthonormal vectors required at initialization). The variable **nlya** determines how many (initially orthonormal) tangent space vectors need to be evolved in time.

```
    inData >> aspRatio;
    inData >> dummyStr;
    cout << "Aspect Ratio (Ly/Lx) = " << aspRatio << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the desired aspect ratio **aspRatio** (A). The aspect ratio is defined as the ratio of the box length in the y-direction to the box length in the x-direction

$$A = \frac{L_y}{L_x}. \tag{B.6.1}$$

```
    inData >> computeBatchAvg;
    inData >> dummyStr;
    cout << "Compute batch average? = " << computeBatchAvg << ↩
       endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the flag to compute batch averages of computations for Lyapunov exponents. This capability is used to evaluate how closely samples of batch averages resemble a random process with normal distribution.

```
    inData >> numMbatch;
    inData >> dummyStr;
    cout << "Batch size = " << numMbatch << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the number of normalization factors to average over to create a batch average. See Subsection *update*.

```
    inData >> writeAllModes;
    inData >> dummyStr;
    cout << "Record all modes? (0=No,1=Yes) = " << writeAllModes↩
        << endl;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the integer flag **writeAllModes** that determines if all of the vectors associated with the MGSR process (along with the states from the reference trajectory) should be output (1) or not (0). The vectors are also called *modes*.

```
    if (!writeAllModes){
        // IF NOT WRITING OUT ALL COMPUTED TANGENT VECTORS, ↩
           CHECK TO SEE IF AN OUTPUT RANGE OF TANGENT VECTORS IS↩
            DESIRED
        inData >> writeRangeModes;
        inData >> dummyStr;
        cout << "Write out range of modes? (0=No,1=Yes) = " << ↩
           writeRangeModes << endl;
        while (dummyStr.compare(str1) != 0){
            inData >> dummyStr;
        } // END while (dummyStr.compare(str1) != 0)
```

If not **writeAllModes**, read-in the integer flag **writeRangeModes**, which allows the user to specify an interval of modes to write. The system state (index 0) is recorded by default.

```
        if (!writeRangeModes){
            // IF NOT WRITING OUT ALL TANGENT VECTORS, AND NOT ↩
               WRITING OUT A RANGE OF VECTORS, READ IN THE ↩
               NUMBER OF VECTORS TO OUTPUT
            inData >> nModes;
            inData >> dummyStr;
            cout << "No. of Modes to write out = " << nModes << ↩
               endl;
            while (dummyStr.compare(str1) != 0){
                inData >> dummyStr;
            } // END while (dummyStr.compare(str1) != 0)

        } // END if (!writeRangeModes)
```

If not **writeRangeModes**, then the desired number of modes to write out is read-in.

```
        else {
            // IF NOT WRITING OUT ALL TANGENT VECTORS, BUT ↩
               RATHER WRITING OUT A RANGE OF VECTORS, READ IN ↩
               THE LOWER AND UPPER BOUNDS FOR THE INTERVAL OF ↩
               VECTORS TO OUTPUT
            inData >> bottomIndx;
            inData >> dummyStr;
            cout << "\tBottom index of range of modes? = " << ↩
               bottomIndx << endl;
            while (dummyStr.compare(str1) != 0){
                inData >> dummyStr;
            } // END while (dummyStr.compare(str1) != 0)

            inData >> topIndx;
            inData >> dummyStr;
            cout << "\tTop index of range of modes? = " << ↩
               topIndx << endl;
            while (dummyStr.compare(str1) != 0){
                inData >> dummyStr;
            } // END while (dummyStr.compare(str1) != 0)

            // NUMBER OF VECTORS TO RECORD (ADDING ONE TO ↩
               ACCOUNT FOR THE PHASE SPACE VECTOR OF THE ↩
               REFERENCE TRAJECTORY)
            nModes = topIndx - bottomIndx + 1 + 1;

        } // END else

    } // END if (!writeAllModes)
```

If **writeRangeModes** has been specified, read in the start and end of the interval of mode numbers to write out. The number of modes **nModes** is set to the interval width (**topIndx** - **bottomIndx**+1) plus 1 to account for the system state (index 0).

```
    else {
        // RECORD ALL VECTORS (ADDING ONE TO ACCOUNT FOR THE ↩
           PHASE SPACE VECTOR OF THE REFERENCE TRAJECTORY)
        nModes = nlya+1;

    } // END else
```

If the **writeAllModes** and **writeRangeModes** flags are set to 0, the number of vectors to record will be the number of Lyapunov exponents **nlya** + 1. The +1 comes from the recording of the reference trajectory state along with the Lyapunov modes. This numbering is required since all of the dynamics are contained in the **y** vector (see declaration below).

```
    // CREATE INTEGER ARRAY TO STORE INDICES OF VECTORS TO ↩
       OUTPUT
    int *modes = new int [nModes];
```

Declare new integer array to contain the desired modes to record.

```
    if (!writeAllModes){
        // IF NOT WRITING OUT ALL VECTORS,
        if (!writeRangeModes){
            // IF NOT WRITING OUT A RANGE OF VECTORS, THE FIRST ↩
               OUTPUT VECTOR IS THE PHASE SPACE TRAJECTORY OF ↩
               THE REFERENCE TRAJECTORY
            modes[0] = 0;
            cout << "\tWriting out state" << endl;
            // LOOP OVER NUMBER OF DESIRED OUTPUT TANGENT ↩
               VECTORS (IF nModes IS SET TO ZERO, THEN LOOP IS ↩
               SKIPPED AND ONLY REFERENCE TRAJECTORY STATE IS ↩
               OUTPUT)
            for (i = 1; i <= nModes; i++){
                inData >> modes[i];
                inData >> dummyStr;
                while (dummyStr.compare(str1) != 0){
                    inData >> dummyStr;
                } // END while (dummyStr.compare(str1) != 0)

                cout << "\tWriting out Mode " << i << " = " << ↩
                   modes[i] << endl;

            } // END for (i = 1; i <= nModes; i++)

            // NUMBER OF VECTORS OUTPUT (ADDING ONE TO ACCOUNT ↩
               FOR THE PHASE SPACE VECTOR OF THE REFERENCE ↩
               TRAJECTORY)
            nModes += 1;

        } // END if (!writeRangeModes)
```

If not **writeAllModes** and not **writeRangeModes**, then the modes to write out are read in and output to the screen. The vector with index 0 (the system state), is recorded by default.

```
        else {
            // IF WRITING OUT A RANGE OF VECTORS, THE FIRST ↩
               OUTPUT VECTOR IS THE PHASE SPACE TRAJECTORY OF ↩
               THE REFERENCE TRAJECTORY
            modes[0] = 0;
            // LOOP OVER NUMBER OF DESIRED OUTPUT TANGENT ↩
               VECTORS (ON THE INTERVAL [bottomIndx,topIndx])
            for (i=1; i < nModes; i++){
                modes[i] = bottomIndx + i - 1;

            } // END for (i=1; i < nModes; i++)

        } // END else

    } // END if (!writeAllModes)
```

If **writeRangeModes** is set, then set the first mode to record to be the system state, then set modes to record sequentially beginning from **bottomIndx** to **topIndx**.

```
    else {
        // IF WRITING OUT ALL VECTORS
        for (i = 0; i < nModes; i++){
            modes[i] = i;
        } // END for (i = 0; i < nModes; i++)

    } // END else
```

If **writeAllModes** = 1, then set the modes vector to contain all non-negative integers up to **nlya**. This assumes indexing from 0 (representing the system state), which is implicit in the C++ language.

```
    inData >> writeVACF;
    cout << "Write velocities for autocorrelation function ↩
```

```
        processing? " << writeVACF << endl;
    inData >> dummyStr;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in (and outputs to screen) the integer flag **writeVACF** that determines if computations of velocity autocorrelation functions [41] should be output to its own file (1) or not at all (0). See Subsection *computeVacf*.

```
   if (writeVACF){
        // IF WRITING OUT VELOCITY AUTOCORRELATION FUNCTION DATA↩
            , SET PARAMETERS TO VALUES READ FROM FILE
        inData >> vacfSettleTime;
        cout << "Time before recording VACF data = " << ↩
           vacfSettleTime << endl;
        inData >> dummyStr;
        while (dummyStr.compare(str1) != 0){
            inData >> dummyStr;
        } // END while (dummyStr.compare(str1) != 0)

        inData >> modVacf;
        cout << "Number of steps between VACF data captures = " ↩
           << modVacf << endl;
        inData >> dummyStr;
        while (dummyStr.compare(str1) != 0){
            inData >> dummyStr;
        } // END while (dummyStr.compare(str1) != 0)

        inData >> vacfCaptureTime;
        cout << "Time duration to record VACF data = " << ↩
           vacfCaptureTime << endl;
        inData >> dummyStr;
        while (dummyStr.compare(str1) != 0){
            inData >> dummyStr;
        } // END while (dummyStr.compare(str1) != 0)

        // IF writeVACF IS NON-ZERO, SET THE INITIAL COLLECTION ↩
           TIME FOR LOCALIZATION MEASURE CALCULATION TO ↩
           vacfSettleTime
        TimeStartCollection = vacfSettleTime;
```

```
    } // if (writeVACF)
```

If the **writeVACF** flag is not set to 0, read in data specifying when to start recording data (**vacfSettleTime**), the number of time steps in between successive data captures (**modVacf**), the time duration for recording VACF data (**vacfCaptureTime**), and the time to start collecting data for computations of the localization measure (See Subsections *accumulateLocalizationWidth*) and the average cosine of the angle between position and momentum components of tangent vectors *computeAvgCosTheta*).

```
    else {
        // IF NOT WRITING OUT VELOCITY AUTOCORRELATION FUNCTION ↩
           DATA, SET PARAMETERS TO DEFAULTS
        vacfSettleTime = 100.;
        modVacf = 20;
        vacfCaptureTime = 0.;
        TimeStartCollection = 100.;
    } // END else
```

If the **writeVACF** flag is set to 0, initialize data to default parameters for when to start recording data (**vacfSettleTime**), the number of time steps in between successive data captures (**modVacf**)the time duration for recording VACF data (**vacfCaptureTime**), and the time to start collecting data for computations of the localization measure (See Subsection *accumulateLocalizationWidth*) and the average cosine of the angle between position and momentum components of tangent vectors (see Subsection *computeAvgCosTheta*).

```
    inData >> outFile;
    outData.open(outFile,ios::out | ios::binary);
    cout << "File to record output: " << outFile << endl;
    inData >> dummyStr;
    while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
```

Reads in and displays the filename to write output to **outFile**. The file handle for file output is **outData**.

```
    inData >> readInInitialState;
  cout << "Read-in initial state? " << readInInitialState << ↩
     endl;
  inData >> dummyStr;
  while (dummyStr.compare(str1) != 0){
    inData >> dummyStr;
  } // END while (dummyStr.compare(str1) != 0)
```

Reads in and displays whether or not the starting (initial) condition is to be read in from a binary file in **initialize()** (see Subsection *initialize*).

```
  if (readInInitialState) {
    inData >> initialFile;
    cout << "Binary file with initial state: " << initialFile <<↩
        endl;
    inData >> dummyStr;
    while (dummyStr.compare(str1) != 0){
      inData >> dummyStr;
    } // END while (dummyStr.compare(str1) != 0)
    initialData.open(initialFile,ios::in|ios::binary);
  }
```

If the starting (initial) condition is to be read in from a binary file, read in and display the filename to the screen and open the binary file.

```
  inData >> processCLVs;
  cout << "Compute covariant vectors? " << processCLVs << endl;
  inData >> dummyStr;
  while (dummyStr.compare(str1) != 0){
    inData >> dummyStr;
  } // END while (dummyStr.compare(str1) != 0)
```

Reads in and displays whether or not covariant Lyapunov vectors will be computed. See Subsection *computeCLVs*.

```
  if (processCLVs){
    inData >> clvSettleTime;
    cout << "Time to run to settle GS transients? " << ↩
```

```
          clvSettleTime << endl;
      inData >> dummyStr;
      while (dummyStr.compare(str1) != 0){
        inData >> dummyStr;
      } // END while (dummyStr.compare(str1) != 0)
    }
    else {
      // IF NOT PROCESSING CLVs, SET THE SETTLE TIME TO SOMETHING ↪
         VERY LARGE
      clvSettleTime = 1.0e9;
    }
```

If covariant vectors are to be computed, read in and display the settling time for transients in the forward Gram-Schmidt process. If not computing covariant vectors, set the settle time to something large to avoid storing unneeded data.

```
      // CLOSE INPUT FILE
      inData.close();

      // END READ IN DATA FROM INPUT FILE AND OUTPUT VALUES TO ↪
         SCREEN
```

The file input process is ended by closing the file handle associated with the input file.

The following lines of source open output files for writing if the inputs from above are set:

The samples of the batch averaging process:

```
      // IF INDICATED FROM INPUT FILE, OPEN OPTIONAL FILES FOR ↪
         OUTPUT
      if (computeBatchAvg){
          string   str2 (outFile);
          string   batchFile("batch_");
          batchFile.append(str2);
          batchData.open(batchFile,ios::out | ios::binary);
      }
```

The samples for α:

```
    if (writeAlpha){
        string   str2 (outFile);
        string   batchFile("alpha_");
        alphaFile.append(str2);
        alphaData.open(alphaFile,ios::out | ios::binary);
    }
```

The covariant Lyapunov vector output (vectors, localization and angle data):

```
    if (processCLVs){
        string   str2 (outFile);
        string   clvFile("clv_");
        clvFile.append(str2);
        clvData.open(clvFile,ios::out | ios::binary);
    } // END if (processCLVs)
```

In the lines that follow, basic setup parameters are set.

```
     // SET PROGRAM CONSTANTS/DEFAULTS
     DIM        = 2;
```

Sets the dimension of position space. **DIM** = 2 indicates a two-dimensional box.

```
    phaseDim   = 2*DIM*nDisks;
```

Sets the dimension of the system phase space. Each disk contributes 2***DIM** degrees-of-freedom (**DIM** for position and **DIM** for momentum).

```
    iSeed      = 112324;
```

Sets the seed for the random number generator used to initialize the disk velocities.

```
    noColl     = -1;
```

Sets the enumerative value of **noColl**. See subsequent usage for more details.

```
    maxFlight = -2;
```

Sets the enumerative value of **maxFlight**. See subsequent usage for more details.

```
    bigTime   = 1.0e11;
```

Sets the value to reinitialize the collision time of disks that have just collided (something big relative to the time step is appropriate).

```
    maxAlphaCount = 5000000;
```

Sets the number of α values to record before closing output file **alphaFile**.

```
    modCLV  = 25;
```

Sets the number of orthonormalization steps in-between storage of Gram-Schmidt orthonormalization data from **mgsr()**.

```
    lastPass = 0;
```

Sets the flag indicating whether or not to store the inverse of the **R** matrix from the Gram-Schmidt process. The matrices are stored once the **processCLVs()** routine has been entered. This variable is used to save on memory usage.

```
    // INITIALIZE COUNTERS AND TIME
    i_coll     = 0;
```

Initializes the counter for number of collisions to 0;

```
    iVacf     = 0;
```

Initializes the counter for writing velocity autocorrelation data to file;

```
    Time        = 0.0;
```

Initializes the sim time to 0.

```
    countMbatch = 0;
```

Initializes the counter for collection of batch averages of Lyapunov exponent computation.

```
    // SET SIZE OF SIMULATION BOX
    // SOLVES N*pi*R^2 = density * A == density*(aspRatio*L^2) ↩
       FOR L, DISKS HAVE RADIUS 0.5 BY CONSTRUCTION
    boxSize[0] = sqrt( ((double) nDisks) * pi / 4.0 / density / ↩
       aspRatio );
    boxSize[1] = aspRatio * boxSize[0];
```

Sets the box size according to input parameters (recall Equation (B.6.1)). The disk diameter is σ.

$$\begin{aligned} N\pi\frac{\sigma^2}{4} &= \rho L_x L_y = \rho L_x (A L_x) \\ (\Longrightarrow)\, L_x &= \sigma\sqrt{\frac{N\pi}{4\rho A}} \\ (\Longrightarrow)\, L_y &= A L_x. \end{aligned}$$

```
    // SET SIZE OF MAXIMUM FLIGHT CONDITION
    // THE MAXIMUM FLIGHT CONDITION IS, ARBITRARILY, SET TO 1/4 ↩
       THE BOX LENGTH IN EACH DIMENSION (MINUS THE DISK RADIUS)
    for (int i = 0; i < DIM; i++){
        maxFlightDblArray[i] = (boxSize[i] / 2.0 - 1.0) / 2.0;
    } // END for (int i = 0; i < DIM; i++)
```

Sets the maximum flight condition length, arbitrarily, to one-quarter the box length in each direction minus the disk radius. The factor of 1.0 comes from the disk diameter ($\sigma = 1$ for convenience). In the routine **updateTimes()**, if a disk meets the maximum flight condition, it is treated similarly to a collision with the “partner” **maxFlight** (defined above). See Subsection *updateTimes* for more details.

```
    // INITIALIZE
    initialize();
    // END INITIALIZE
```

Initializes the disk positions and momenta. See Subsection *initialize*.

```
    ///////////////////////////////////////////////////////
    //                MAIN LOOP                   //
    ///////////////////////////////////////////////////////
    for (i = 0; i < nSteps; i++){
```

Begins main simulation loop over time.

```
        if ( (i % nOrtho) == 0 ){

            // PERFORM GRAM-SCHMIDT REORTHONORMALIZATION OF ↩
               TANGENT VECTORS
            mgsr();
            // UPDATE COMPUTATION OF STRETCHING FACTORS AND ↩
               LYAPUNOV EXPONENTS
            update();

            if (Time >= TimeStartCollection){
                // IF Time EXCEEDS A THRESHOLD, BEGIN COLLECTING↩
                    DATA FOR LOCALIZATION WIDTH AND AVERAGE ↩
                   COSINE OF THE ANGLE BETWEEN POSITION AND ↩
                   MOMENTUM CONTRIBUTIONS TO TANGENT VECTORS
                accumulateLocalizationWidth();
                computeAvgCosTheta();
            } // END if (Time >= TimeStartCollection)

        } // END if ( (i % nOrtho) == 0 )
```

If $\bmod(i, \textbf{nOrtho}) = 0$, then perform the MGSR process (**mgsr()**, Subsection *mgsr*), compute the log of the normalization factors (**update()**, Subsection *update*) and, when **Time** is greater than or equal to **TimeStartCollection**, call **accumulateLocalizationWidth()** (Subsection *accumulateLocalizationWidth* routine to compute the average of the localization measure for Lyapunov vectors [91], and finally call the **computeAvgCos()** (Subsection *computeAvgCosTheta*) routine to compute the average of the cosine of the angle between the position and momentum perturbations for each Lyapunov vector [21, 30].

```
        if ( (i % nOutput) == 0 ){
            // OUTPUT CURRENT TIME AND LARGEST LYAPUNOV EXPONENT VALUES TO THE SCREEN AS A STATUS CHECK
            cout << Time << "\t" <<lSpec[0] << endl;

            // OUTPUT CURRENT TIME TO FILE
            outData.write((char *) &Time, sizeof(double));

            // OUTPUT SPECIFIED VECTORS AT CURRENT TIME TO FILE (SEQUENTIALLY BY VECTOR INDEX)
            for (ii = 0; ii < nModes; ii++){
                for (j = 0; j < phaseDim; j++){
                    outData.write((char *) &y[modes[ii]*phaseDim+j], sizeof(double));
                } // END for (j = 0; j < phaseDim; j++)

            } // END for (ii = 0; ii < nModes; ii++)

            // OUTPUT LYAPUNOV SPECTRUM AT CURRENT TIME
            for (ii = 0; ii < nlya; ii++){
                outData.write((char *) &lSpec[ii], sizeof(double));
            } // END for (ii = 0; ii < nlya; ii++)

        } // END if ( (i % nOutput) == 0 )
```

If $\mod(i, \mathbf{nOutput}) = 0$, then write the current sim time and the value of the largest Lyapunov exponent computed. The logic writes the current time, all of the desired output modes (the 0th is the reference trajectory) current values, and the current calculations of the Lyapunov exponents at time steps such that $\mod(i, \mathbf{nOutput}) = 0$.

```
        // PERFORM TIME STEP PROPAGATION OF SYSTEM
        hardStep(stepSize);

        // INCREMENT TIME
        Time += stepSize;
```

Performs the integration of system states and iterates the time forward one time step.

```
        if (checkOverlap()){
            // IF DISKS OVERLAP, EXIT
            cerr << "Disks overlap... exiting" << endl;
            exit(1);
        }
```

If the distance between the centers of any of the pairs of disks is less than one disk diameter, something has gone wrong in the computation, so exit the program.

```
        if ( writeVACF ){
            // IF writeVACF WAS SET TO NON-ZERO VALUE,
            if (Time > vacfSettleTime + vacfCaptureTime){
                // IF TIME EXCEEDS THE ALLOTTED TIME FOR DATA ↩
                   COLLECTION, OPEN OUTPUT FILE FOR WRITING ↩
                   VELOCITY AUTOCORRELATION FUNCTION DATA
                char *vacfFile = new char [256];
                strcat( vacfFile, "vacf_" );
                strcat( vacfFile, outFile );
                vacfData.open(vacfFile,ios::out | ios::binary);

                double tau;  // time for velocity ↩
                   autocorrelation function computation
                double vacf; // value of velocity ↩
                   autocorrelation function computation at time ↩
                   tau

                double deltaTau = (double)(modVacf)*stepSize; //↩
                    time-delay between successive samples
                unsigned long numTimeSteps = velocities.size() /↩
                    2; // number of time steps to process; / 2 ↩
                   to avoid index out-of-bounds
                // INITIALIZE tau
                tau = 0.;

                for (ii = 0; ii < numTimeSteps; ii++){
                    // COMPUTE VELOCITY AUTOCORRELATION FUNCTION↩
                         VALUE AT CURRENT TIME tau
                    vacf = computeVacf(ii);

                    // WRITE OUT TIME AND VELOCITY AUTOCORRELATION FUNCTION VALUE
                    vacfData.write((char *) &tau, sizeof(double));
                    vacfData.write((char *) &vacf, sizeof(double));

                    // INCREMENT tau
                    tau += deltaTau;
                } // END for (ii = 0; ii < numTimeSteps; ii++)

                // CLOSE OUTPUT FILE
                vacfData.close();

                // TOGGLE writeVACF TO ZERO TO ENSURE "if" CONDITION IS NOT ENTERED AGAIN
                writeVACF = 0;

                // CLEAR velocities VECTOR (VALUES NO LONGER NEEDED)
                velocities.clear(); //clear velocity vector

            } // END if (Time > vacfSettleTime + vacfCaptureTime)

        } // END if ( writeVACF )
```

If the current time exceeds the period for data acquisition for the velocity autocorrelation computation and the input **writeVACF** is not 0, open a file for writing the output data, set the delta time step to evaluate the autocorrelation function at (**deltaTau**) and the number of time steps to evaluate the autocorrelation for (**numTimeSteps**). Loop over time step, computing the velocity autocorrelation function (**computeVacf()**, see Subsection *computeVacf*), then write out the time and value of the autocorrelation. After exiting the loop, close the output file, toggle the **writeVACF** flag to avoid entering this code branch again, and clear memory allocated to the vector **velocities** for other use.

```
    } // END for (i = 0; i < nSteps; i++)
```

Ends main simulation loop over time.

The next lines are a finalize method that performs wrap-up and file closure.

```
   // FINALIZE
    // OUTPUT MEAN FREE TIME (AVERAGE TIME BETWEEN SUCCESSIVE ↩
       COLLISIONS FOR AN INDIVIDUAL DISK)
    cout << "Mean free time = " << (double)(nDisks)*Time / ((↩
       double)(i_coll)) << endl;
```

Computes and prints out the mean free time (*MFT*), which is the average time in-between collisions for an individual disk (t is simulation end time):

$$MFT = \frac{Nt}{i_{\text{coll}}}.$$

```
    //COMPUTE LOCALIZATION WIDTH AND WRITE OUT TO FILE
    computeLocalizationWidth();
    for (ii = 0; ii < nlya; ii++){
        outData.write((char *) &stats.Wn.at(ii), sizeof(double))↩
           ;
    }
```

Call to compute localization measure (Subsection *computeLocalizationWidth*) and write out to file with handle **outData**.

```
    // WRITE DATA OF AVERAGE COSINE OF THE ANGLE BETWEEN ↩
       POSITION AND MOMENTUM CONTRIBUTIONS TO EACH TANGENT ↩
       VECTOR
    for (ii = 0; ii < nlya; ii++){
        outData.write((char *) &stats.avgCosTheta.at(ii), sizeof↩
           (double));
    } // END for (ii = 0; ii < nlya; ii++)
```

Write out data for average cosine of the angle between position and momentum contributions for each tangent vector to file with handle **outData**.

```
    // CLOSE MAIN OUTPUT FILE
    outData.close();

    // IF INDICATED FROM INPUT FILE, CLOSE OPTIONAL OUTPUT FILES
    if (computeBatchAvg){
        batchData.close();
    } // END if (computeBatchAvg)

    if (writeAlpha){
        alphaData.close();
    } // END if (writeAlpha)
```

Close output files (if opened).

```
  // IF INDICATED FROM INPUT FILE, COMPUTE COVARIANT LYAPUNOV ↩
     VECTORS
  // USING GINELLI'S METHOD
  if (processCLVs){
    computeCLVs();
  }
```

If covariant Lyapunov vectors are to be computed, enter the **computeCLVs()** routine.

```
    // EXIT NORMALLY
    return 0;

} // END int main (int argc, const char * argv[])
```

End the routine and exit normally.

B.6.2 utilities.cpp

hardStep

This routine is responsible for the forward time evolution of the reference trajectory and tangent vectors.

```
//
//  FUNCTION:  hardStep
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function performs one−time step integration/iteration of ↩
    hard disk system and stores velocity values for velocity ↩
    autocorrelation processing.
//
//  INPUTS:
//   t − time step (updated throughout the call; interpreted as ↩
    the time remaining on the current step)
//
//  OUTPUTS:
//   (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   Time
//   vacfSettleTime
//   vacfCaptureTime
//   modVacf
//   noColl
//   maxFlight
//  DIM
//   nDisks
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//   iVacf  (incremented)
//   i_coll (incremented)
//
//  REVISION HISTORY:
//   Dinius, J.       Modified from C. Dellago               ↩
    10/08/11
//   Dinius, J.       Comments added for v1.0 release        ↩
    05/27/13
//   Dinius, J.       Cleaned up for dissertation release    ↩
    11/25/13
//
void hardStep(double t)
{
     // INCREMENT iVacf
     ++iVacf;
```

Increment counter for velocity autocorrelation function data storage and processing.

```
    if ( writeVACF & (Time > vacfSettleTime) & (Time < ↩
       vacfSettleTime + vacfCaptureTime) & ( iVacf % modVacf == ↩
       0)){
        // IF WRITING OUT VELOCITY AUTOCORRELATION FUNCTION DATA↩
           AND THE CURRENT TIME (AND INDEX COUNT) IS IN THE ↩
           VALID CAPTURE WINDOW,

        // STORE CURRENT VELOCITIES IN VECTOR
        vector<double> singleStepVelocity;
        for (int j = DIM*nDisks; j < 2*DIM*nDisks; j++){
            singleStepVelocity.push_back(y[j]);
        }
        // PUT CURRENT VELOCITY VECTOR AT THE END OF velocities
        velocities.push_back(singleStepVelocity);
        singleStepVelocity.clear();
    }
```

If the flag to output data (**writeVACF**) is set to a non-zero value, **Time** is in an interval specified by inputs **vacfSettleTime** and **vacfCaptureTime**), and the counter **iVacf** modulo the input **modVacf** is 0, store the disk velocities at each time step (**singleStepVelocity**) and then push into the vector of these individual time step vectors (**velocities**).

```
    // DECREMENT TIME t UNTIL IT IS EQUAL TO 0
    while (t > 0.0){
        // FIND NEXT COLLISION AND TUMBLE
        c = nextColl();
        tmb = nextTumble();

        // FIND MINIMUM TIME OF NEXT EVENT RELATIVE TO THE ↩
           CURRENT TIME
        tm  = fmin(c.time, tmb.time);
```

While the time remaining on the current frame is greater than 0, find out when the next discrete event occurs. The times **c.time** and **tmb.time** represent the times of next collision and next tumble, respectively. The next discrete event occurs at the minimum of these two times.

```
        if (t < tm){
            // IF t IS SMALLER THAN THE TIMES OF NEXT TUMBLE AND↩
                COLLISION, PROPAGATE THE POSITIONS FORWARD WITH ↩
                AN EULER INTEGRATION STEP
            freeFlight(t);

            // ENFORCE PERIODIC BOUNDARY CONDITIONS
            boxSet();

            // UPDATE COLLISION TIMES FOR ALL DISKS
            updateTimes(t, 0, noColl, noColl, 0);

            // SET t TO ZERO, INDICATING CURRENT CALL TO ↩
                hardStep IS COMPLETE
            t = 0.0;
        } // END if (t < tm)
```

If the time remaining on the current frame (the value of the variable **t** is time remaining) is less than the minimum of the next collision time and the next tumble time, perform the free-flight dynamics integration, enforce the periodic boundary conditions (**boxSet()**, Subsection *boxSet*) and compute the next collision times (relative to the current time, NOT an absolute time) and partners for all of the disks. Tumble time is not sensitive to disk position, and therefore tumble times do not need to be updated. Setting **t = 0.0** provides the exit condition to return control to the calling routine (in this case, the **main()** routine).

```
        else if (c.time < tmb.time) {
            // COLLISION EVENT IS NEXT

            // PROPAGATE DISK POSITIONS FORWARD BY TIME OF NEXT ↩
                COLLISION
            freeFlight(c.time);
            t -= c.time;

            // ENFORCE PERIODIC BOUNDARY CONDITIONS
            boxSet();

            if (c.partner != maxFlight){
                // IF THE NEXT COLLISION EVENT OCCURS BETWEEN ↩
                    TWO DISKS (NOT A MAX-FLIGHT CONDITION), ↩
                    UPDATE PHASE SPACE AND TANGENT VECTORS AFTER ↩
```

```
                        COLLISION
                    collision(c.index,c.partner);
                } // END if (c.partner != maxFlight)

                // INCREMENT COLLISION COUNTER
                ++i_coll;

                // UPDATE COLLISION TIMES FOR ALL DISKS
                updateTimes(c.time, 0, c.index, c.partner, 0);
            } // END else if (c.time < tmb.time)
```

If a collision is next, perform the free-flight dynamics up to the next collision time **c.time**, decrement the time remaining on the frame by **c.time** and enforce the periodic boundary conditions.

```
                if (c.partner != maxFlight){
                    // IF THE NEXT COLLISION EVENT OCCURS BETWEEN ↩
                       TWO DISKS (NOT A MAX-FLIGHT CONDITION), ↩
                       UPDATE PHASE SPACE AND TANGENT VECTORS AFTER ↩
                       COLLISION
                    collision(c.index,c.partner);

                    // INCREMENT COLLISION COUNTER
                    ++i_coll;
                } // END if (c.partner != maxFlight)
```

If an actual collision occurs, not a maximum flight condition, call **collision()** which applies the collision rule to the reference trajectory and the tangent vectors (see Subsection *collision*) and increment the collision counter.

```
                // UPDATE COLLISION TIMES FOR ALL DISKS
                updateTimes(c.time, 0, c.index, c.partner, 0);
            } // END else if (c.time < tmb.time)
```

Compute the next collision times relative to the current time and the partners for all of the disks.

```
    else {
            // TUMBLE EVENT IS NEXT

            // PROPAGATE DISK POSITIONS FORWARD BY TIME OF NEXT ↩
               TUMBLE
            freeFlight(tmb.time);
            t -= tmb.time;

            // ENFORCE PERIODIC BOUNDARY CONDITIONS
            boxSet();

            // TUMBLE DISK
            tumble(tmb.index);

            // UPDATE COLLISION TIMES FOR ALL DISKS
            updateTimes(tmb.time, 1, noColl, noColl, tmb.index);
        } // END else

    } // while (t > 0.0)
```

This is the tumble condition. If a tumble occurs next sequentially, perform the free-flight dynamics up to the next tumble time **tmb.time**, decrement the time remaining on the frame by **tmb.time**, enforce the periodic boundary conditions, tumble the disk with index defined by **tmb.index**, and update the next tumble time for the disk with index **tmb.index** by calling **updateTimes()** (see Subsection *updateTimes*).

```

    }
    return;
}
```

Exits the routine.

nextColl

This routine finds the next collision time and the participating disks.

```
//
//  FUNCTION: nextColl
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
//   Function returns parameters for the next collision based ↩
    upon the current sim time.  The next collision is determined ↩
    through a simple minimizing search.
//
//   INPUTS:
//   (none)
//
//   OUTPUTS:
//   c - COLL structure containing the next colliding disk's ↩
    index, partner and the associated collision time (relative to↩
     current sim time)
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   collArray
//   nDisks
//   bigTime
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   (none)
//
//   REVISION HISTORY:
//   Dinius, J.       Modified from C. Dellago             ↩
    10/08/11
//   Dinius, J.       Comments added for v1.0 release      ↩
    05/27/13
//   Dinius, J.       Cleaned up for dissertation release  ↩
    11/25/13
//
COLL nextColl(void)
{
    int i;             // loop index
    COLL c;            // instance of COLL containing next ↩
       collision states (index, partner and time)
    c.time = bigTime; // initialize next collision time to ↩
       something large
```

Initialize the minimizing search routine with a large time.

```
    for (i = 0; i < nDisks; i++) {
        // FOR ALL DISKS
        if ( (collArray[i].time < c.time) && (collArray[i].time ↩
           > 0.0) ){
            // IF CURRENT VALUE FOR NEXT COLLISION TIME IS ↩
               LARGER THAN THE VALUE CONTAINED IN collArray (↩
               WITH INDEX i) AND THAT VALUE IS LARGER THAN 0.0 (↩
               INDICATING THE COLLISION TIME IS VALID), THE NEXT↩
                COLLISION TIME, INDEX, AND PARTNER ARE UPDATED ↩
               WITH VALUES FROM collArray (WITH INDEX i)
            c = collArray[i];

        } // END for (i = 0; i < nDisks; i++)

    } // END if ( (collArray[i].time < c.time) && (collArray[i].↩
         time > 0.0) )

    // RETURN GLOBAL MINIMIZER OF NEXT COLLISION TIME, AND THE ↩
       ASSOCIATED INDEX AND PARTNER
    return c;
}
```

Performs a simple minimization routine to find the next collision time and the participating disks, stored in the instance **c** of the class **COLL**.

nextTumble

This routine finds the next tumble time and the disk index of the tumbling disk.

```
//
//  FUNCTION:  nextTumble
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function returns the disk index and time (relative to the ↩
    current sim time) of the next tumble.  The next tumble is ↩
    determined through a simple minimizing search.
//
//  INPUTS:
//   (none)
//
//  OUTPUTS:
```

```
//   c - TUMBLE structure containing the next tumbling disk's ↩
    index and the associated tumble time (relative to current sim↩
     time)
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   tumbleArray
//   nDisks
//   bigTime
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   (none)
//
//   REVISION HISTORY:
//   Dinius, J.        Modified from C. Dellago              ↩
    10/08/11
//   Dinius, J.        Comments added for v1.0 release       ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release   ↩
    11/25/13
//
TUMBLE nextTumble(void)
{
    int i;              // loop index
    TUMBLE c;           // instance of TUMBLE containing next ↩
       tumble states (index and time)
    c.time = bigTime; // initialize next collision time to ↩
       something large
```

Initialize the minimizing search routine with a large number.

```
    for (i = 0; i < nDisks; i++) {
        // FOR ALL DISKS
        if ( (tumbleArray[i].time < c.time) && (tumbleArray[i].↩
           time > 0.0) ){
            // IF CURRENT VALUE FOR NEXT COLLISION TIME IS ↩
               LARGER THAN THE VALUE CONTAINED IN tumbleArray (↩
               WITH INDEX i) AND THAT VALUE IS LARGER THAN 0.0 (↩
               INDICATING THE TUMBLE TIME IS VALID), THE NEXT ↩
               TUMBLE TIME AND INDEX ARE UPDATED WITH VALUES ↩
               FROM tumbleArray (WITH INDEX i)
            c = tumbleArray[i];

        } // END if ( (tumbleArray[i].time < c.time) && (↩
            tumbleArray[i].time > 0.0) )

    } // END for (i = 0; i < nDisks; i++)

    // RETURN GLOBAL MINIMIZER OF NEXT TUMBLE TIME, AND THE ↩
        ASSOCIATED INDEX
    return c;
```

Performs a simple minimization routine to find the next tumble time and the participating disk, stored in the instance **c** of the class **TUMBLE**.

freeFlight

Performs simple integration of the free flight portion of the dynamics. This routine does not take into account periodic boundary conditions (see Subsection *boxSet*).

```
//
//  FUNCTION:  freeFlight
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function updates state and tangent space offset vectors ↩
    during the free-flight portion of the dynamics by an Euler ↩
    integration step.
//
//  INPUTS:
//   dt - Euler integration timestep
//
//  OUTPUTS:
//   (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   nlya
//  DIM
//   nDisks
//   phaseDim
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//   y (positions integrated forward in time, velocities left ↩
    unchanged)
```

```
//
//   REVISION HISTORY:
//   Dinius, J.        Modified from C. Dellago             10/08/11
//   Dinius, J.        Comments added for v1.0 release      05/27/13
//   Dinius, J.        Cleaned up for dissertation release  11/25/13
//
void freeFlight(double dt)
{
    int i,j; // loop indices
    for (i = 0; i < nlya+1; i++){
        // FOR EACH OF THE COMBINED STATE AND TANGENT VECTORS
        for (j = 0; j < DIM*nDisks; j++){
            // UPDATE THE POSITION COMPONENTS WITH THE TIME-INTEGRATED MOMENTUM COMPONENTS
            y[i*phaseDim+j] += y[DIM*nDisks+i*phaseDim+j]*dt;
        } // END for (j = 0; j < DIM*nDisks; j++)

    } // END for (i = 0; i < nlya+1; i++)
```

This is the Euler integration of position states (which is exact for the problem of motion in the absence of an external force)

$$y(t) = y(0) + \frac{p}{m}dt.$$

```
    return;
}
```

Exits the routine.

```
\indent \par This routine enforces periodic boundary conditions for disks in the box.
//
//   FUNCTION:  boxSet
//   MODULE:    diskDynamics
//
```

```
//  DESCRIPTION:
//  Function applies periodic boundary conditions to the ↩
    position components of the state vector to ensure all disks ↩
    remain in the simulation box at all times.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  boxSize
//  nDisks
//  DIM
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  y (positions updated according to periodic boundary ↩
    conditions)
//
//  REVISION HISTORY:
//  Dinius, J.       Modified from C. Dellago               ↩
    10/08/11
//  Dinius, J.       Comments added for v1.0 release        ↩
    05/27/13
//  Dinius, J.       Cleaned up for dissertation release    ↩
    11/25/13
//
void boxSet(void)
{
    int i,j; // loop indices

    for (i = 0; i < nDisks; i++){
        // FOR ALL DISKS
```

For all of the disks,

```
        for (j = 0; j < DIM; j++){
            // FOR EACH SPATIAL COMPONENT (X AND Y)
```

For both spatial dimensions,

```
            while ( y[i*DIM+j] < 0.0 ){
                // WHILE THE PARTICLE POSITION IS LEFT (X-DIR) ↩
                   OR BELOW (Y-DIR) THE BOX SIZE IN THE CURRENT ↩
                   DIMENSION (boxSize[j]), ADD boxSize[j] UNTIL ↩
                   THE VALUE IS POSITIVE AND LESS THAN boxSize[j↩
                   ]
                y[i*DIM+j] += boxSize[j];
            } // END while ( y[i*DIM+j] < 0.0 )
```

If a disk moves left of the box ($x < 0$), or below the box ($y < 0$), perform the modulus operation until $x, y \geq 0$.

```
            while ( y[i*DIM+j] > boxSize[j] ){
                // WHILE THE PARTICLE POSITION IS RIGHT (X-DIR) ↩
                   OR ABOVE (Y-DIR) THE BOX SIZE IN THE CURRENT ↩
                   DIMENSION (boxSize[j]), SUBTRACT boxSize[j] ↩
                   UNTIL THE VALUE IS POSITIVE AND LESS THAN ↩
                   boxSize[j]
                y[i*DIM+j] -= boxSize[j];
            } // END while ( y[i*DIM+j] > boxSize[j] )
```

If a disk moves right of the box ($x > 0$), or above the box ($y > 0$), perform the modulus operation until $x \leq L_x$ and $y \leq L_y$.

```
        }// END for (j = 0; j < DIM; j++)

    } // END for (i = 0; i < nDisks; i++)

    return;
}
```

Exits the routine.

updateTimes

This routine updates the next collision and tumble times (relative to the current simulation time, **Time**, for all of the disks.

```
//
//  FUNCTION:  updateTimes
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function updates the collision and tumble times of all disks↩
    and stores them in the collArray and tumbleArray structures,↩
    respectively.  This routine is called to update the ↩
    collision times and partners for each disk after a collision ↩
    has occurred.
//
//  INPUTS:
//  tLast    - time states have been propagated forward since ↩
    the last call
//  isTmbl   - flag indicating whether or not a tumble is to ↩
    occur in the
//              current call of this function
//  d1       - index of first disk undergoing collision
//  d2       - index of second disk undergoing collision
//  tmblIndx - index of disk undergoing tumble
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  nDisks
//  maxFlight
//  noColl
//  maxFlightDblArray
//  dtTmbl
//  DIM
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  collArray   (updated values for collision partners and ↩
    associated times for all nDisks disks)
//  tumbleArray (updated values for tumble times for all nDisks ↩
    disks)
//
//  REVISION HISTORY:
//  Dinius, J.       Modified from C. Dellago               ↩
    10/08/11
//  Dinius, J.       Comments added for v1.0 release        ↩
    05/27/13
//  Dinius, J.       Cleaned up for dissertation release    ↩
```

```
    11/25/13
//
void updateTimes(double tLast, int isTmbl, int d1, int d2, int ↩
    tmblIndx)
{
    int i,j;    // loop indices
    double t1; // temporary variable for comparisons

    // UPDATE COLLISION AND TUMBLE TIMES FROM THOSE COMPUTED ↩
       DURING LAST FUNCTION CALL
    for (i = 0; i < nDisks; i++){
        tumbleArray[i].time -= tLast;
        collArray[i].time   -= tLast;
    } // END for (i = 0; i < nDisks; i++)
```

Move all of the collision and tumble times up by the previous update time.

```
    if (isTmbl == 0){
        // IF THE CURRENT EVENT IS NOT A TUMBLE, UPDATE ↩
           COLLISION TIMES (RELATIVE TO CURRENT SIM TIME)
```

If the next event is not a tumble, but rather a collision

```
    if ( d1 != noColl ){
            // IF A COLLISION HAS OCCURRED (noColl IS USED TO ↩
               UPDATE COLLISION TIMES AFTER A FREE-FLIGHT)
```

If the last time of interest was associated with a collision (not just free-flight) for disk with index **d1**

```
        for (i = 0; i < nDisks; i++){
                // FOR ALL DISKS
```

Loop over all disks

```
            if ( (i == d1) || // IF i IS EQUAL TO d1
                    (collArray[i].partner == d1) || // OR THE ↩
                       COLLISION PARTNER OF i IS d1
                    (i == d2) || // OR i IS EQUAL TO d2
                    (collArray[i].partner == d2) ) { // OR THE ↩
                       COLLISION PARTNER OF i IS d2
```

If the current index **i** is the same as either of the two disks that were involved in the last collision,

```
                // IF ONE OF THE ABOVE CONDITIONS, DISK i WAS ↩
                   INVOLVED IN THE LAST COLLISION

                    // RESET THE TIME OF NEXT COLLISION TO A ↩
                       LARGE NUMBER AND SET THE PARTNER TO BE ↩
                       maxFlight
                    collArray[i].time    = bigTime;
                    collArray[i].partner = maxFlight;
```

Reset the collision time of previous colliding disks to something big and set the partner to **maxFlight**. The computation will most likely yield a new collision time and partner for each of **d1**, **d2** (the colliding disks).

```
                // NOW SEARCH FOR THE MINIMUM TIME AND THE ↩
                   ASSOCIATED PARTNER OF THE NEXT COLLISION FOR ↩
                   DISK i
                    for (j = 0; j < nDisks; j++){
                        // LOOP OVER ALL DISKS
                        if ( i != j ){
```

Compare the **i**th disk against all other disks

```
                        // IF j IS DIFFERENT THAN i, TRY TO ↩
                           IDENTIFY IF A COLLISION BETWEEN DISKS↩
                            j AND i IS POSSIBLE COMPUTE ↩
                           COLLISION TIME BETWEEN DISKS j AND i
                            t1 = binTime(i,j);
```

Compute the collision time of the **i**th and **j**th disks

```
                    if (t1 > 0.0) {
                        // IF COLLISION TIME IS POSITIVE↩
                           , THEN THE COLLISION IS A ↩
                           VALID ONE
                        if (t1 < collArray[i].time) {
                            // IF THE COLLISION TIME ↩
                               BETWEEN DISKS j AND i IS ↩
                               SMALLER THAN THE CURRENT ↩
                               CALCULATED COLLISION TIME↩
                                FOR DISK i, UPDATE THE ↩
                               TIME AND PARTNER FOR DISK↩
                                i
                            collArray[i].time = t1;
                            collArray[i].partner = j;
                        } // END if (t1 < collArray[i].↩
                           time)

                        if (t1 < collArray[j].time) {
                            // IF THE COLLISION TIME ↩
                               BETWEEN DISKS j AND i IS ↩
                               SMALLER THAN THE CURRENT ↩
                               CALCULATED COLLISION TIME↩
                                FOR DISK j, UPDATE THE ↩
                               TIME AND PARTNER FOR DISK↩
                                j
                            collArray[j].time = t1;
                            collArray[j].partner = i;
                        } // END if (t1 < collArray[j].↩
                           time)

                    } // END if (t1 > 0.0)
```

If a collision is geometrically possible (see Subsection *binTime*) and would occur sooner than the current computed collision time for the current disk(s), update the collision time(s).

```
                } // END if ( i != j )

            } // END for (j = 0; j < nDisks; j++)

        } // END if ( (i == d1) || ...

    } // END for (i = 0; i < nDisks; i++)
```

```

        } // END if ( d1 != noColl )

    } // END if (isTmbl == 0)
```

```
    else {
        // THE LAST EVENT WAS A TUMBLE, UPDATE THE TUMBLE AND ↩
           COLLISION TIMES (RELATIVE TO CURRENT SIM TIME) FOR ↩
           THE DISK WITH INDEX tmblIndx

        // UPDATE THE NEXT TUMBLE TIME
        tumbleArray[tmblIndx].time = dtTmbl*ranf();
```

Perform a uniform random draw on the interval [0,**dtTmbl**] for the next tumble time of disk with index **tmblindex**.

```
        for (j = 0; j < nDisks; j++){
            // FOR ALL DISKS
            if ( tmblIndx != j ){
                // FIND COLLISION TIME OF NEXT COLLISION ↩
                   INVOLVING DISKS j AND tmblIndx
                t1 = binTime(tmblIndx,j);

                if (t1 > 0.0) {
                    // IF COLLISION TIME IS POSITIVE, THEN THE ↩
                       COLLISION IS A VALID ONE
                    if (t1 < collArray[tmblIndx].time) {
                        // IF THE COLLISION TIME BETWEEN DISKS j↩
                           AND tmblIndx IS SMALLER THAN THE ↩
                          CURRENT CALCULATED COLLISION TIME FOR↩
                           DISK tmblIndx, UPDATE THE TIME AND ↩
                          PARTNER FOR PARTICLE tmblIndx
                        collArray[tmblIndx].time = t1;
                        collArray[tmblIndx].partner = j;
                    } // END if (t1 < collArray[tmblIndx].time)

                    if (t1 < collArray[j].time) {
                        // IF THE COLLISION TIME BETWEEN DISKS j↩
                           AND tmblIndx IS SMALLER THAN THE ↩
                          CURRENT CALCULATED COLLISION TIME FOR↩
```

```
                                    DISK j , UPDATE THE TIME AND PARTNER ↩
                                   FOR PARTICLE j
                               collArray[j].time = t1;
                               collArray[j].partner = tmblIndx;
                           } // END if (t1 < collArray[j].time)

                       } // END if (t1 > 0.0)

                   } // if ( tmblIndx != j )

               } // for (j = 0; j < nDisks; j++)

           } // END else
```

Compute the new collision partner and times for the disk with index **tmblIndx**

```
    // CHECK FOR MAX FLIGHT CONDITION
    for (i = 0; i < nDisks; i++){
        // FOR ALL DISKS
        for (j = 0; j < DIM; j++){
            // FOR BOTH SPATIAL DIMENSIONS
            if ( fabs(collArray[i].time * y[DIM*(i+nDisks)+j]) >↩
                maxFlightDblArray[j] ){
                // IF THE DISTANCE TRAVELED IN EITHER SPATIAL ↩
                   DIRECTION BEFORE A VALID COLLISION INVOLVING ↩
                   DISK i EXCEEDS THE MAXIMUM FLIGHT CONDITION ↩
                   maxFlight , THEN SET THE COLLISION TIME AND ↩
                   PARTNER FOR DISK i ACCORDINGLY
                collArray[i].time    = fabs(maxFlightDblArray[j]↩
                    / y[DIM*(i+nDisks)+j]);
                collArray[i].partner = maxFlight;
            }// END if ( fabs(collArray[i].time * y[DIM*(i+↩
               nDisks)+j]) > maxFlightDblArray[j] )

        } // END for (j = 0; j < DIM; j++)

    } // END for (i = 0; i < nDisks; i++)
```

For all disks and in both spatial dimensions, if the absolute value of the product of the next collision time and the disk velocity in the *x*- and/or *y*-directions exceeds the maximum flight condition, update the collision time to reflect that the maximum flight condition occurs before the next possible collision.

```
    return;
}
```

Ends the routine.

binTime

This routine computes collision times for a pair of particles.

```
//
//  FUNCTION:  binTime
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function finds time when particles i1 and j1 undergo ↩
    collision.  Function returns −1 if collision between the two ↩
    disks is impossible.
//
//  INPUTS:
//   i1 − index of first disk
//   j1 − index of second disk
//
//  OUTPUTS:
//   t1 − collision time relative to current sim time (−1 if no ↩
    valid collision was found)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  DIM
//   y
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//   yi (dummy variable)
//   yj (dummy variable)
//   vi (dummy variable)
//   vj (dummy variable)
//   dq1 (dummy variable, used to store shortest distance between↩
      disk centers)
//
//  REVISION HISTORY:
```

```
//   Dinius, J.        Modified from C. Dellago              ↩
    10/08/11
//   Dinius, J.        Comments added for v1.0 release       ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release   ↩
    11/25/13
//
double binTime(int i1, int j1)
{
    int ii;     // loop index
    double t1; // return value

    // DECLARE VECTORS FOR CONVENIENCE
    for (ii = 0; ii < DIM; ii++){
        yi[ii] = y[DIM*i1+ii]; // position of ith particle
        yj[ii] = y[DIM*j1+ii]; // position of jth particle
        vi[ii] = y[DIM*(i1+nDisks)+ii]; // momentum of ith ↩
            particle
        vj[ii] = y[DIM*(j1+nDisks)+ii]; // momentum of jth ↩
            particle
    } // END for (ii = 0; ii < DIM; ii++)
```

Declare temporary variables for ease of processing later in routine.

```
    // FIND SHORTEST DISTANCE BETWEEN DISKS i AND j CENTERS-OF-↩
       MASS
    image(yi,yj);
```

Find the shortest vector connecting the centers-of-mass of the **i1**th and **j1**th disks.

The remainder of the routine solves for the collision time t according to the relation:

$$\left\|\mathbf{q} + \frac{\mathbf{p}}{m}t\right\|_2^2 = \sigma^2 \qquad \text{(B.6.2.0.2)}$$

where $\mathbf{q}, \mathbf{p}$ are the relative position and momentum of the **i1**th and **j1**th disks. See [16] for more details. This results in the quadratic equation

$$\begin{aligned} t &= \frac{-\mathbf{q}\cdot\mathbf{p} \pm \sqrt{(\mathbf{q}\cdot\mathbf{p})^2 - (\mathbf{p}\cdot\mathbf{p})(\mathbf{q}\cdot\mathbf{q} - \sigma^2)}}{\frac{\mathbf{p}\cdot\mathbf{p}}{m}} \\ &\overset{\text{def}}{=} \frac{-\Delta \pm \sqrt{d}}{\frac{p^2}{m}} \end{aligned}$$

```
    // INITIALIZE delta
    double delta = 0.0;

    // COMPUTE delta EQUALS DOT PRODUCT OF RELATIVE POSITION ↩
       VECTOR WITH RELATIVE MOMENTUM VECTOR
    for (ii = 0; ii < DIM; ii++){
        delta += dq1[ii] * (vi[ii]−vj[ii]);
    } // END for (ii = 0; ii < DIM; ii++)
```

Compute $\Delta = \mathbf{q} \cdot \mathbf{p}$.

```
    if (delta>0) {
        // IF THE DOT PRODUCT IS POSITIVE, THEN DISKS ARE MOVING↩
           AWAY FROM EACH OTHER AND NO COLLISION IS POSSIBLE
        t1 = −1.0;
        return t1;
    } // END if (delta >0)
```

If $\Delta > 0$, the disks are moving away from each other, and won't collide, so return -1 for the collision time (this indicates an invalid collision).

```
    // INITIALIZE dq2 and dv2
    double dq2 = 0.0;
    double dv2 = 0.0;

    // COMPUTE dq2 EQUALS DOT PRODUCT OF RELATIVE POSITION ↩
       VECTOR WITH ITSELF AND dv2 EQUALS DOT PRODUCT OF RELATIVE↩
        MOMENTUM VECTOR
    // WITH ITSELF
    for (ii = 0; ii < DIM; ii++){
        dq2 += pow(dq1[ii],2.0);
        dv2 += pow( (vi[ii]−vj[ii]), 2.0 );
    } // END for (ii = 0; ii < DIM; ii++)
```

Compute **dq2** $= \mathbf{q} \cdot \mathbf{q}$ and **dv2** $= \frac{\mathbf{p} \cdot \mathbf{p}}{m} = \mathbf{p} \cdot \mathbf{p}$, since $m = 1$ by assumption.

```
    // COMPUTE DISCRIMINANT IN QUADRATIC EQUATION FOR COLLISION ↩
       TIME; 1.0 FACTOR COMES FROM ASSUMPTION THAT EACH DISK ↩
       DIAMETER IS 1.0
    double discr = pow(delta,2.0) - dv2*(dq2 - 1.0);
```

Compute the discriminant in the quadratic formula $d = (\mathbf{q}\cdot\mathbf{p})^2 - (\mathbf{p}\cdot\mathbf{p})(\mathbf{q}\cdot\mathbf{q} - \sigma^2)$. It is assumed that $\sigma = 1$.

```
    if (discr < 0){
        // IF THE DISCRIMINANT IS LESS THAN ZERO, THEN THE ↩
           SQUARE ROOT OF THE DISCRIMINANT WILL BE COMPLEX ↩
           INDICATING THAT NO COLLISION IS POSSIBLE
        t1 = -1.0;
        return t1;
    } // END if (discr < 0)
```

If the roots are not real, then a valid collision cannot be computed, so return -1 (this indicates an invalid collision).

```
    // COMPUTE COLLISION TIME OF DISKS i AND j AS THE "-" ↩
       SOLUTION OF QUADRATIC EQUATION (IT IS THE SOONER OF THE ↩
       TWO TIMES)
    t1 = (-delta - sqrt(discr)) / dv2;

    return t1;
}
```

If the roots are real, a valid collision time has been computed. Return the value computed for "$-\sqrt{d}$" in the quadratic formula, since this is the sooner of the two collisions.

image

This routine computes the shortest vector between two disks.

```
//
//   FUNCTION:  image
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
//    Function outputs vector of shortest distance between two xy ↩
        positions in the simulation box
```

```
//
//   INPUTS:
//   y1 - xy position of first disk
//   y2 - xy position of second disk
//
//   OUTPUTS:
//   (none)
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   DIM
//   boxSize
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   dq1 (computed vector of shortest length connecting positions↩
     y1 and y2 incorporating periodic boundary
//        conditions)
//
//   REVISION HISTORY:
//   Dinius, J.       Modified from C. Dellago              ↩
    10/08/11
//   Dinius, J.       Comments added for v1.0 release       ↩
    05/27/13
//   Dinius, J.       Cleaned up for dissertation release   ↩
    11/25/13
//
void image(double *y1,double *y2)
{
    int i; // loop index
    for (i = 0; i < DIM; i++){
        // FOR EACH SPATIAL DIRECTION, COMPUTE THE RELATIVE ↩
           POSITION COMPONENT ALONG THE CURRENT DIRECTION
        dq1[i] = y1[i] - y2[i];
        // THE SHORTEST DISTANCE BETWEEN TWO DISKS IN THE BOX ↩
           CAN BE NO GREATER THAN 1/2 THE BOX LENGTH IN EITHER ↩
           DIRECTION (X OR Y)
        if (dq1[i] > boxSize[i] / 2.0){
            // POSITIVE CHECK
            dq1[i] -= boxSize[i];
        } // END if (dq1[i] > boxSize[i] / 2.0)

        else if (dq1[i] < -boxSize[i] / 2.0){
            // NEGATIVE CHECK
            dq1[i] += boxSize[i];
        } // END else if (dq1[i] < -boxSize[i] / 2.0)

    } // END for (i = 0; i < DIM; i++)

    return;
}
```

If the component-wise difference of the positions **y1**,**y2** of the two disks exceeds half of the box length along that component (x or y) then a shorter path (using periodic boundary conditions) between the two disks exists, so update the value of **dq1** with that updated relative position vector.

collision

This routine computes the map for the reference trajectory and tangent vectors after a collision. See Chapter 3 for more details.

```
//
//  FUNCTION:  collision
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function updates the post-collision state and tangent ↩
    vectors after a collision has occurred.
//
//   Reference: Dinius, J. and J. Lega. TBD
//
//  INPUTS:
//   i - index of first disk involved in collision
//   j - index of first disk involved in collision
//
//  OUTPUTS:
//   (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   phaseDim
//   dq1
//  DIM
//   nDisks
//   beta
//   alpha
```

```
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  y    (velocity components of state vector and position and momentum components of tangent vectors are updated
//        after collision)
//  yi   (dummy variable)
//  yj   (dummy variable)
//  v    (dummy variable)
//  vq   (updated dot product of relative momentum and relative position of disks undergoing collision)
//  vv   (updated dot product of relative momentum with itself of disks undergoing collision)
//  dq   (dummy variable)
//  dv   (dummy variable)
//  dqc  (dummy variable)
//  dq1  (dummy variable, used to store shortest distance between disk centers)
//
//  REVISION HISTORY:
//  Dinius, J.       Modified from C. Dellago                10/08/11
//  Dinius, J.       Comments added for v1.0 release         05/27/13
//  Dinius, J.       Cleaned up for dissertation release     11/25/13
//
void collision(int i, int j)
{
    int ii,jj; // loop indices
    int pInd;  // index into tangent vector components
    double dqq, dvq, dqcv, dtc, velOff, dvv; // temporary variables for tangent vector updates

    for (ii = 0; ii < DIM; ii++){
        yi[ii] = y[DIM*i+ii]; // position vector of disk i
        yj[ii] = y[DIM*j+ii]; // position vector of disk j
        v[ii]  = y[DIM*(nDisks+j)+ii] - y[DIM*(nDisks+i)+ii]; // relative momentum between disks i and j
    } // END for (ii = 0; ii < DIM; ii++)

    // FIND SHORTEST DISTANCE CONNECTING CENTERS-OF-MASS OF DISKS i AND j
    image(yj,yi);
```

Compute relative position **q**, momentum **p** and the shortest path between the two colliding disks **i** and **j**

```
    // INITIALIZE vq AND vv
    vq = 0.0;
    vv = 0.0;

    // COMPUTE vq EQUALS THE DOT PRODUCT OF RELATIVE POSITION ↩
       VECTOR WITH RELATIVE MOMENTUM VECTOR AND vv EQUALS THE ↩
       DOT PRODUCT OF THE RELATIVE MOMENTUM VECTOR WITH ITSELF
    for (ii = 0; ii < DIM; ii++){
        vq    += dq1[ii]*v[ii];
        vv    += v[ii]*v[ii];
    } // END for (ii = 0; ii < DIM; ii++)
```

Compute **vq** = **q** · **p** and **vv** = **p** · **p**.

```
     // DECLARE TEMPORARY VARIABLES (TO AVOID OVERWRITING DURING↩
         UPDATE)
    double dvkOld[2]; // store previous values of tangent vector↩
        components associated with momentum perturbations of ↩
       disk i
    double dvlOld[2]; // store previous values of tangent vector↩
        components associated with momentum perturbations of ↩
       disk j
    double vkOld[2] = {y[DIM*(nDisks+i)],y[DIM*(nDisks+i)+1]}; ↩
       // store previous values of state vector components ↩
       associated with momentum of disk i
    double vlOld[2] = {y[DIM*(nDisks+j)],y[DIM*(nDisks+j)+1]}; ↩
       // store previous values of state vector components ↩
       associated with momentum of disk i
```

Setup copy of previous momentum states to prevent overwriting and declare temporary storage location for momentum contribution to tangent vectors (**dvkOld** and **dvlOld**)

```
    // COMPUTE ALPHA
    computeAlpha();
```

Compute **alpha**(α) for use in computation of modified collision rule below

```
    // UPDATE MOMENTA OF DISKS i AND j AFTER COLLISION (SEE ↩
       REFERENCES)
    for (ii = 0; ii < DIM; ii++){
        y[DIM*(nDisks+i)+ii] =     beta *vkOld[ii] + (1.-beta)*↩
           vlOld[ii] + alpha * vq * dq1[ii];
        y[DIM*(nDisks+j)+ii] = (1.-beta)*vkOld[ii] +     beta *↩
           vlOld[ii] - alpha * vq * dq1[ii];
    } // END for (ii = 0; ii < DIM; ii++)
```

Update the momentum of each disk after the collision according to the generalized collision rule 3:

$$\begin{aligned}\mathbf{p}_f^i &= \beta\mathbf{p}_i^i + (1-\beta)\mathbf{p}_i^j + \alpha d\mathbf{p},\\ \mathbf{p}_f^j &= (1-\beta)\mathbf{p}_i^i + \beta\mathbf{p}_i^j - \alpha d\mathbf{p}.\end{aligned}$$

```
    // UPDATE TANGENT VECTORS ASSOCIATED WITH DISKS i AND j ↩
       AFTER COLLISION (SEE REFERENCES)
    for (ii = 1; ii < nlya+1; ii++){
        // FOR ALL TANGENT VECTORS

        // DETERMINE STARTING INDEX OF CURRENT TANGENT VECTOR ↩
           WITHIN y
        pInd = ii*phaseDim;

        // SET PRE-COLLISION PERTURBATIONS AND RELATIVE ↩
           QUANTITIES
        for (jj = 0; jj < DIM; jj++){
            dq[jj] = y[pInd+DIM*j+jj] - y[pInd+DIM*i+jj]; ↩
                                     // relative position ↩
               perturbation between disks i and j before ↩
               collision
            dv[jj] = y[pInd+DIM*(nDisks+j)+jj] - y[pInd+DIM*(↩
               nDisks+i)+jj]; // relative momentum perturbation ↩
               between disks i and j before collision
            dvkOld[jj] = y[pInd+DIM*(nDisks+i)+jj]; ↩
                                          // momentum perturbation ↩
               of disk i before collision
            dvlOld[jj] = y[pInd+DIM*(nDisks+j)+jj]; ↩
                                          // momentum perturbation ↩
               of disk j before collision
        } // END for (jj = 0; jj < DIM; jj++)
```

Compute the relative position and momentum of the **nlya** tangent vectors (indexed by **ii**): **dq** $= \delta\mathbf{q}^j - \delta\mathbf{q}^i$ and **dv** $= \delta\mathbf{p}^j - \delta\mathbf{p}^i$. Setup copy of previous momentum perturbation states to prevent overwriting (**dvkOld**, **dvlOld**).

```
        // INITIALIZE DOT PRODUCT ACCUMULATORS
        dqq = 0.0;
        dvq = 0.0;
        dvv = 0.0;
        dqcv = 0.0;

        for (jj = 0; jj < DIM; jj++){
            dqq += dq[jj]*dq1[jj]; // dqq EQUALS DOT PRODUCT OF ↩
               RELATIVE POSITION PERTURBATION VECTOR WITH ↩
               RELATIVE POSITION VECTOR
            dvq += dv[jj]*dq1[jj]; // dvq EQUALS DOT PRODUCT OF ↩
               RELATIVE MOMENTUM PERTURBATION VECTOR WITH ↩
               RELATIVE POSITION VECTOR
            dvv += dv[jj]*v[jj];   // dvv EQUALS DOT PRODUCT OF ↩
               RELATIVE MOMENTUM PERTURBATION VECTOR WITH ↩
               RELATIVE MOMENTUM VECTOR
        } // END for (jj = 0; jj < DIM; jj++)
```

Compute **dqq** $= \delta\mathbf{q} \cdot \mathbf{q}$, **dvq** $= \delta\mathbf{p} \cdot \mathbf{q}$ and **dvv** $= \delta\mathbf{p} \cdot \mathbf{p}$.

```
        // COMPUTE TIME OFFSET OF COLLISION IN PERTURBED ↩
           TRAJECTORY (SEE REFERENCES)
        dtc = -dqq / vq;
```

Compute the time offset of the collision map in the offset trajectory:

$$\delta\tau_c = -\frac{\delta\mathbf{q} \cdot \mathbf{q}}{\mathbf{p}/m \cdot \mathbf{q}}$$

```
        // COMPUTE DELTA RELATIVE POSITION OFFSET VECTOR BETWEEN↩
            COLLISIONS (SEE REFERENCES)
        for (jj = 0; jj < DIM; jj++){
            dqc[jj] = dq[jj] + v[jj]*dtc;
        } // for (jj = 0; jj < DIM; jj++)
```

Compute the propagation of the relative position vector due to the time offset of the collision in the offset trajectory:

$$\delta\mathbf{q}_c \quad = \quad \delta\mathbf{q} + \frac{\mathbf{p}}{m}\delta\tau_c,$$

```
        for (jj = 0; jj < DIM; jj++){
            dqcv += v[jj]*dqc[jj]; // dqcv EQUALS DOT PRODUCT OF↩
                 DELTA RELATIVE POSITION OFFSET VECTOR WITH ↩
                RELATIVE MOMENTUM VECTOR
        } // END for (jj = 0; jj < DIM; jj++)
```

Compute **dqcv** $= \delta\mathbf{q}_c \cdot \mathbf{p}$.

```
        // COMPUTE kappa (SEE REFERENCES)
        double kappa = 2.*beta*(beta-1.)/(b_alpha+2.*a_alpha*↩
           alpha);

        // UPDATE TANGENT VECTOR COMPONENTS ASSOCIATED WITH ↩
           DISKS i AND j AFTER COLLISION (SEE REFERENCES)
        for (jj = 0; jj < DIM; jj++){
            velOff = (dvq + dqcv)*dq1[jj] + vq*dqc[jj];
            y[pInd+DIM*i+jj] += ( (1.-beta)*v[jj] + alpha*vq*dq1↩
                [jj] )*(dqq/vq);
            y[pInd+DIM*j+jj] -= ( (1.-beta)*v[jj] + alpha*vq*dq1↩
                [jj] )*(dqq/vq);

            y[pInd+DIM*(nDisks+i)+jj] =     beta *dvkOld[jj] + ↩
                (1.-beta)*dvlOld[jj] + alpha*velOff
                                         + kappa*dq1[jj]*( vv*(dvq ↩
                                            + dqcv) - vq*dvv );
            y[pInd+DIM*(nDisks+j)+jj] = (1.-beta)*dvkOld[jj] + ↩
                      beta *dvlOld[jj] - alpha*velOff
                                         - kappa*dq1[jj]*( vv*(dvq ↩
                                            + dqcv) - vq*dvv );
        } // END for (jj = 0; jj < DIM; jj++)

    } // END for (ii = 1; ii < nlya+1; ii++)
```

Update the offset position and momentum according to the generalized collision rule 3:

$$
\begin{aligned}
\delta\mathbf{q}_f^j &= \delta\mathbf{q}_i^j + \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{q}_f^k &= \delta\mathbf{q}_i^k - \left[(1-\beta)\mathbf{p} + \frac{\alpha(\mathbf{q}\cdot\mathbf{p})\mathbf{q}}{\sigma^2}\right]\left(\frac{\delta\mathbf{q}\cdot\mathbf{q}}{\mathbf{p}\cdot\mathbf{q}}\right),\\
\delta\mathbf{p}_f^j &= \beta\delta\mathbf{p}_i^j + (1-\beta)\delta\mathbf{p}_i^k + \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&+ \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c - \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q},\\
\delta\mathbf{p}_f^k &= (1-\beta)\delta\mathbf{p}_i^j + \beta\delta\mathbf{p}_i^k - \left(\frac{\alpha}{\sigma^2} + \kappa|\mathbf{p}|^2\right)(\delta\mathbf{p}\cdot\mathbf{q} + \delta\mathbf{q}_c\cdot\mathbf{p})\mathbf{q}\\
&- \frac{\alpha}{\sigma^2}(\mathbf{q}\cdot\mathbf{p})\delta\mathbf{q}_c + \kappa(\mathbf{q}\cdot\mathbf{p})(\delta\mathbf{p}\cdot\mathbf{p})\mathbf{q}.
\end{aligned}
$$

where $\kappa := \frac{2\beta(\beta-1)}{\sigma^2(b_\alpha+2a_\alpha\alpha)}$. See Subsection *computeAlpha* for computations of the coefficients $\alpha, a_\alpha, b_\alpha$.

```
    return;
}
```

Exit the routine.

tumble

This routine performs the tumble of the disk momentum. This routine treats the dynamics of the tumble as an instantaneous stretch and rotation of the disk momentum.

```
//
//  FUNCTION:  tumble
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function updates state velocity vector after a tumble has ↩
    occurred.  Tumbling is modeled as a random draw from the ↩
    initial energy (temperature) distribution of velocities.  For↩
     tangent space dynamics, tumbling is treated as an ↩
    instantaneous stretch and rotation of the pre-tumble velocity↩
     vector.
//
```

```
//   Reference:
//   Dinius, J. and J. Lega. TBD
//
//   INPUTS:
//   i - index of disk undergoing tumble
//
//   OUTPUTS:
//   (none)
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   DIM
//   phaseDim
//   nDisks
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   y (velocity of disk i is updated by tumble)
//
//   REVISION HISTORY:
//   Dinius, J.        Created                                     ↩
    10/08/11
//   Dinius, J.        Comments added for v1.0 release             ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release         ↩
    11/25/13
//
void tumble(int i)
{
    int ii; // loop index
    int pInd; // index into tangent vector components
    double vp[2],vn[2],dvp[2]; // temporary variables for state ↩
        updates

    for (ii = 0; ii < DIM; ii++){
        // STORE MOMENTUM BEFORE TUMBLE
        vp[ii] = y[DIM*(nDisks+i)+ii];

        // PERFORM RANDOM NUMBER DRAW FOR MOMENTUM AFTER TUMBLE
        vn[ii] = randn();
        y[DIM*(nDisks+i)+ii] = vn[ii];
    } // END for (ii = 0; ii < DIM; ii++)
```

Store previous values of momentum **vp** ($\mathbf{p}_i$) of tumbling disk and perform draw from a normal distribution based on the initial energy (temperature) of the system ($T = 1$) for the new value **vn** ($\mathbf{p}_f$).

```
    // COMPUTE FACTOR BY WHICH MOMENTUM VECTOR IS STRETCHED OR ↩
       CONTRACTED
    double nrmN = sqrt(vn[0]*vn[0] + vn[1]*vn[1]);
    double nrmP = sqrt(vp[0]*vp[0] + vp[1]*vp[1]);
    double chiS = nrmN / nrmP;
```

Compute the stretching factor **chiS**:

$$\chi_s = \frac{||\mathbf{p}_f||}{||\mathbf{p}_i||}.$$

```
    // COMPUTE ANGLE THAT MOMENTUM VECTOR IS ROTATED BY TUMBLE ↩
       EVENT
    double phi  = atan2(vn[1],vn[0]) - atan2(vp[1],vp[0]);
```

Solve for the angle **phi** (ϕ) that the velocity vector is rotated by tumble (ϕ_1, ϕ_2 are the angles of $\mathbf{p}_i$, $\mathbf{p}_f$ make with the $+x$-axis):

$$\begin{aligned}\phi &= \phi_2 - \phi_1,\\ \phi &= \tan^{-1}\left(\frac{p_f^{(2)}}{p_f^{(1)}}\right) - \tan^{-1}\left(\frac{p_i^{(2)}}{p_i^{(1)}}\right).\end{aligned}$$

```
    for (ii = 1; ii < nlya+1; ii++){
        // DETERMINE STARTING INDEX OF CURRENT TANGENT VECTOR ↩
           WITHIN y
        pInd = ii*phaseDim;

        // STORE VALUES OF MOMENTUM PERTURBATION BEFORE TUMBLE
        dvp[0]  = y[pInd+DIM*(nDisks+i)];
        dvp[1]  = y[pInd+DIM*(nDisks+i)+1];

        // UPDATE MOMENTUM PERTURBATION WITH STRETCH AND ↩
           ROTATION OF MOMENTUM VECTOR
        y[pInd+DIM*(nDisks+i)]   = chiS * (cos(phi)*dvp[0] - sin↩
           (phi)*dvp[1]);
        y[pInd+DIM*(nDisks+i)+1] = chiS * (sin(phi)*dvp[0] + cos↩
           (phi)*dvp[1]);
    } // END for (ii = 1; ii < nlya+1; ii++)
```

For each tangent vector in **y**, find the index into the tangent vector (**pInd**), store the previous values of the momentum perturbation, and then update the momentum after tumble by a stretch and rotation using stretching factor χ_s (**chiS**) and ϕ (**phi**).

```
    return;
}
```

Exit the routine.

checkOverlap

This routine checks to make sure that the distance between the centers-of-mass of each disk pair is not smaller than one disk diameter. This routine was primarily implemented for debugging.

```
//
//  FUNCTION:  checkOverlap
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function checks if any pair of disks overlaps.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  1 - if any pair of disks overlap
//  0 - if no pair of disks overlap
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  nDisks
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  (none)
//
```

```
//   REVISION HISTORY:
//   Dinius, J.        Modified from C. Dellago              ↩
    10/08/11
//   Dinius, J.        Comments added for v1.0 release       ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release   ↩
    11/25/13
//
int checkOverlap(void)
{
    int i,j; // loop indices
    for (i = 0; i < nDisks - 1; i++){
        // FOR ALL DISKS (EXCEPT THE LAST)
        for (j = i+1; j < nDisks; j++){
            // CHECK IF DISK i COLLIDES WITH ANY SUBSEQUENTLY ↩
                INDEXED DISKS
            if (overlap(i, j)){
                // AN OVERLAP WAS DETECTED BETWEEN DISKS i AND j↩
                    , RETURN VALUE INDICATING OVERLAP
                return 1;
            } // END if (overlap(i, j))

        } // END for (j = i+1; j < nDisks; j++)

    } // END for (i = 0; i < nDisks - 1; i++)

    // NO OVERLAP WAS DETECTED, RETURN VALUE INDICATING NO ↩
        OVERLAP
    return 0;
}
```

overlap

This routine compares disk-by-disk the distance between two disks whose indices are given as input. This routine was primarily implemented for debugging.

```
//
//   FUNCTION:  overlap
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
```

```
//  Function checks if any single interacting disk pair overlaps↩
    .
//
//  INPUTS:
//  i - index of disk 1
//  j - index of disk 2
//
//  OUTPUTS:
//  1 - if disks i and j overlap
//  0 - if disks i and j don't overlap
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  DIM
//  y
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  dq1 (dummy variable, used to store shortest distance between↩
     disk centers)
//
//  REVISION HISTORY:
//  Dinius, J.       Modified from C. Dellago               ↩
    10/08/11
//  Dinius, J.       Comments added for v1.0 release        ↩
    05/27/13
//  Dinius, J.       Cleaned up for dissertation release    ↩
    11/25/13
//
int overlap(int i, int j)
{
    int ii; // loop index

    for (ii = 0; ii < DIM; ii++){
        yi[ii] = y[DIM*i+ii]; // position vector of disk i
        yj[ii] = y[DIM*j+ii]; // position vector of disk J
    } // END for (ii = 0; ii < DIM; ii++)

    // FIND SHORTEST DISTANCE BETWEEN DISKS i AND j
    image(yi,yj);
```

Compute the relative position vector between disks **i** and **j**; call it **q**.

```
    // INITIALIZE ACCUMULATOR FOR DISTANCE BETWEEN DISK CENTERS ↩
```

```
       OF  i  AND  j
    double s = 0.0;

    // COMPUTE LENGTH (SQUARED) OF DISTANCE BETWEEN DISK CENTERS
    for (ii = 0; ii < DIM; ii++){
        s += pow(dq1[ii],2.0);
    } // END for (ii = 0; ii < DIM; ii++)
```

Compute **s** = **q** · **q**.

```
    // COMPARE COMPUTED LENGTH (SQUARED) WITH SQUARE OF DISK ↩
       DIAMETER (ASSUMES DISK DIAMETER IS 1)
    if (s < 1.0)
    {
        // IF DISTANCE IS LESS THAN DISK DIAMETER, DECLARE THAT ↩
           OVERLAP HAS OCCURRED
        return 1;
    } // END if (s < 1.0)
```

If **s** $< \sigma^2$, return 1 indicating that the disks overlap and there is a problem in the computation.

```
    // IF DISTANCE BETWEEN DISK CENTERS IS GREATER THAN (OR ↩
       EQUAL TO) 1, NO OVERLAP HAS OCCURRED
    return 0;
}
```

If no overlap was detected, exit the routine normally (return 0).

initialize

This routine initializes states and reference quantities.

```
//
//   FUNCTION:   initialize
//   MODULE:     diskDynamics
//
//   DESCRIPTION:
//   Function initializes simulation quantities.
```

```
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  boxSize
//  phaseDim
//  nlya
//  nDisks
//  DIM
//  bigTime
//  noColl
//  maxFlight
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  y           (state vector and orthonormal basis spanning ↩
    tangent space are initialized)
//  cum         (initialize vector to all zeros )
//  collArray   (initialize with first collision times and ↩
    partners for each disk)
//  tumbleArray (initialize with first tumble times for each ↩
    disk)
//  stats       (initialize members of structure with all zeros)
//
//  REVISION HISTORY:
//  Dinius, J.      Modified from C. Dellago              ↩
    10/08/11
//  Dinius, J.      Comments added for v1.0 release       ↩
    05/27/13
//  Dinius, J.      Cleaned up for dissertation release   ↩
    11/25/13
//
void initialize(void)
{
    int i,j; // loop indices
    int n;   // round-off value of square root of nDisks ↩
        calculation (check if nDisks is a perfect square).  This ↩
        is used for initial configuration of disk positions.
    int sqrtN = sqrt(nDisks);        // square root of number of↩
         disks.  This is used for initial configuration of disk ↩
        positions.
    double offSet = 1.e-5; // small additive offset to ensure that initial positions are well-defined within the simulation box
    double t1; // temporary variable for storing collision times between disks.  This is used in the initialization of collision times and partners for each disk.

    // ALLOCATE MEMORY TO STORE THE (TWO-DIMENSIONAL) MEAN OF MOMENTA OF THE DISKS
    double *mean;
    mean = new double [DIM];
```

Set local variables and define a small offset to apply to disk position. This convention was adopted from [15] in order to ensure that disk positions are initialized in the box (nothing on the boundary).

```
    // INITIALIZE ALL ELEMENTS OF y TO ZERO (THE NECESSARILY UPDATED STATES WILL BE UPDATED BELOW)
    for (i = 0; i < (nlya+1)*phaseDim; i++){
        y[i] = 0.0;
    } // END for (i = 0; i < (nlya+1)*phaseDim; i++)

    // INITIALIZE PARAMETER n USED FOR DETERMINING INITIAL DISK LOCATIONS
    if (sqrtN*sqrtN == nDisks){
        // IF nDisks IS A PERFECT SQUARE, SET n EQUAL TO sqrtN
        n = sqrtN;
    } // END if (sqrtN*sqrtN == nDisks)

    else {
        // OTHERWISE, SET n TO THE INTEGER AFTER sqrtN
        n = sqrtN + 1;
    } // END else

    // SET DISK SPACING WIDTH IN X (dx) AND Y (dy)
    double dx = boxSize[0] / ((double) n);
    double dy = boxSize[1] / ((double) n);

    for (i = 0; i < nDisks; i++){

        // SET Y-COMPONENT OF DISK i
        y[DIM*i+1] = dy / 2.0 + (i / n) * dy + offSet;

        // SET X-COMPONENT OF DISK i
        if ( (i / n) % 2 == 0){
            // FOR EVERY OTHER ROW, STAGGER THE DISKS BY dx / 2 ↩
               (CRYSTAL LATTICE PACKING)
            y[DIM*i] = (i % n) * dx + dx / 4.0 + offSet;
        } // END if ( (i / n) % 2 == 0)
        else {
            y[DIM*i] = dx / 2.0 + (i % n) * dx + dx / 4.0 + ↩
               offSet;
        } // END else

    } // END for (i = 0; i < nDisks; i++)

    // ENFORCE PERIODIC BOUNDARY CONDITIONS
    boxSet();
```

Arrange the disks in a triangular lattice (see [76]) pg. 68)and enforce the periodic boundary conditions.

```
    // INITIALIZE ACCUMULATOR OF MOMENTUM AVERAGE IN EACH ↩
       DIRECTION
    for (i = 0; i < DIM; i++){
        mean[i] = 0.0;
    }

    for (i = 0; i < nDisks; i++){

        for (j = 0; j < DIM; j++){
            // DRAW INITIAL MOMENTA FOR EACH DISK FROM A NORMAL ↩
               DISTRIBUTION OF MEAN 0 AND STANDARD DEVIATION ↩
               EQUAL TO SQRT(TEMPERATURE). TEMPERATURE IS HELD ↩
               CONSTANT (AT 1) THROUGHOUT THE SIMULATION.
            y[DIM*(nDisks+i)+j] = randn();
            // INCREMENT THE COMPUTATION OF THE MEAN MOMENTUM
            mean[j] += y[DIM*(nDisks+i)+j] / nDisks;
        } // END for (j = 0; j < DIM; j++)

    } // END for (i = 0; i < nDisks; i++)
```

Perform a normal draw with mean 0 and standard deviation 1 on each component of each disk’s velocity and update the mean velocity calculation.

```
    // TRANSLATE EACH DISK'S MOMENTUM BY -MEAN SO THAT THE TOTAL↩
        SYSTEM CENTER-OF-MASS HAS ZERO MOMENTUM
    for (int i = 0; i < nDisks; i++){
        for (int j = 0; j < DIM; j++){
            y[DIM*(nDisks+i)+j] -= mean[j];
        } // END for (int j = 0; j < DIM; j++)

    } // END for (int i = 0; i < nDisks; i++)
```

Translate each disk’s velocity so that the system center-of-mass has zero velocity. This is a computational convenience.

```
    // INITIALIZE THE ACCUMULATOR FOR KINETIC ENERGY
    double ke = 0.0;

    // COMPUTE THE KINETIC ENERGY
    for (i = 0; i< nDisks; i++){
        for (j = 0; j < DIM; j++){
            ke += 0.5*pow(y[DIM*(nDisks+i)+j],2.0);
        } // END for (j = 0; j < DIM; j++)

    }  // END for (i = 0; i< nDisks; i++)

    // RESCALE EACH DISK VELOCITY SO THAT THE TOTAL KINETIC ↩
       ENERGY IS EQUAL TO nDisks
    double c1 = sqrt(nDisks / ke); // kinetic energy scaling ↩
       factor

    for (i = 0; i < nDisks; i++){

        for (int j = 0; j < DIM; j++){
            y[DIM*(nDisks+i)+j] *= c1;
        } // END for (int j = 0; j < DIM; j++)

    } // END for (i = 0; i < nDisks; i++)
    }
```

Compute the kinetic energy and then rescale each disk’s momentum so that the average energy per disk is equal to 1.

```
    if (initialData.is_open()){
    // read in initial state from file
    double tmp;
    for (i = 0; i < 2*DIM*nDisks; i++){
      initialData.read( reinterpret_cast<char*>( &tmp ), sizeof ↩
         tmp );
      y[i] = tmp;
    }
    cout << "Read state from file..." << endl;
    initialData.close();
    } // END (initialData.is_open())
```

If the file with handle **initialData** is open, read the initial phase space trajectory values from the file and close the file when all $4N$ values have been read in.

```
    // INITIALIZE ORTHONORMAL TANGENT VECTORS TO THE STANDARD ↩
       BASIS IN R^{4*nDisks} AND INITIALIZE THE ACCUMULATOR OF ↩
       THE LOG OF THE TANGENT VECTOR STRETCHING FACTORS
    for (i = 1; i < nlya+1; i++){
        y[i*(phaseDim + 1) - 1] = 1.0;
        cum[i-1] = 0.0;
    } // END for (i = 1; i < nlya+1; i++)
```

Initialize the **nlya** tangent space vectors to be the first **nlya** standard basis elements of $\mathbb{R}^{4N}$ and the log of the stretching factor from the modified GSR process **cum**.

```
    // INITIALIZE COLLISION EVENT STRUCTURE WITH LARGE TIME AND ↩
       NULL PARTNER (noColl)
    for (i = 0; i < nDisks; i++){
        collArray[i].time    = bigTime;
        collArray[i].index   = i;
        collArray[i].partner = noColl;
    } // END for (i = 0; i < nDisks; i++)

    // TO INITIALIZE COLLISION TIMES, FOLLOW THE SAME PROCESS AS↩
        IN updateTimes ROUTINE:
    for (i = 0; i < nDisks - 1; i++){
        // SEARCH FOR THE MINIMUM TIME AND THE ASSOCIATED PARTNER OF THE NEXT COLLISION FOR DISK i
        for (j = i+1; j < nDisks; j++){
            // SEARCH OVER ALL DISKS WITH INDEX GREATER THAN i

            // COMPUTE TIME OF COLLISION BETWEEN DISKS i AND j
            t1 = binTime(i, j);

            if (t1 > 0.0){

                // IF COLLISION TIME IS POSITIVE, THEN THE COLLISION IS A VALID ONE
                if (t1 < collArray[i].time){
                    // IF THE COLLISION TIME BETWEEN DISKS j AND i IS SMALLER THAN THE CURRENT CALCULATED COLLISION
                    // TIME FOR DISK i, UPDATE THE TIME AND PARTNER FOR DISK i
                    collArray[i].time    = t1;
                    collArray[i].partner = j;
                } // END if (t1 < collArray[i].time)

                if (t1 < collArray[j].time){
                    // IF THE COLLISION TIME BETWEEN DISKS j AND i IS SMALLER THAN THE CURRENT CALCULATED COLLISION
                    // TIME FOR DISK j, UPDATE THE TIME AND PARTNER FOR DISK j
                    collArray[j].time    = t1;
                    collArray[j].partner = i;
                } // END if (t1 < collArray[j].time)

            } // END if (t1 > 0.0)

        } // END for (j = i+1; j < nDisks; j++)

    } // for (i = 0; i < nDisks - 1; i++)

    for (i = 0; i < nDisks; i++){

        // TUMBLE INITIALIZATION:
        if (tmbl){
            // IF TUMBLE FLAG IS SET GREATER THAN 0, INITIALIZE THE TUMBLE TIMES USING UNIFORM DISTRIBUTION
```

```
                tumbleArray[i].time = dtTmbl*ranf();
            } // END if (tmbl)

            else {
                // IF TUMBLING HAS BEEN DISABLED, INITIALIZE THE ↩
                   TUMBLE TIMES TO A VALUE LARGER THAN THE ↩
                   SIMULATION RUN TIME
                tumbleArray[i].time = bigTime;
            } // END else

            // CHECK FOR MAX FLIGHT CONDITION
            for (j = 0; j < DIM; j++){

                if ( fabs(collArray[i].time * y[DIM*(i+nDisks)+j]) >↩
                    maxFlightDblArray[j] ){
                    // IF THE DISTANCE TRAVELED IN EITHER SPATIAL ↩
                       DIRECTION BEFORE A VALID COLLISION INVOLVING ↩
                       DISK i EXCEEDS THE MAXIMUM FLIGHT CONDITION ↩
                       maxFlight, THEN SET THE COLLISION TIME AND ↩
                       PARTNER FOR DISK i ACCORDINGLY
                    collArray[i].time    = fabs(maxFlightDblArray[j]↩
                        / y[DIM*(i+nDisks)+j]);
                    collArray[i].partner = maxFlight;
                } // END if ( fabs(collArray[i].time * y[DIM*(i+↩
                   nDisks)+j]) > maxFlightDblArray[j] )

            } // END for (j = 0; j < DIM; j++)

        } // END for (i = 0; i < nDisks; i++)
```

Compute first collision and tumble times; see Subsection *updateTimes* as the logic is the same.

```
        // CHECK TO MAKE SURE THAT NO DISKS OVERLAP
        if (checkOverlap()){
            // EXIT THE SIMULATION IF DISKS OVERLAP
            cerr << "Disks shouldn't overlap" << endl;
            exit(1);
        } // END for (i = 0; i < nDisks; i++)
```

Indicate whether any disks overlap and exit if it is so. This should not ever happen, but is useful for debugging.

```
    // CREATE nDisks-DIMENSIONAL VECTOR OF ALL ZEROS FOR ↩
       INITIALIZATIONS BELOW
    vector<double> temp.assign(nDisks,0.);

     // INITIALIZE ACCUMULATORS WITH temp VECTOR FROM ABOVE (THIS↩
        IS USED FOR ACCUMULATION AND AVERAGING OF (I) THE ↩
       LOCALIZATION WIDTH COMPUTATION AND (II) THE ANGLE BETWEEN↩
        POSITION AND MOMENTUM PERTURBATIONS IN THE TANGENT ↩
       VECTORS)
     stats.avgPlaceholder.assign(nlya,temp);
     stats.cumPlaceholder.assign(nlya,temp);
     stats.cumCosTheta.assign(nlya,0.);
     stats.avgCosTheta.assign(nlya,0.);
```

Create vectors (and initialize to all zeros) for storage of quantities for localization measure from [91] (see Subsections *accumulateLocalizationWidth* and *computeLocalizationWidth*) and the average of the cosine of the angle between position and momentum components of tangent vectors (see Subsection *computeAvgCosTheta*).

```
     // PERFORM CLEANUP OF LOCAL MEMORY ALLOCATED FOR THE ↩
        COMPUTATION OF mean
     delete [] mean;

     return;
}
```

Cleanup memory allocated to local variable **mean** and exit the routine.

randn

This routine generates a Gaussian random variable of mean 0 and standard deviation 1 from a uniform draw using a Box-Müller transformation [9].

```
//
//   FUNCTION: randn
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function returns a Gaussian random variable drawn from a ↩
    normal distribution of mean 0 and standard deviation 1 using ↩
    the Box−Muller transform.
//
//  Reference:
//  Box, G. E. P. and Muller, M. E. "A Note on the Generation of↩
    Random Normal Deviates." Ann. Math. Stat. 29, 610−
//  611, 1958.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  rndm − Gaussian random variable drawn from a distribution of↩
    mean 0 and standard deviation 1
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  (none)
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  (none)
//
//  REVISION HISTORY:
//  Dinius, J.       Created                              ↩
    10/08/11
//  Dinius, J.       Comments added for v1.0 release      ↩
    05/27/13
//  Dinius, J.       Cleaned up for dissertation release  ↩
    11/25/13
//
double randn(void)
{
    // SEE REFERENCE FOR DETAILS.
    double r1,r2;
    double pi = 4.0*atan(1.0);
    double rndm;

    r1 = −log(1.0 − ranf());
    r2 = 2*pi*ranf();
    r1 = sqrt(2.0*r1);

    rndm = r1*cos(r2);

    return rndm;
}
```

ranf

This routine generates a uniform random number using the Lehmer generator [72]

```
//
//  FUNCTION:  ranf
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function returns a random number drawn from a uniform ↩
    distribution on the interval [0,1].
//
//  Reference:
//  Pang, T.  "An Introduction to Computational Physics", ↩
    Cambridge University Press, 1997.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  unif - random number drawn from a uniform distribution on ↩
    the interval [0,1]
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  (none)
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  iSeed (iterated to ensure a different pseudorandom number is↩
    drawn during next call)
//
//  REVISION HISTORY:
//  Dinius, J.        Created                                ↩
    10/08/11
//  Dinius, J.        Comments added for v1.0 release        ↩
    05/27/13
//  Dinius, J.        Cleaned up for dissertation release    ↩
    11/25/13
```

```
//
double ranf()
// Uniform random number generator x(n+1)= a*x(n) mod c
// with a = pow(7,5) and c = pow(2,31)-1.  SEE REFERENCE FOR ↩
    MORE DETAILS.
{
    const int ia=16807,ic=2147483647,iq=127773,ir=2836;
    int il,ih,it;
    double rc, unif;
    ih = iSeed/iq;
    il = iSeed%iq;
    it = ia*il-ir*ih;
    if (it > 0)
    {
        iSeed = it;
    } // END if (it > 0)
    else
    {
        iSeed = ic+it;
    } // END else
    rc = ic;
    unif = iSeed/rc;

    return unif;
}
```

mgsr

This routine performs modified Gram-Schmidt reorthonormalization process on the tangent vectors in **y**. The process follows Algorithm 8.1 in Lecture 8 of [93].

```
//
//  FUNCTION:  mgsr
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function performs modified Gram-Schmidt reorthonormalization↩
     to prevent collapse of all tangent vectors onto the ↩
    direction of maximal instability in forward-time.
//
//   Reference:
```

```
//   Trefethen, L. and D. Bau III.  Numerical Linear Algebra, ↩
    SIAM 1997. Pgs. 56-68
//
//   INPUTS:
//   (none)
//
//   OUTPUTS:
//   (none)
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   Time
//   phaseDim
//   nlya
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   norm (the set of stretching factors of the tangent vectors ↩
     is updated)
//   qi    (dummy variable)
//   y     (the set of tangent vectors are orthonormalized)
//   GS    (the vector containing GS bases at previous time steps ↩
     is updated when
//          modCLV == 0)
//   Rinv (the vector containing the inverse of the R matrix at ↩
     previous time
//          steps is updated when modCLV == 0 AND lastPass is true↩
     )
//   countCLV (the counter determining when to update GS and Rinv↩
       is incremented)
//
//   REVISION HISTORY:
//   Dinius, J.        Modified from C. Dellago              ↩
     10/08/11
//   Dinius, J.        Comments added for v1.0 release      ↩
     05/27/13
//   Dinius, J.        Added output for processing          ↩
     11/17/13
//                     covariant Lyapunov vectors
//   Dinius, J.        Cleaned up for dissertation release  ↩
     11/25/13
//
void mgsr(void)
{
    int i,j,k; // loop indices
    double riisq, rij, tmp1, rii; // local variables for ↩
```

```
        orthogonalization/normalization
    vector<double> Rmat, Gmat, invR, ps;

    // NOTE: Rmat IS WRITTEN OUT BY ROW, Gmat IS BY COLUMN!  ↩
        THIS IS IMPORTANT FOR
    // ENSURING THE APPROPRIATE MATRIX PRODUCTS ARE COMPUTED!

    // IF COMPUTING CLVs, WRITE OUT CURRENT PHASE SPACE ↩
        TRAJECTORY POINT
    if (Time > clvSettleTime && ((countCLV % modCLV) ==0)){
      for (i=0;i<phaseDim;i++){
          ps.push_back(y[i]);
      } // END for (i=0;i<phaseDim;i++)
      traj.push_back(ps);
    } // END if (Time > clvSettleTime && ((countCLV % modCLV) ↩
        ==0))
```

If computing covariant vectors, the settling time **clvSettleTime** has passed, and it has been **modCLV** steps since last storing the values of the phase space trajectory, store the values to **ps** and store the vector inside the vector **traj**, which stores the time-history of the trajectories.

```
    for (i = 1; i < nlya; i++){

        // COMPUTE rii IN ALGORITHM 8.1 FROM REFERENCE
        // rii IS THE NORM OF THE COLUMN VECTOR IN y ASSOCIATED ↩
            WITH INDEX i
        riisq = 0.0;
        for (j = 0; j < phaseDim; j++){
            riisq += pow(y[i*phaseDim+j],2.0);
        } // END for (j = 0; j < phaseDim; j++)

        rii = sqrt(riisq);

        if (Time > clvSettleTime && ((countCLV % modCLV) == 0) ↩
            && lastPass){
          // PUT IN 0'S IN LOWER TRIANGULAR BLOCK (EXCEPT FOR ↩
              DIAGONAL)
          for (j = 0; j < i-1; j++){
              Rmat.push_back(0.);
          }
```

```
                // PUT DIAGONAL ELEMENT INTO R ARRAY (FOR CLV ↩
                    COMPUTATION)
                Rmat.push_back(rii);
            }
```

If computing covariant vectors, the settling time **clvSettleTime** has passed, it has been **modCLV** steps since last storing the **R** matrix and the flag **lastPass** is true (i.e. the routine **processCLVs()** has been entered), put the diagonal element (along with 0’s to the left of it) into the vector container **Rmat**.

```
            // COMPUTE qi IN ALGORITHM 8.1 FROM REFERENCE
            // qi IS THE NORMALIZATION OF COLUMN VECTOR IN y ↩
                ASSOCIATED WITH INDEX i
            for (j = 0; j < phaseDim; j++){
                qi[j] = y[i*phaseDim+j] / rii;
            } // END for (j = 0; j < phaseDim; j++)

            for (j = i+1; j < nlya + 1; j++){
                // COMPUTE rij IN ALGORITHM 8.1 FROM REFERENCE
                // rij IS THE DOT PRODUCT OF COLUMN VECTOR IN y ↩
                    ASSOCIATED WITH INDEX j AND qi (PROJECTION)
                rij = 0.0;
                for (k = 0; k < phaseDim; k++){
                    rij += qi[k]*y[j*phaseDim+k];
                } // END for (k = 0; k < phaseDim; k++)

                if (Time > clvSettleTime && ((countCLV % modCLV) == ↩
                    0) && lastPass ){
                    // PUT DIAGONAL ELEMENT INTO R ARRAY (FOR CLV ↩
                        COMPUTATION)
                    Rmat.push_back(rij);
                }
```

If computing covariant vectors, the settling time **clvSettleTime** has passed, it has been **modCLV** steps since last storing the **R** matrix and the flag **lastPass** is true (i.e. the routine **processCLVs()** has been entered), then put the off-diagonal elements of the matrix **R** into the vector container **Rmat**. NOTE: It is important to note here that the vector **Rmat** is packed row-wise, whereas all other vectors in the simulation are column-packed.

```
                // FROM THE THE COLUMN VECTOR IN y ASSOCIATED WITH ↩
                   INDEX j, REMOVE THE PROJECTION OF qi ONTO THE ↩
                   COLUMN VECTOR IN y ASSOCIATED WITH INDEX j (THIS ↩
                   IS THE ORTHOGONALIZATION STEP)
                for (k = 0; k < phaseDim; k++){
                    y[j*phaseDim+k] -= rij*qi[k];
                } // END for (k = 0; k < phaseDim; k++)

            } // END for (j = i+1; j < nlya + 1; j++)

        } // END for (i = 1; i < nlya; i++)

        // NORMALIZE ALL VECTORS AND RECORD THE STRETCHING/↩
           CONTRACTION FACTORS FOR LYAPUNOV EXPONENTS CALCULATION
        for (i = 1; i < nlya+1; i++){
            tmp1 = 0.0;
            for (j = 0; j < phaseDim; j++){
                tmp1 += pow(y[i*phaseDim+j],2.0);
            } // END for (j = 0; j < phaseDim; j++)

            norm[i-1] = sqrt(tmp1);

            for (j = 0; j < phaseDim; j++){
                y[i*phaseDim+j] /= norm[i-1];
                if (Time > clvSettleTime && ((countCLV % modCLV) == ↩
                   0)){
                    // STORE GS BASIS (FOR CLV COMPUTATION)
                    Gmat.push_back( y[i*phaseDim+j] );
                } // END if (Time > clvSettleTime && ((countCLV % ↩
                   modCLV) == 0))
            } // END for (j = 0; j < phaseDim; j++)
```

Normalize each column-wise portion of the vector **y** and, when conditions are met, store column-wise the normalized Gram-Schmidt basis values in the vector container **Gmat**. This step is performed at each step after the transients are settled, as these vectors are used in the forward propagation step of **processCLVs()**.

```
        } // END for (i = 1; i < nlya+1; i++)

        if (Time > clvSettleTime && ((countCLV % modCLV) == 0)){
            // IF lastPass IS TRUE, BEGIN STORING THE (INVERTED) R ↩
```

```
                MATRICES FROM THE
            // GS PROCESS
            if (lastPass){
                // INVERT THE R MATRIX
                invR = invertRmat( Rmat );

                // STORE THE INVERTED MATRIX (FOR CLV COMPUTATION)
                Rinv.push_back(invR);
            } // if (lastPass)
            // PUSH VECTORS INTO VECTOR CONTAINER
            // THESE VECTORS EITHER DEFINE THE INITIAL CONDITION (IF↩
                !lastPass)
            // OR ARE USED TO COMPUTE THE CLVs DIRECTLY (IF lastPass↩
                )
            GS.push_back(Gmat);
        } // END if (Time > clvSettleTime && ((countCLV % modCLV) ==↩
            0))

        // INCREMENT CLV COUNTER countCLV
        countCLV += 1;

        // EXIT NORMALLY
        return;
}
```

To save on memory and computation time, only compute the inverse of the matrix **R** only when the **processCLVs()** routine has been entered (the **R** matrix is not needed until then). Store the current Gram-Schmidt basis in the vector container **GS**. Once computing the covariant vectors, store the inverted matrix **invR** in the vector container **Rinv**, which is contains the time history of the **R** matrices corresponding to the Gram-Schmidt orthonormal bases in the **GS** container and, finally, exit normally.

update

This routine computes Lyapunov exponents by taking averages of cumulative stretch factor and computes batch averages.

```
//
//  FUNCTION: update
// MODULE:   diskDynamics
//
// DESCRIPTION:
// Function updates accumulation of the log of the tangent ↩
   vector stretching factors and the time average of the ↩
   accumulation (Lyapunov exponents).  The option exists to ↩
   record batch averages of the accumulated logarithms of the ↩
   stretching factors; turn this functionality on by setting ↩
   computeBatchAvg to a nonzero value.
//
// INPUTS:
// (none)
//
// OUTPUTS:
// (none)
//
// GLOBAL INPUTS (UNCHANGED DURING CALL):
// nlya
// Time
// norm
// computeBatchAvg
// numMbatch
// batchData
//
// GLOBAL OUTPUTS (UPDATED DURING CALL):
// cum          (the log of the current stretching factors is ↩
   added to the previous values)
// lSpec        (the time average of cum is updated)
// cumBatch     (the log of the current stretching factors is ↩
   added to the previous values, or reset to zero if the
//               number of batch samples is equal to numMbatch)
// avgCumBatch (the average of cumBatch is recorded when the ↩
   number of batch samples equals numMbatch)
// countMbatch (iterated or reset to zero if counter limit has ↩
   been reached)
//
// REVISION HISTORY:
// Dinius, J.      Modified from C. Dellago             ↩
   10/08/11
// Dinius, J.      Added batch averaging                ↩
   03/15/12
// Dinius, J.      Comments added for v1.0 release      ↩
   05/27/13
// Dinius, J.      Cleaned up for dissertation release  ↩
```

```
    11/25/13
//
void update(void)
{
    int i; // loop index
    for (i = 0; i < nlya; i++){
        // FOR EACH TANGENT VECTOR:

        // ACCUMULATE THE LOG OF THE STRETCHING FACTOR OF THE ↩
           iTH TANGENT VECTOR
        cum[i] += log(norm[i]);

        // TAKE THE TIME AVERAGE OF cum; THIS IS THE CURRENT ↩
           ESTIMATE OF THE iTH LYAPUNOV EXPONENT
        lSpec[i] = cum[i] / Time;
```

Update the computation of the log of the total stretch of each column vector in the Gram-Schmidt matrix. The Lyapunov spectrum **lSpec** is the time-average of **cum**.

```
        if (computeBatchAvg){
            // IF COMPUTING BATCH AVERAGES IS SPECIFIED, ADD THE↩
               LOG OF THE STRETCHING FACTOR OF THE iTH TANGENT ↩
               VECTOR TO THE BATCH AVERAGE ACCUMULATOR cumBatch[↩
               i]
            cumBatch[i] += log(norm[i]);
```

If the input flag **computeBatchAvg** is set to a non-zero value, accumulate logarithms of the total stretch of each column vector in the Gram-Schmidt matrix.

```
            if (countMbatch == numMbatch - 1){
                // IF THE NUMBER OF SAMPLES FOR THE BATCH ↩
                   AVERAGE HAS BEEN REACHED, COMPUTE THE AVERAGE↩
                   OF cumBatch OVER THE numMbatch SAMPLES
                avgCumBatch[i] = cumBatch[i] / numMbatch;
            } // END if (countMbatch == numMbatch - 1)

        } // if (computeBatchAvg)

    } // for (i = 0; i < nlya; i++)
```

When the counter **countMbatch** reaches the specified reset value **numMbatch**-1, compute the batch average.

```
    if (computeBatchAvg){
        // IF COMPUTING BATCH AVERAGES IS SPECIFIED:

        // INCREMENT THE COUNTER
        countMbatch += 1;

        if (countMbatch == numMbatch){

            // IF THE COUNTER HAS REACHED ITS UPPER THRESHOLD, ↩
               RESET THE COUNTER
            countMbatch = 0;

            for (i = 0; i < nlya; i++){

                // RESET THE BATCH ACCUMULATOR cumBatch TO ZERO
                cumBatch[i] = 0.0;

                // WRITE THE BATCH AVERAGE VALUE TO OUTPUT FILE ↩
                   SPECIFIED BY HANDLE batchData
                batchData.write((char *) &avgCumBatch[i], sizeof↩
                   (double));
            } // END for (i = 0; i < nlya; i++)

        } // if (countMbatch == numMbatch)

    } // if (computeBatchAvg)
```

If the input flag **computeBatchAvg** is set to a non-zero value, increment **countMbatch**. When **countMbatch** counter hits **numMbatch**, reset **countMbatch** and write batch averages of Lyapunov exponents **avgCumBatch** for the **nlya** exponents to output file **batchData**.

```
    return;
}
```

Exit the routine.

accumulateLocalizationWidth

This routine increments vectors holding cumulative values for computing localization width determined by individual disk contributions to offset vectors [91].

```
//
//  FUNCTION:  accumulateLocalizationWidth
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function accumulates states for localization width ↩
    computation.
//
//  Reference: Taniguchi, T. and Morriss G.P.  "Localized ↩
    behavior in the Lyapunov vectors for quasi-one-dimensional ↩
    many-hard-disk systems".  Phys. Rev. E 68, 046203 (2003)
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  Time
//  TimeStartCollection
//  stepSize
//  nOrtho
//  nlya
//  DIM
//  nDisks
//  phaseDim
//  y
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  stats.cumPlaceholder (accumulated value of individual disk ↩
    contributions to each tangent vector norm is updated)
//  stats.avgPlaceholder (averaged value of individual disk ↩
    contributions to each tangent vector norm is updated)
//
//  REVISION HISTORY:
//  Dinius, J.       Created                              ↩
```

```
    04/01/13
//   Dinius, J.        Comments added for v1.0 release        ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release   ↩
    11/25/13
//
void accumulateLocalizationWidth(void)
{
    double gammaSq; // variable to hold each disks contribution ↩
        to the (squared) norm of each tangent vector
    int i,j;        // loop indices
    int cnt;        // disk number counter
    int numSamples = (Time - TimeStartCollection)/ (stepSize * (↩
        double)(nOrtho) );  // number of samples of localization ↩
        width taken (equivalently, the number of times this ↩
        routine has been called)
```

Initialize local variables, particularly **numSamples** for averaging; it represents the integer number of calls to this routine made by **main()** method.

```
    for (i = 1; i < nlya+1; i++){

        // FOR EACH TANGENT VECTOR:

        // RESET CURRENT DISK INDEX TO 0
        cnt = 0;

        for (j = 0; j < DIM*nDisks; j+=2){

            // FOR EACH DISK:

            // COMPUTE CURRENT DISK (INDEXED BY cnt) ↩
                CONTRIBUTION TO THE (SQUARED) NORM OF THE CURRENT↩
                 TANGENT VECTOR:  THIS CONTRIBUTION IS THE SUM OF↩
                 THE SQUARE OF EACH OF THE 4 CONTRIBUTIONS FROM ↩
                THE CURRENT DISK (X,Y,Px, Py).  EQUATION (3) FROM↩
                 REFERENCE
            gammaSq = pow(y[i*phaseDim+j],2.0) + pow(y[i*↩
                phaseDim+j+1],2.0) + pow(y[DIM*nDisks+i*phaseDim+↩
                j],2.0) + pow(y[DIM*nDisks+i*phaseDim+j+1],2.0);
```

Compute the sum of the squares of each tangent vector component associated with each disk (this is inner loop):

$$\gamma^{(i)}_{cnt} \overset{\text{def}}{=} (\delta q^{(cnt)}_x)^2_i + (\delta q^{(cnt)}_y)^2_i + (\delta p^{(cnt)}_x)^2_i + (\delta p^{(cnt)}_y)^2_i \qquad \text{(B.6.2.0.3)}$$

and for each tangent vector (outer loop). Here, **i** is the index over the number of tangent vectors, and **cnt** is the index over disk number. *IMPORTANT NOTE*: Notice the interchange of super- and subscripts in the LHS and RHS of Equation (B.6.2.0.3). This is deliberate. Reference [91] uses the opposite convention to what I have presented in Section B.4. I wish to keep the current convention for referencing states in the vector **y** as outlined in Section B.4 while also maintaining the convention in [91] for the localization measure calculation.

Reinitialize the counter **cnt** for the inner loop at the beginning of each outer loop execution.

```
                // ADD TO ACCUMULATOR THE CURRENT CONTRIBUTION OF ↩
                   THE "ENTROPY-LIKE" FACTOR FROM REFERENCE.  ↩
                   ACCUMULATION STEP IN EQUATION (8) FROM REFERENCE ↩
                   IMPLEMENTATION
                stats.cumPlaceholder[i-1].at(cnt) += gammaSq * log(↩
                   gammaSq);
```

Add current computation of disk **cnt**'s contribution to the entropy-like factor for tangent vector **i**.

```
                // COMPUTE CURRENT TIME-AVERAGE OF "ENTROPY-LIKE" ↩
                   FACTOR FROM REFERENCE.  TIME-AVERAGING STEP IN ↩
                   EQUATION (8) FROM REFERENCE IMPLEMENTATION
                stats.avgPlaceholder[i-1].at(cnt)  = stats.↩
                   cumPlaceholder[i-1].at(cnt) / ( (double)↩
                   numSamples );

                // INCREMENT DISK COUNTER
                cnt += 1;
            } // END for (j = 0; j < DIM*nDisks; j+=2

        } // END for (i = 1; i < nlya+1; i++)
```

Perform time average; that is, compute the quantity

$$\langle \gamma_{cnt}^{(i)} \log \gamma_{cnt}^{(i)} \rangle,$$

in Equation (8) from [91] and increment **cnt**.

```
    return;
}
```

Exit the routine.

computeLocalizationWidth

This routine performs computation of localization measure using time-averaged states from **accumulateLocalizationWidth** method [91].

```
//
//  FUNCTION:  computeLocalizationWidth
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function performs exponential of sum of averaged states for ↩
    localization width computation.
//
//  Reference: Taniguchi, T. and Morriss G.P.  "Localized ↩
    behavior in the Lyapunov vectors for quasi-one-dimensional ↩
    many-hard-disk systems".  Phys. Rev. E 68, 046203 (2003)
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  nlya
//  nDisks
//  stats.avgPlaceholder
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
```

```
//   stats.Sn (sum of entropy-like quantity Sn for each tangent ↩
    vector is computed)
//   stats.Wn (exponential of stats.Sn is computed)
//
//   REVISION HISTORY:
//   Dinius, J.        Created                                  ↩
    04/01/13
//   Dinius, J.        Comments added for v1.0 release          ↩
    05/27/13
//   Dinius, J.        Cleaned up for dissertation release      ↩
    11/25/13
//
void computeLocalizationWidth(void)
{
     int i,j;     // loop indices
     double sum; // accumulator for sum in Eqn (8) from reference

     for (i = 0; i < nlya; i++){

          // FOR EACH TANGENT VECTOR:

          // INITIALIZE THE TERM IN THE SUM FROM EQN (8) IN ↩
             REFERENCE TO ZERO
          sum = 0;

          // SUM EACH DISK'S CONTRIBUTION TO "ENTROPY-LIKE" TERM (↩
             MINUS SIGN BELOW TO REFLECT SIGN-CONVENTION IN ↩
             EQUATION (8) FROM REFERENCE).  THIS IS THE SUMMATION ↩
             STEP IN EQUATION (8) FROM REFERENCE.
          for (j = 0; j < nDisks; j+=1){
               sum -= stats.avgPlaceholder[i].at(j);
          } // END for (j = 0; j < nDisks; j+=1)

          // STORE SUM OF TIME-AVERAGE TERM IN Sn IN EQUATION (8) ↩
             FROM REFERENCE.  THIS IS THE VECTORIZATION OF THE ↩
             QUANTITY Sn IN EQUATION (8) FROM REFERENCE.
          stats.Sn.push_back(sum);
```

Compute

$$S^{(i)} \;:=\; -\sum_{j=1}^{N} \langle \gamma_j^{(i)} \log \gamma_j^{(i)} \rangle,$$

in Equation (8) from [91] where $\langle \cdot \rangle$ denotes time average and store its result in the global instance of the **STATS** class, **stats**.

```
        stats.Wn.push_back(exp(sum)/( (double)nDisks) );
    }
```

Compute $W^{(i)}$ in Equation (9) from [91]:

$$W^{i)} := \exp S^{(i)}.$$

and store its result in the global instance of the **STATS** class, **stats**.

```
    return;
}
```

Exit routine.

computeAlpha

This routine computes α that conserves energy in the modified collision rule 3.

```
//
//  FUNCTION:  computeAlpha
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function computes coefficients for quadratic equation in ↩
    alpha and alpha for the generalized collision rule with ↩
    parameter beta.
//
//  Reference:
//  Dinius, J.  "Dynamical Properties of a Generalized Collision↩
    Rule for Multi-Particle Systems" Ph.D dissertation.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
```

```
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   beta
//   writeAlpha
//   i_coll
//   maxAlphaCount
//   vq
//   vv
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//   a_alpha (coefficient of alpha^2 in quadratic expression is ↩
    computed)
//   b_alpha (coefficient of alpha in quadratic expression is ↩
    computed)
//   c_alpha (constant coefficient in quadratic expression is ↩
    computed)
//   alpha   (solution of quadratic equation is computed)
//
//  REVISION HISTORY:
//   Dinius, J.       Created                                   ↩
    02/08/13
//   Dinius, J.       Comments added for v1.0 release           ↩
    05/27/13
//   Dinius, J.       Cleaned up for dissertation release       ↩
    11/25/13
//
void computeAlpha()
{
    a_alpha = vq*vq;                     // coefficient of alpha^2 ↩
        in quadratic expression
    b_alpha = -(2.*beta-1.)*a_alpha; // coefficient of alpha in ↩
        quadratic expression
    c_alpha = beta*(beta-1.)*vv;       // constant coefficient in ↩
        quadratic expression
```

Setup constants in quadratic equation for computing α 3:

$$a_\alpha \alpha^2 + b_\alpha \alpha + c_\alpha = 0,$$

where

$$\begin{aligned} a_\alpha &:= \frac{(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ b_\alpha &:= -\frac{(2\beta-1)(\mathbf{q}\cdot\mathbf{p})^2}{\sigma^2}, \\ c_\alpha &:= \beta(\beta-1)|\mathbf{p}|^2. \end{aligned}$$

```
    if ( (b_alpha*b_alpha - 4.*a_alpha*c_alpha < 0.) ){
        // COMPUTED VALUE OF alpha IS COMPLEX, THEREFORE THE ↩
           COLLISION COMPUTED IS INVALID; EXIT THE SIMULATION.
        cerr << "Computed alpha should not be complex!" << endl;
        exit(1);
    } // END if ( (b_alpha*b_alpha - 4.*a_alpha*c_alpha < 0.) )
```

If the discriminant is negative, then **alpha** (α) would be complex and inadmissible; display error message and exit cleanly.

```
    // COMPUTE VALUE OF alpha USING QUADRATIC EQUATION
    alpha = (-b_alpha + sqrt(b_alpha*b_alpha - 4.*a_alpha*↩
       c_alpha)) / (2.*a_alpha);
```

Compute **alpha** (α) using the quadratic formula.

```
    if ( writeAlpha && (i_coll < maxAlphaCount) ){
        // IF WRITING OUT ALPHA AFTER EACH COLLISION HAS BEEN ↩
           SPECIFIED (BY writeAlpha INPUT FLAG) AND THE ↩
           COLLISION COUNT DOES NOT EXCEED THE MAXIMUM FOR ALPHA↩
            DATA RECORDING, WRITE OUT THE VALUE OF ALPHA, ALONG ↩
           WITH (I) THE DOT PRODUCT OF RELATIVE MOMENTUM WITH ↩
           RELATIVE POSITION OF COLLIDING DISKS AND (II) THE DOT↩
            PRODUCT OF RELATIVE MOMENTUM WITH ITSELF OF ↩
           COLLIDING DISKS
        alphaData.write((char *) &alpha, sizeof(double));
        alphaData.write((char *) &vq, sizeof(double));
        alphaData.write((char *) &vv, sizeof(double));
    } // END if ( writeAlpha && (i_coll < maxAlphaCount) )
```

If **writeAlpha** flag is set to a non-zero value and the maximum number of recorded **alpha**, **maxAlphaCount** has not been exceeded, write **alpha** (α), **vq** $=\mathbf{p}\cdot\mathbf{q}$, and **vv** $=|\mathbf{p}|^2$ to output file **alphaData**.

```
    return;

}
```

Exit the routine.

computeVacf

Compute the velocity autocorrelation function with delay specified by index timeInd (time delay = **timeInd** * **stepSize**) [41].

```
//
//  FUNCTION:  computeVacf
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function computes velocity autocorrelation function at given↩
     delay time (specified by input argument).
//
//   Reference:
//   Haile, J.M. "Molecular Dynamics Simulation". Wiley ↩
   Professional Paperback Series, 1997. Pgs. 282-288.
//
//  INPUTS:
//   timeInd - delay time at which to evaluate the velocity ↩
   autocorrelation function value
//
//  OUTPUTS:
//   sum - value of the velocity autocorrelation function at ↩
   delay time timeInd
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   velocities
//   nDisks
//
```

```
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   (none)
//
//   REVISION HISTORY:
//   Dinius, J.       Created                                    ↩
    04/09/13
//   Dinius, J.       Comments added for v1.0 release            ↩
    05/27/13
//   Dinius, J.       Cleaned up for dissertation release        ↩
    11/25/13
//
double computeVacf(int timeInd)
{
    vector<double> vtk; //velocity at time origin
    vector<double> vtkPlusTau; //velocity at time origin + delay↩
         time tau (determined by input timeInd for which tau = ↩
        timeInd*deltaTau)
    unsigned long numTimeOrigins = velocities.size() / 2; // ↩
        divide by 2 to guard against out-of-bounds indexing ↩
        issues
```

Setup local variables. Next, follow the steps for Algorithm I in [41] Pg. 284:

```
    //outer loop: loop over time origins
    double sum = 0.;
```

Initialize sum for velocity autocorrelation processing.

```
    // SET DELAY TIME BY INPUT timeInd (STEP 1 IN ALGORITHM I PG↩
        . 284 FROM REFERENCE; THIS IS DONE AUTOMATICALLY)

    // OUTER LOOP: LOOP OVER TIME ORIGINS (STEP 2)
    double sum = 0.; // initialize sum accumulator

    for (unsigned long i = 0; i < numTimeOrigins; i++){
        // READ VELOCITY AT TIME INDEX i (STEP 3)
        vtk        = velocities.at(i);

        // READ VELOCITY DISTRIBUTION AT TIME INDEX i PLUS THE ↩
            INDEX ASSOCIATED WITH TIME DELAY tau (timeInd FROM ↩
            INPUT) (STEP 4)
        vtkPlusTau = velocities.at(i+timeInd);

        // ACCUMULATE THE INTEGRAND FOR AUTOCORRELATION FUNCTION↩
            CALCULATION (STEP 5)
        for (unsigned long j = 0; j < vtk.size(); j++){
            sum = sum + vtk.at(j)*vtkPlusTau.at(j);
        } // END for (unsigned long j = 0; j < vtk.size(); j++)

        // CLEAR VECTORS USED (MEMORY MANAGEMENT)
        vtk.clear();
        vtkPlusTau.clear();
    } // END for (unsigned long i = 0; i < numTimeOrigins; i++)
```

Perform Steps 1-5 in Algorithm I and free memory allocated to vectors. The cleanup is not necessary, but it represents a good programming standard.

```
    // PERFORM TIME AND SPATIAL AVERAGING IN EQUATION 7.29 FROM ↩
        REFERENCE.
    sum /= (double)(numTimeOrigins*nDisks);

    return sum;
```

Normalize the sum by number of time steps and number of disks and return the result.

```
}
```

Exit the routine.

computeAvgCosTheta

Compute the time-average of the cosine of the angle between the position and momentum components of each tangent vector. This routine determines how closely aligned the two 2***nDisk** component vectors are aligned, which allows for a better understanding of how to visualize the Lyapunov modes [21, 30].

```
/
//   FUNCTION:  computeAvgCosTheta
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function computes the average of the cosine of the angle ↩
    between position and momentum perturbations in tangent ↩
    vectors
//
//  Reference:
//  Eckmann, J.P. et. al. "Lyapunov Modes in Hard−Disk Systems".↩
     Journal of Stat. Phys., Vol. 118, Nos. 5/6.  March 2005.  ↩
    Pgs. 813−847.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  Time
//  TimeStartCollection
//  stepSize;
//  nOrtho
//  nlya
//  DIM
//  nDisks
//  phaseDim
//  y
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  stats.cumCosTheta (accumulated value of the cosine of the ↩
    angle between the position and momentum offset
//                    components of each tangent vector is ↩
    updated)
//  stats.avgCosTheta (averaged value of the cosine of the angle↩
     between the position and momentum offset
//                    components of each tangent vector is ↩
    updated)
//
//  REVISION HISTORY:
//  Dinius, J.      Created                              ↩
    04/09/13
//  Dinius, J.      Comments added for v1.0 release      ↩
    05/27/13
```

```
//   Dinius, J.        Cleaned up for dissertation release  ↩
    11/25/13
//
void computeAvgCosTheta()
{

    int i,j;          // loop indices
    double nrmSqPos;  // norm squared of position perturbation ↩
       contribution to individual tangent vector
    double nrmSqMom;  // norm squared of momentum perturbation ↩
       contribution to individual tangent vector
    double dot;       // dot product of vector of position ↩
       perturbation components of individual tangent vector with↩
        vector of momentum perturbation components of the same ↩
       individual tangent vector

    int numSamples = (Time - TimeStartCollection)/ (stepSize * (↩
       double)(nOrtho) ); // number of samples of localization ↩
       width taken (equivalently, the number of times this ↩
       routine has been called)
```

Setup local variables.

```
// FOR EACH TANGENT VECTOR:

        // INITIALIZE ACCUMULATORS FOR NORM SQUARED AND DOT ↩
           PRODUCT TERMS
        nrmSqPos = 0.;
        nrmSqMom = 0.;
        dot = 0;

        for (j = 0; j < DIM*nDisks; j++){
            // ADD THE CONTRIBUTION OF THE CURRENT INDEX TO THE ↩
               NORM SQUARED AND DOT PRODUCT TERMS
            nrmSqPos += pow(y[i*phaseDim+j],2.);
            nrmSqMom += pow(y[DIM*nDisks+i*phaseDim+j],2.);
            dot      += y[i*phaseDim+j] * y[DIM*nDisks+i*↩
               phaseDim+j];
        } // END for (j = 0; j < DIM*nDisks; j++)

        // COMPUTE (AND ADD TO ACCUMULATOR) THE COMPUTED VALUE ↩
           OF THE COSINE OF THE ANGLE BETWEEN THE POSITION AND ↩
```

```
                MOMENTUM COMPONENTS OF THE iTH TANGENT VECTOR
        stats.cumCosTheta.at(i-1) += dot/(sqrt(nrmSqPos*nrmSqMom↩
          ));
```

Solve the dot product formula for $\cos\theta$:

$$\cos\theta = \frac{\delta\mathbf{Q}^{(i)} \cdot \delta\mathbf{P}^{(i)}}{\|\delta\mathbf{Q}^{(i)}\|_2 \, \|\delta\mathbf{P}^{(i)}\|_2}.$$

where $\delta\mathbf{Q}^{(i)}$ and $\delta\mathbf{P}^{(i)}$ are, respectively, the position and momentum 2***nDisk**-sized vectors corresponding to the **i**th Lyapunov vector (mode). The standard Euclidean norm on $\mathbb{R}^{\mathbf{2nDisks}}$ is $\|\cdot\|_2$. Accumulate the running totals for time-averaging in the **stats.cumCosTheta** vector.

```
// COMPUTE THE TIME-AVERAGE OF THE COSINE OF THE ANGLE BETWEEN ↩
   THE POSITION AND MOMENTUM COMPONENTS OF THE iTH TANGENT ↩
   VECTOR
        stats.avgCosTheta.at(i-1) = stats.cumCosTheta.at(i-1)/( ↩
          (double)numSamples );

    } // END for (i = 1; i < nlya+1; i++)
```

Compute the time-average corresponding to the **i**th Lyapunov vector (mode).

```
    return;
}
```

Exit the routine.

invertRmat

This routine computes the inverse of an input upper-triangular matrix packed column-wise into a vector container (**Rvec**). This routine uses forward-substitution (see [39]). No input checking is perfored, so the user must be careful to ensure that the input passed actually represents an upper-triangular matrix. In order to remove the storage burden, zero entries in the lower-triangular block are omitted from the output. These zeros are added back subsequently in the **processCLVs()** routine.

```
//
//  FUNCTION:  invertRmat
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  This routine computes the inverse of an upper triangular matrix using forward substitution.
//
//  Reference:
//  Golub, Gene H. and Charles F. Van Loan, "Matrix Computations: Third Edition".  The Johns Hopkins
//  University Press, 1996.  Pg. 88.
//
//  INPUTS:
//  Rvec- R matrix from MGSR (in vector form)
//
//  OUTPUTS:
//  invRred- inverted R matrix (with zeros in lower-triangular block trimmed to save memory)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  phaseDim
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//
//  REVISION HISTORY:
//  Dinius, J.       Created                                  11/02/13
//  Dinius, J.       Cleaned up for dissertation release      11/25/13
//
vector<double> invertRmat(vector<double> Rvec)
{
  int i, j, k;
  vector<double> invR,invRred;

  // INITIALIZE INVERSE CONTAINER WITH ZEROS
  invR.assign(phaseDim*phaseDim,0.);

  // PERFORM INVERSE COMPUTATION (USING FORWARD SUBSTITUTION)
  for (i=0; i<phaseDim; i++){
    invR.at(phaseDim*i+i) = 1. / Rvec.at(phaseDim*i+i);

    for (j=0; j<i; j++){
      for (k=0; k<i; k++){
        invR.at(phaseDim*j+i) += invR.at(phaseDim*j+k)*Rvec.at(↩
           phaseDim*k+i);
      } // END for (k=0; k<i; k++)
    } // END for (j=0; j<i; j++)

    for (j=0; j<i; j++){
      invR.at(phaseDim*j+i) /= -Rvec.at(phaseDim*i+i);
    } // END for (j=0; j<i; j++)
  } // END for (i=0; i<phaseDim; i++)

  // TRIM ZEROS (SAVE ON STORAGE REQUIREMENT)
  for (i=0; i<phaseDim; i++){
    for (j=i; j<phaseDim; j++){
      invRred.push_back( invR.at(i*phaseDim+j) );
    } // END for (j=i; j<phaseDim; j++)
  } // END for (i=0; i<phaseDim; i++)

  // CLEANUP LOCAL MEMORY
  invR.clear();

  // RETURN INVERSE MATRIX
  return invRred;
}
```

computeCLVs

This routine is the main driver for computing the covariant Lyapunov vectors using Ginelli’s method (see Algorithm 4). Memory-saving techniques suggested by Bosetti and Posch [8] are employed.

```
//
//  FUNCTION: computeCLVs
//  MODULE:   diskDynamics
//
//  DESCRIPTION:
//   This routine implements Ginelli's method for computing ↩
    covariant Lyapunov vectors for colliding disk systems. ↩
    Memory-saving tricks
//   discussed in Bosetti and Posch are employed.
//
```

```
//  Reference:
//  Bosetti, H. and H.A. Posch.  "Covariant Lyapunov vectors for↩
    rigid disk systems".  Chemical Physics 375 (2010) 296-308.
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  phaseDim
//  nOrtho
//  nDisks
//  stepSize
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  clvStats.avgPlaceholder (initialized)
//  clvStats.cumPlaceholder (initialized)
//  clvStats.cumCosTheta    (initialized)
//  clvStats.avgCosTheta    (initialized)
//  cumback                 (accumulator of log of stretching/↩
   contraction factors of C matrix is initialized and updated)
//  expBack                 (backward Lyapunov exponent ↩
   container is initialized and updated)
//  traj                    (vector containing phase space ↩
   trajectory time history is cleared to avoid overwriting after
//                           local copy is created)
//  GS                      (vector containing orthonormalized ↩
   GS bases time history is cleared to avoid overwriting after
//                           local copy is created.  Once CLV ↩
   computation process has begun, the GS bases container is
//                           updated and entries trimmed to save↩
    memory)
//  Rinv                    (Once CLV computation process has ↩
   started, the Rinv container is
//                           updated and entries trimmed to save↩
    memory)
//  modCLV                  (Once forward transients have ↩
   settled, update modCLV to enable capture orthonormalization ↩
   data
//                           at each time step)
//  lastPass                (Once CLV computation process has ↩
   started, update lastPass to enable capture of R matrix from
```

```
//                              mgsr() routine)
//
//  REVISION HISTORY:
//   Dinius, J.        Created                                   ↩
    11/02/13
//   Dinius, J.        Cleaned up for dissertation release   ↩
    11/25/13
//
void computeCLVs()
{
  int i, j, k, counter, jj; // loop indices
  vector<double> C, Cp, V, Vp, invRfull, invRred, G, Gp, expBack↩
     , cumback, ps; // local vector containers
  double nrm; // scalar to renormalize columns of C matrix
  vector< vector<double> > GSloc, trajLoc; // local vector ↩
     containers (avoid overwriting globals traj and GS when re-↩
     entering mgsr() routine)
  double dt = (double)(modCLV) * stepSize * (double)(nOrtho); //↩
      timestep between successive covariant vector output
  double t  = 0.; // local timekeeping variable
  double tprop = 0; // local timekeeping variable for forward ↩
     propagation step (reset each advance in time of dt)
  double dtInner = stepSize * (double)(nOrtho); // timestep ↩
     between successive backward iterations (used in computing ↩
     covariant vectors)
  int fcount = 0; // counter for forward propagation steps (↩
     resets when fcount==timesteps)
  double currTime = 0.; // local timekeeping variable for ↩
     backward propagation step (reset each advance in time of dt↩
     )
  int timesteps = modCLV*nOrtho; // number of timesteps between ↩
     successive forward propagations

  // RESET modCLV (WE WANT EACH ORTHONORMALIZATION STORED FROM ↩
     NOW ON)
  modCLV = 1;

  // SET lastPass FLAG TO INDICATE THAT WE NOW WANT TO STORE THE↩
      INVERSE OF THE R MATRIX FROM mgsr()
  lastPass = 1;

  // CREATE LOCAL COPIES OF GS, AND traj VARIABLES (CLEAR ↩
     GLOBALS FOR IN-BETWEEN UPDATES)
  GSloc   = GS;
```

```
trajLoc = traj;
GS.clear();
traj.clear();

// INITIALIZE BACKWARD ACCUMULATOR AND EXPONENT CONTAINERS (↩
   FOR LYAPUNOV EXPONENT VERIFICATION)
cumback.assign(phaseDim,0.);
expBack.assign(phaseDim,0.);

// INITIALIZE clvStats CONTAINERS
vector<double> temp;
  temp.assign(nDisks,0.);

clvStats.avgPlaceholder.assign(phaseDim,temp);
clvStats.cumPlaceholder.assign(phaseDim,temp);
clvStats.cumCosTheta.assign(phaseDim,0.);
clvStats.avgCosTheta.assign(phaseDim,0.);

// INITIALIZE C MATRIX TO THE IDENTITY MATRIX
C.assign(phaseDim*phaseDim,0.);
for (i=0; i<phaseDim; i++){
  C.at(i*phaseDim+i) = 1.;
} // END for (i=0; i<phaseDim; i++)

// INITIALIZE V TO THE LAST GS STEP
// ( V(end) = G(end)*C(end) = G(end)*I = G(end) )
V = GSloc.back();
```

Set $\mathbf{W}_k = \mathbf{G}_{n+\omega k}\mathbf{C}_k$ as in Equation (2.3.14), the final value $\mathbf{C}_k$ is chosen arbitrarily as the $4N \times 4N$ identity matrix. The variable ω denotes the number of steps between successive orthonormalizations.

```
// CLEAN UP MEMORY ON THE FLY BY REMOVING THE LAST ELEMENT OF ↩
   trajLoc and GSloc
// (FINAL TIME POINT NOT NEEDED ANY LONGER)
trajLoc.pop_back();
GSloc.pop_back();
```

Remove the final point in each of the **trajLoc** and **GSloc**, as they are no longer needed for covariant vector computation.

```
// INITIALIZE OUTER INDEX FOR CLV COMPUTATION
i = GSloc.size();

// INITIALIZE ACCUMULATION COUNTER (SINCE OUTER LOOP ↩
   DECREMENTS INDEX i)
int countAcc = 0;
```

The following loop has an implicit indexing; call this index *n*. For what follows, the final point in the **GSloc** and **trajLoc** vectors are used for (1) forward propagation in time of both the phase and tangent space dynamics and (2) computation of the covariant Lyapunov vectors using backward propagation and Ginelli's method [36, 37].

```
while (!GSloc.empty() || !trajLoc.empty()){

  // DECREMENT INDEX i
  i -= 1;

  // WRITE OUT CURRENT INDEX AND BACKWARD EXPONENT CALCULATION
  // (VERIFICATION AND STATUS CHECK)
  cout << i << " " << expBack.at(0) << endl;

  // WRITE OUT CURRENT TIME (t), COVARIANT VECTORS (V) AND ↩
     EXPONENTS (expBack) AT EACH
  // MAIN LOOP ITERATION
  clvData.write((char *) &t, sizeof(double));
  for (j=0; j<phaseDim; j++){
    for (k=0; k<phaseDim; k++){
      clvData.write((char *) &V.at(j*phaseDim+k), sizeof(↩
         double));
    } // END for (k=0; k<phaseDim; k++)
  } // END for (j=0; j<phaseDim; j++)
  for (j=0; j<phaseDim; j++){
    clvData.write((char *) &expBack.at(j), sizeof(double));
  } // END for (j=0; j<phaseDim; j++)

  // POPULATE INITIAL CONDITION FOR FORWARD PROPAGATION PART
  ps = trajLoc.back(); // y(n-1)
  G  = GSloc.back();   // G(n-1)
```

Set up the initial condition for the forward propagation step (from time step $n - 1$ to n).

```
    // DELETE LAST POINT IN VECTORS trajLoc AND GSloc (SAVE ↩
       MEMORY)
    trajLoc.pop_back();
    GSloc.pop_back();

    // OVERWRITE y ARRAY WITH PHASE SPACE AND TANGENT VECTORS ↩
       FROM GS BASIS
    // FOR USE IN NEXT FORWARD PROPAGATION
    for (j=0; j<phaseDim; j++){
      for (k=0; k <= phaseDim; k++){
        if (k==0){
          y[j] = ps.at(j);
        } // END if (k==0)
        else{
          y[k*phaseDim + j] = G.at( (k-1)*phaseDim+j );
        } // END else
      } // END for (k=0; k <= phaseDim; k++)
    } // END for (j=0; j<phaseDim; j++)

    // INITIALIZE COLLISION TIMES FOR NEXT FORWARD PROPAGATION
    initializeCLVcollTimes();
```

Initialize the next collision times for the initial condition defined by **y**.

```
    // CLEAR LOCALS (MEMORY CLEANUP)
    G.clear();
    ps.clear();

    // FORWARD PROPAGATE INTERMEDIATE STATES (FROM i TO i+1) TO ↩
       SAVE ON MEMORY ALLOCATION
    while (fcount <= timesteps){
      // AT EACH nOrtho'TH STEP PERFORM mgsr()
      if ( (fcount % nOrtho) == 0 ){
        mgsr();
      } // END if ( (fcount % nOrtho) == 0 )

      // PERFORM STEP FORWARD IN TIME (INCLUDES COLLISIONS AND ↩
         FREE-FLIGHT)
      hardStep(stepSize);

      // INCREMENT tprop
      tprop += stepSize;

      // CHECK TO ENSURE THAT DISKS DON'T OVERLAP
      if (checkOverlap()){
        // IF DISKS OVERLAP, EXIT
        cout << "Disks overlap... exiting" << endl;
      } // END if (checkOverlap())

      // INCREMENT LOOP COUNTER
      fcount+= 1;
    } // END while (fcount <= timesteps)
```

Propagate phase space and tangent states at $n-1$ time step $y(n-1)$ forward to time step n. This process is the same as in **main()**. The routine **mgsr()** stores the Gram-Schmidt orthornormalization matrices at each step for constructing the covariant Lyapunov vectors.

```
    // FINAL POINT IN GS IS NOT NEEDED FOR CLV COMPUTATION SO ↩
       REMOVE IT
    // TO SAVE MEMORY
    GS.pop_back();

    // UNTIL THE CONTAINER OF GS COLUMN-PACKED MATRICES IS EMPTY↩
       , RECONSTRUCT
    // COVARIANT VECTORS USING GINELLI'S METHOD (INDEX "I" IS ↩
       USED AS A
    // PLACEHOLDER IN COMMENTS THOUGH NO ACTUAL INDEX IS USED IN↩
        THIS LOOP)
    while (!GS.empty()){
```

At each orthonormalization step between time steps $n-1$ and n, construct the covariant Lyapunov vectors using Ginelli's method. Like in the main loop above, this inner loop has an implicit index; call this index j.

```
      // UNPACK RED RINV MATRIX  (invRred^{-1}(I))
      invRred = Rinv.back(); //

      // DELETE LAST ENDPOINT IN Rinv (SAVE MEMORY)
      Rinv.pop_back();

      // INITIALIZE COUNTER FOR ADDING ZEROS BACK TO COLUMN-↩
```

```
        PACKED
    // INVERSE MATRIX
    counter = 0;

    // INITIALIZE CONTAINER FOR FULL COLUMN-PACKED INVERSE MATRIX
    invRfull.assign(phaseDim*phaseDim,0.);

    // PUT ZEROS BACK INTO THE INVERSE OF THE R MATRIX
    for (j=0;j < phaseDim; j++){
      for (k=j;k < phaseDim; k++){
        invRfull.at(j*phaseDim+k) = invRred.at(counter);
        counter += 1;
      } // END for (k=j;k < phaseDim; k++)
    } // END for (j=0;j < phaseDim; j++)

    // COMPUTE C MATRIX AT PREVIOUS TIMESTEP
    // C(I-1) = R^{-1}(I)*C(I-1)
    // NOTE: invRfull MATRIX IS ROW-PACKED, SO CONVERSION IS NEEDED
    // BEFORE MULTIPLICATION WITH A COLUMN-PACKED MATRIX
    Cp = matrixMultiply( rowToColumn(invRfull) , C);
```

Compute $\mathbf{C}_{j-1} = \mathbf{R}^{-1}_{(n-1)+\omega j}\mathbf{C}_j$ from Equation (2.3.16). In the above expression, the row-packed **invRfull** vector must be converted to column-wise indexing for the appropriate multiplication to be computed.

```
    // RESCALE THE COLUMNS BY THEIR RESPECTIVE NORMS (REMOVE STRETCHING AND CONTRACTION FACTORS),
    // ACCUMULATE cumback AND COMPUTE THE BACKWARD LYAPUNOV EXPONENTS (FOR FUNCTIONAL VERIFICATION)
    for (j=0; j < phaseDim; j++){
      nrm = columnNorm(Cp,j);
      cumback.at(j) += log(nrm);
      Cp  = rescaleCol(Cp,j,nrm);

      // LYAPUNOV EXPONENTS COMPUTATION (USES CURRENT TIME WHICH INCORPORATES TIME PASSAGE
      // INSIDE OF THE while(!GS.empty()) LOOP)
      expBack.at(j) = cumback.at(j) / (t+currTime);
    } // END for (j=0; j < phaseDim; j++)
```

Renormalize each column of the $\mathbf{C}_{j-1}$ matrix, and record the log of the stretching/contraction factors. The running sum of these factors are averaged in time to approximate the Lyapunov exponents.

```
        // INITIALIZE G(I-1)
        Gp = GS.back();

        // V(I-1) = G(I-1)*C(I-1)
        Vp = matrixMultiply(Gp,Cp);
```

Compute $\mathbf{W}_{j-1} = \mathbf{G}_{(n-1)+\omega(j-1)}\mathbf{C}_{j-1}$ as in Equation (2.3.14).

```
        // INCREMENT ACCUMULATION COUNTER AND ACCUMULATE ↩
           LOCALIZATION AND AVERAGE ANGLE MEASURES
        countAcc += 1;
        accumulateCLVlocalizationWidth(Vp, countAcc);
        computeCLVavgCosTheta(Vp, countAcc);

        // UPDATE VALUE OF PLACEHOLDER C FOR NEXT LOOP ITERATION
        C = Cp;

        // INCREMENT currTime TIMEKEEPER (INSIDE THE while(!GS.↩
           empty()) LOOP)
        currTime += dtInner;

        // CLEAR UNNEEDED VARIABLES TO SAVE MEMORY
        Cp.clear();
        Gp.clear();
        GS.pop_back();

      } // END while (!GS.empty())

      // UPDATE COVARIANT VECTORS CONTAINER FOR OUTPUT ON NEXT ↩
         MAIN LOOP
      // (while (!GSloc.empty() || !trajLoc.empty())) ITERATION
      V = Vp;

      // RESET INNER LOOP TIMEKEEPING AND COUNTER VARIABLES
      tprop = 0.;
      fcount = 0;
```

```
    currTime = 0.;

    // UPDATE MAIN TIMEKEEPING VARIABLE
    t += dt;

  } // END while (!GSloc.empty() || !trajLoc.empty())

  // FINALIZATION STEPS:

  // COMPUTE LOCALIZATION WIDTH OF CLVs
  computeCLVlocalizationWidth();

  // WRITE OUT RESULTS FOR AVERAGE LOCALIZATION WIDTH AND ↩
     AVERAGE ANGLE BETWEEN
  // MOMENTUM AND POSITION PERTURBATION COMPONENTS FOR EACH ↩
     COVARIANT
  // VECTOR
  for (j=0; j<phaseDim; j++){
    clvData.write((char *) &clvStats.Wn.at(j), sizeof(double));
  } // END for (j=0; j<phaseDim; j++)

  for (j=0; j<phaseDim; j++){
    clvData.write((char *) &clvStats.avgCosTheta.at(j), sizeof(↩
       double));
  } // END for (j=0; j<phaseDim; j++)

  // CLOSE OUTPUT FILE
  clvData.close();

  // CLEANUP REMAINING MEMORY ALLOCATED TO VECTOR CONTAINER
  V.clear();

  // EXIT NORMALLY
  return;
}
```

rowToColumn

This routine changes a matrix representation as a vector from row-wise indexing to column-wise.

```
//
//   FUNCTION:  rowToColumn
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
//   This routine converts a matrix representation as a row-↩
    packed vector to a
//   a column-packed vector.
//
//   Reference:
//
//   INPUTS:
//   V- vector containing a row-packed matrix
//
//   OUTPUTS:
//   rowToColSwap- vector containing the column-packed matrix ↩
    extracted from V.
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   phaseDim
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//
//   REVISION HISTORY:
//   Dinius, J.       Created                                   ↩
    11/02/13
//   Dinius, J.       Cleaned up for dissertation release       ↩
    11/25/13
//
vector<double> rowToColumn( vector<double> V )
{
  // INITIALIZE OUTPUT CONTAINER
  vector<double> rowToColSwap;
  rowToColSwap.assign(phaseDim*phaseDim,0.);

  // SWITCH ROW-COLUMN (i,j) WITH COLUMN-ROW (j,i) INDEXING
  for (int i=0; i<phaseDim; i++){
    for (int j=0; j<phaseDim; j++){
      rowToColSwap.at(j*phaseDim+i) = V.at(i*phaseDim+j);
    } // END for (int j=0; j<phaseDim; j++)
  } // END for (int i=0; i<phaseDim; i++)

  // RETURN COLUMN-PACKED MATRIX
  return rowToColSwap;
}
```

matrixMultiply

This routine performs matrix multiplication of two matrices indexed column-wise inside of vector containers. The result is returned as a column-packed vector.

```
//
//  FUNCTION:  matrixMultiply
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   This function returns the (column-packed) product of two ↩
    matrices (represented by column-packed vectors).
//  C = A*B
//
//   Reference:
//
//  INPUTS:
//  v1- column-packed vector representing first matrix (A above)
//  v2- column-packed vector representing second matrix (B above↩
    )
//
//  OUTPUTS:
//   prod- column-packed vector representing the matrix product (↩
    C above)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//   phaseDim
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//
//  REVISION HISTORY:
//   Dinius, J.        Created                                  ↩
    11/02/13
//   Dinius, J.        Cleaned up for dissertation release      ↩
    11/25/13
//
vector<double> matrixMultiply(vector<double> v1, vector<double> ↩
    v2)
{
  // INITIALIZE OUTPUT CONTAINER
  vector<double> prod;
  prod.assign(phaseDim*phaseDim,0.);

  // PERFORM MATRIX PRODUCT
  // SUM THE PRODUCT OF ELEMENTS OF ROW OF FIRST MATRIX
  // TIMES ELEMENTS OF COLUMN OF SECOND MATRIX
  for (unsigned i=0; i < phaseDim; i++){ //COLUMN

    for (unsigned k=0; k < phaseDim; k++){ //ROW

      for (unsigned j = 0; j<phaseDim; j++){

        prod.at(i*phaseDim+k) += v1.at(j*phaseDim+k)*v2.at(i*↩
           phaseDim+j);

      } // END for (unsigned j = 0; j<phaseDim; j++)

    } // END for (unsigned k=0; k < phaseDim; k++)

  } // END for (unsigned i=0; i < phaseDim; i++)

  // RETURN THE MATRIX PRODUCT
  return prod;
}
```

columnNorm

This routine computes the Euclidean norm of a segment of a vector corresponding to a column of a matrix (represented by a column-packed vector **v**) specified by input **col**.

```
//
//   FUNCTION:  columnNorm
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
//   This routine computes the (Euclidean) norm of a section of ↩
   the input vector
//   corresponding to a matrix column (set by input).
//
//   Reference:
```

```
//
//   INPUTS:
//   v- vector containing a column-packed matrix
//   col- column of the (column-packed) matrix for which the norm⟵
     is to be
//         computed
//
//   OUTPUTS:
//   sqrt(sum)
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//
//   REVISION HISTORY:
//   Dinius, J.       Created                                    ⟵
    11/02/13
//   Dinius, J.       Cleaned up for dissertation release     ⟵
    11/25/13
//
double columnNorm(vector<double> v, int col)
{
  double tmp;

  // INITIALIZE SUM TERM (FOR SQUARED ENTRIES)
  double sum = 0.;

  // SUM SQUARED TERMS OVER ENTIRE COLUMN
  for (unsigned j = 0; j < phaseDim; j++){
    sum += v.at(col*phaseDim+j)*v.at(col*phaseDim+j);
  }

  // COMPUTE sqrt OF sum
  tmp = sqrt(sum);

  // RETURN EUCLIDEAN NORM OF COLUMN col
  return tmp;
}
```

rescaleCol

This routine rescales the entries of the column-packed vector representation of a matrix **v** corresponding to a matrix column (specified by input **col**) by an input constant **nrm**.

```
//
//  FUNCTION:  rescaleCol
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  This routine scales the entries of a vector corresponding to↩
    an input
//  matrix column by an input scalar quantity (corresponding to ↩
   the column
//  norm)
//
//  Reference:
//
//  INPUTS:
//  v- vector containing a column-packed matrix
//  col- column of the (column-packed) matrix for which the norm↩
    is to be
//        computed
//  nrm- column-scaling factor
//
//  OUTPUTS:
//  vn- vector containing column-packed matrix with column col ↩
   scaled by 1/nrm.
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  phaseDim
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//
//  REVISION HISTORY:
//  Dinius, J.       Created                                  ↩
   11/02/13
//  Dinius, J.       Cleaned up for dissertation release    ↩
   11/25/13
//
vector<double> rescaleCol(vector<double> v, int col, double nrm)
{
  // MAKE COPY OF INPUT v
  vector<double> vn = v;

  // RESCALE vn ENTRIES OF COLUMN col BY 1/nrm
  for (unsigned j = 0; j < phaseDim; j++){
    vn.at(col*phaseDim+j) /= nrm;
  } // END for (unsigned j = 0; j < phaseDim; j++)

  // RETURN vn
  return vn;
}
```

initializeCLVcollTimes

This routine is a direct copy of the initialization logic for collision times from **initialize()**. It is copied here for simplicity of implementation in the **computeCLVs()** routine.

```
//
//   FUNCTION:  initializeCLVcollTimes
//   MODULE:    diskDynamics
//
//   DESCRIPTION:
//   This routine is called to initialize the collision times and partners for each disk for each successive pass of the main loop of processCLVs().
//   This reinitialization is required given the memory-saving strategy invoked in processCLVs().
//
//   Reference:
//
//   GLOBAL INPUTS (UNCHANGED DURING CALL):
//   nDisks
//   maxFlight
//   noColl
//   maxFlightDblArray
//   DIM
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   collArray   (updated values for collision partners and associated times for all nDisks disks)
//   tumbleArray (updated values for tumble times for all nDisks disks)
//
//   REVISION HISTORY:
//   Dinius, J.       Created from initialize() logic       11/02/13
//   Dinius, J.       Cleaned up for dissertation release   11/25/13
```

```
//
void initializeCLVcollTimes()
{
  int i,j; // loop indices
    double t1;

  // INITIALIZE COLLISION EVENT STRUCTURE WITH LARGE TIME AND ↩
     NULL PARTNER (noColl)
    for (i = 0; i < nDisks; i++){
        collArray[i].time    = bigTime;
        collArray[i].index   = i;
        collArray[i].partner = noColl;
    } // END for (i = 0; i < nDisks; i++)

    // TO INITIALIZE COLLISION TIMES, FOLLOW THE SAME PROCESS AS↩
        IN updateTimes ROUTINE:
    for (i = 0; i < nDisks - 1; i++){
        // SEARCH FOR THE MINIMUM TIME AND THE ASSOCIATED ↩
           PARTNER OF THE NEXT COLLISION FOR DISK i
        for (j = i+1; j < nDisks; j++){
            // SEARCH OVER ALL DISKS WITH INDEX GREATER THAN i

            // COMPUTE TIME OF COLLISION BETWEEN DISKS i AND j
            t1 = binTime(i, j);

            if (t1 > 0.0){

                // IF COLLISION TIME IS POSITIVE, THEN THE ↩
                   COLLISION IS A VALID ONE
                if (t1 < collArray[i].time){
                    // IF THE COLLISION TIME BETWEEN DISKS j AND↩
                       i IS SMALLER THAN THE CURRENT CALCULATED↩
                       COLLISION
                    // TIME FOR DISK i, UPDATE THE TIME AND ↩
                       PARTNER FOR DISK i
                    collArray[i].time    = t1;
                    collArray[i].partner = j;
                } // END if (t1 < collArray[i].time)

                if (t1 < collArray[j].time){
                    // IF THE COLLISION TIME BETWEEN DISKS j AND↩
                       i IS SMALLER THAN THE CURRENT CALCULATED↩
                       COLLISION
                    // TIME FOR DISK j, UPDATE THE TIME AND ↩
```

```
                              PARTNER FOR DISK j
                          collArray[j].time    = t1;
                          collArray[j].partner = i;
                      } // END if (t1 < collArray[j].time)

                  } // END if (t1 > 0.0)

              } // END for (j = i+1; j < nDisks; j++)

      } // for (i = 0; i < nDisks - 1; i++)

  for (i = 0; i < nDisks; i++){
  // CHECK FOR MAX FLIGHT CONDITION
        for (j = 0; j < DIM; j++){

              if ( fabs(collArray[i].time * y[DIM*(i+nDisks)+j]) >↩
                  maxFlightDblArray[j] ){
                  // IF THE DISTANCE TRAVELED IN EITHER SPATIAL ↩
                     DIRECTION BEFORE A VALID COLLISION INVOLVING ↩
                     DISK i EXCEEDS THE MAXIMUM FLIGHT CONDITION ↩
                     maxFlight, THEN SET THE COLLISION TIME AND ↩
                     PARTNER FOR DISK i ACCORDINGLY
                  collArray[i].time    = fabs(maxFlightDblArray[j]↩
                      / y[DIM*(i+nDisks)+j]);
                  collArray[i].partner = maxFlight;
              } // END if ( fabs(collArray[i].time * y[DIM*(i+↩
                 nDisks)+j]) > maxFlightDblArray[j] )

        } // END for (j = 0; j < DIM; j++)
  } // END for (i = 0; i < nDisks; i++)

  // EXIT NORMALLY
  return;
}
```

accumulateCLVlocalizationWidth

This routine accumulates the localization width of the covariant vectors output from Ginelli's method in **computeCLVs()**. The code is very nearly a direct copy of the **accumulateLocalizationWidth()**. For simplicity of implementation in the **computeCLVs()** routine, the vector whose data is to be accumulated **v**, along with the number of samples to average over **numSamples**, are input to the routine directly (rather than as globals).

```
//
//  FUNCTION:  accumulateCLVlocalizationWidth
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function accumulates states for localization width ↩
    computation.
//
//  Reference: Taniguchi, T. and Morriss G.P.  "Localized ↩
    behavior in the Lyapunov vectors for quasi-one-dimensional ↩
    many-hard-disk systems".  Phys. Rev. E 68, 046203 (2003)
//
//  INPUTS:
//  V- column-packed matrix containing covariant Lyapunov ↩
    vectors at current timestep
//  numSamples- number of samples to perform average over
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  DIM
//  nDisks
//  phaseDim
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//  clvStats.cumPlaceholder (accumulated value of individual ↩
    disk contributions to each covariant tangent vector norm is ↩
    updated)
//  clvStats.avgPlaceholder (averaged value of individual disk ↩
    contributions to each covariant tangent vector norm is ↩
    updated)
//
//  REVISION HISTORY:
//  Dinius, J.       Copied from accumulateLocalizationWidth() ↩
        11/02/13
//  Dinius, J.       Cleaned up for dissertation release ↩
              11/25/13
//
void accumulateCLVlocalizationWidth(vector<double> V, int ↩
    numSamples)
{
    double gammaSq; // variable to hold each disks contribution ↩
        to the (squared) norm of each tangent vector
    int i,j;        // loop indices
    int cnt;        // disk number counter

    for (i = 0; i < phaseDim; i++){

        // FOR EACH TANGENT VECTOR:

        // RESET CURRENT DISK INDEX TO 0
        cnt = 0;

        for (j = 0; j < DIM*nDisks; j+=2){

            // FOR EACH DISK:

            // COMPUTE CURRENT DISK (INDEXED BY cnt) ↩
                CONTRIBUTION TO THE (SQUARED) NORM OF THE CURRENT↩
                 TANGENT VECTOR:  THIS CONTRIBUTION IS THE SUM OF↩
                 THE SQUARE OF EACH OF THE 4 CONTRIBUTIONS FROM ↩
                THE CURRENT DISK (X,Y,Px, Py).  EQUATION (3) FROM↩
                 REFERENCE
            gammaSq = pow(V.at(i*phaseDim+j),2.0) + pow(V.at(i*↩
                phaseDim+j+1),2.0) + pow(V.at(DIM*nDisks+i*↩
                phaseDim+j),2.0) + pow(V.at(DIM*nDisks+i*phaseDim↩
                +j+1),2.0);

            // ADD TO ACCUMULATOR THE CURRENT CONTRIBUTION OF ↩
                THE "ENTROPY-LIKE" FACTOR FROM REFERENCE.  ↩
                ACCUMULATION STEP IN EQUATION (8) FROM REFERENCE ↩
                IMPLEMENTATION
            clvStats.cumPlaceholder[i].at(cnt) += gammaSq * log(↩
                gammaSq);

            // COMPUTE CURRENT TIME-AVERAGE OF "ENTROPY-LIKE" ↩
                FACTOR FROM REFERENCE.  TIME-AVERAGING STEP IN ↩
                EQUATION (8) FROM REFERENCE IMPLEMENTATION
            clvStats.avgPlaceholder[i].at(cnt)  = clvStats.↩
                cumPlaceholder[i].at(cnt) / ( (double)numSamples ↩
                );

            // INCREMENT DISK COUNTER
```

```
                cnt += 1;
            } // END for (j = 0; j < DIM*nDisks; j+=2

        } // END for (i = 0; i < phaseDim; i++)

    // EXIT NORMALLY
    return;
}
```

computeCLVlocalizationWidth

This routine computes the average localization width [91] of the covariant vectors output from Ginelli's method in **computeCLVs()**. The code is very nearly a direct copy of **computeLocalizationWidth()**. The declaration of this routine is to simplify processing in **computeCLVs()**. The only difference is that the **clvStats** global is used rather than **stats**. This routine computes the localization width of the covariant Lyapunov vectors which are output from Ginelli's method.

```
//
//  FUNCTION:  computeCLVlocalizationWidth
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//  Function performs exponential of sum of averaged states for ↩
    localization width computation.
//
//  Reference: Taniguchi, T. and Morriss G.P.  "Localized ↩
    behavior in the Lyapunov vectors for quasi-one-dimensional ↩
    many-hard-disk systems".  Phys. Rev. E 68, 046203 (2003)
//
//  INPUTS:
//  (none)
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  phaseDim
//  nDisks
//   clvStats.avgPlaceholder
//
//   GLOBAL OUTPUTS (UPDATED DURING CALL):
//   clvStats.Sn (sum of entropy-like quantity Sn for each ↩
    covariant tangent vector is computed)
//   clvStats.Wn (exponential of clvStats.Sn/nDisks is computed)
//
//   REVISION HISTORY:
//   Dinius, J.       Copied from computeLocalizationWidth() ↩
    routine     11/02/13
//   Dinius, J.       Cleaned up for dissertation release ↩
                   11/25/13
//
void computeCLVlocalizationWidth(void)
{
    int i,j;    // loop indices
    double sum; // accumulator for sum in Eqn (8) from reference

    for (i = 0; i < phaseDim; i++){

        // FOR EACH TANGENT VECTOR:

        // INITIALIZE THE TERM IN THE SUM FROM EQN (8) IN ↩
           REFERENCE TO ZERO
        sum = 0;

        // SUM EACH DISK'S CONTRIBUTION TO "ENTROPY-LIKE" TERM (↩
           MINUS SIGN BELOW TO REFLECT SIGN-CONVENTION IN ↩
           EQUATION (8) FROM REFERENCE).  THIS IS THE SUMMATION ↩
           STEP IN EQUATION (8) FROM REFERENCE.
        for (j = 0; j < nDisks; j+=1){
            sum -= clvStats.avgPlaceholder[i].at(j);
        } // END for (j = 0; j < nDisks; j+=1)

        // STORE SUM OF TIME-AVERAGE TERM IN Sn IN EQUATION (8) ↩
           FROM REFERENCE.  THIS IS THE VECTORIZATION OF THE ↩
           QUANTITY Sn IN EQUATION (8) FROM REFERENCE.
        clvStats.Sn.push_back(sum);

        // COMPUTE LOCALIZATION WIDTH EXPONENTIAL.  THIS IS THE ↩
           IMPLEMENTATION OF EQUATION (9) FROM REFERENCE.
        clvStats.Wn.push_back(exp(sum)/( (double)nDisks) );
    } // END for (i = 0; i < phaseDim; i++)

```

```
  // EXIT NORMALLY
    return;
}
```

computeCLVavgCosTheta

This routine computes the average angle between position and momentum perturbation components of the covariant Lyapunov vectors output from Ginelli's method in **computeCLVs()**. The code is very nearly a direct copy of the **computeAvgCosTheta()**. For simplicity of implementation in the **computeCLVs()** routine, the vector whose data is to be accumulated **v**, along with the number of samples to average over **numSamples**, are input to the routine directly (rather than as globals).

```
//
//  FUNCTION:  computeAvgCosTheta
//  MODULE:    diskDynamics
//
//  DESCRIPTION:
//   Function computes the average of the cosine of the angle ↩
    between position and momentum perturbations in tangent ↩
    vectors
//
//   Reference:
//  Eckmann, J.P. et. al. "Lyapunov Modes in Hard-Disk Systems".↩
      Journal of Stat. Phys., Vol. 118, Nos. 5/6.  March 2005.  ↩
    Pgs. 813-847.
//
//  INPUTS:
//  V- column-packed matrix containing covariant Lyapunov ↩
    vectors at current timestep
//  numSamples- number of samples to perform average over
//
//  OUTPUTS:
//  (none)
//
//  GLOBAL INPUTS (UNCHANGED DURING CALL):
//  DIM
//  nDisks
//  phaseDim
```

```
//
//  GLOBAL OUTPUTS (UPDATED DURING CALL):
//   clvStats.cumCosTheta (accumulated value of the cosine of the↩
     angle between the position and momentum offset
//                           components of each covariant tangent ↩
   vector is updated)
//   clvStats.avgCosTheta (averaged value of the cosine of the ↩
   angle between the position and momentum offset
//                           components of each covariant tangent ↩
   vector is updated)
//
//  REVISION HISTORY:
//   Dinius, J.       Copied from computeAvgCosTheta()      ↩
   11/02/13
//   Dinius, J.       Cleaned up for dissertation release  ↩
   11/25/13
//
void computeCLVavgCosTheta(vector<double> V, int numSamples)
{

    int i,j;          // loop indices
    double nrmSqPos; // norm squared of position perturbation ↩
        contribution to individual tangent vector
    double nrmSqMom; // norm squared of momentum perturbation ↩
        contribution to individual tangent vector
    double dot;      // dot product of vector of position ↩
        perturbation components of individual tangent vector with↩
         vector of momentum perturbation components of the same ↩
        individual tangent vector

    //int numSamples = (Time − TimeStartCollection)/ (stepSize *↩
         (double)(nOrtho) ); // number of samples of localization↩
         width taken (equivalently, the number of times this ↩
        routine has been called)

    for (i = 0; i < phaseDim; i++){

        // FOR EACH TANGENT VECTOR:

        // INITIALIZE ACCUMULATORS FOR NORM SQUARED AND DOT ↩
           PRODUCT TERMS
        nrmSqPos = 0.;
        nrmSqMom = 0.;
        dot = 0;
```

```

         for (j = 0; j < DIM*nDisks; j++){
            // ADD THE CONTRIBUTION OF THE CURRENT INDEX TO THE ↩
               NORM SQUARED AND DOT PRODUCT TERMS
            nrmSqPos += pow(V.at(i*phaseDim+j),2.);
            nrmSqMom += pow(V.at(DIM*nDisks+i*phaseDim+j),2.);
            dot      += V.at(i*phaseDim+j) * V.at(DIM*nDisks+i*↩
               phaseDim+j);
         } // END for (j = 0; j < DIM*nDisks; j++)

         // COMPUTE (AND ADD TO ACCUMULATOR) THE COMPUTED VALUE ↩
            OF THE COSINE OF THE ANGLE BETWEEN THE POSITION AND ↩
            MOMENTUM COMPONENTS OF THE iTH TANGENT VECTOR
         clvStats.cumCosTheta.at(i) += dot/(sqrt(nrmSqPos*↩
            nrmSqMom));

         // COMPUTE THE TIME-AVERAGE OF THE COSINE OF THE ANGLE ↩
            BETWEEN THE POSITION AND MOMENTUM COMPONENTS OF THE ↩
            iTH TANGENT VECTOR
         clvStats.avgCosTheta.at(i) = clvStats.cumCosTheta.at(i)↩
            /( (double)numSamples );

      } // END for (i = 0; i < phaseDim; i++)

   // EXIT NORMALLY
      return;
}
```

B.7 File I/O Description

This section is broken into two subsections. Subsection B.7.1 gives examples of valid input files for use with the **diskDynamics** routine and Subsection B.7.2 gives sample Matlab® function files for reading in formatted data from a **diskDynamics** run. An example for how to read-in the optional output file for the velocity autocorrelation data is provided; from which the user can tailor this example to read in other optional output files.

B.7.1 Sample Input Files

This section gives specific examples of input files that can be used to run **diskDynamics** from the command line. There are examples that execute all of the different functionalities defined by inputs.

Tumble Disks

inputSample.dat

```
1               /* tumble? */
0.5            /* uniform distribution interval width between disk ↩
    tumbles */
1.0               /* beta */
0               /* write Alpha? (1=YES) */
0.001           /* stepsize                  */
500000            /* nsteps                  */
50               /* northo                   */
1000               /* noutput                  */
64             /* number of disks          */
0.5             /* rho                       */
256               /* nlya                    */
1.0      /* aspect ratio             */
0                /* compute batch average */
100              /* batch size */
1                /* write out all modes */
0                /* write velocities for vacf processing */
N64_B1p0.bin        /* output file                  */
0     /* read state from file */
0     /* Store data for, and process, covariant vectors? */
```

This example specifies to execute tumbling dynamics with 0.5 as the interval width between disk tumbles (see Subsection *tumble*). The initial condition is not read-in from an input file and covariant Lyapunov vectors will not be computed.

Write Out All Modes

inputSample0.dat

```
1  0            /* tumble? */
2  1.0            /* beta */
3  0            /* write Alpha? (1=YES) */
4  0.001        /* stepsize                 */
5  500000         /* nsteps                 */
6  50            /* northo                  */
7  1000            /* noutput                */
8  64          /* number of disks        */
9  0.5          /* rho                      */
10 256            /* nlya                     */
11 1.0     /* aspect ratio            */
12 0             /* compute batch average */
13 100           /* batch size */
14 1             /* write out all modes */
15 1             /* write velocities for vacf processing */
16 100           /* time before recording vacf data (Settling) */
17 5             /* number of time steps between vacf captures */
18 100           /* time to record vacf data */
19 N64_B1p0.bin       /* output file                   */
20 0    /* read state from file */
21 1    /* Store data for, and process, covariant vectors? */
22 1000.000000    /* Time before Ginelli method */
```

This input file will setup a run with 64 disks that computes 256 Lyapunov exponents (all of them, does not compute batch averages, writes out all modes that are computed (limited by **nlya**) and computes and writes out velocity autocorrelation data (Lines 16-19). The initial condition is not read-in from a file, and covariant Lyapunov vectors are computed after 1000 time units have elapsed.

Select Modes to Output

inputSample1.dat

```
1  0            /* tumble? */
2  1.0            /* beta */
3  0            /* write Alpha? (1=YES) */
4  0.001        /* stepsize                 */
5  500000         /* nsteps                 */
6  50            /* northo                  */
7  1000            /* noutput                */
```

```
64           /* number of disks        */
 0.5          /* rho                     */
256            /* nlya                     */
 1.0      /* aspect ratio            */
0              /* compute batch average */
100            /* batch size */
0              /* write out all modes */
0              /* write out range of modes */
3              /* modes to read */
1              /* write out mode 1 */
114        /* write out mode 114 */
256        /* write out mode 256 */
0              /* write velocities for vacf processing */
N64_B1p0.bin       /* output file                    */
1    /* read state from file */
sampleInput.bin /* input file to read initial condition from */
0    /* Store data for, and process, covariant vectors? */
```

This input file will setup a run with 64 disks that computes 256 Lyapunov exponents (all of them), does not compute batch averages, writes out 3 modes (in addition to the state): modes 1, 114 and 256. This input file specifies to not write out velocity autocorrelation data. The number preceding the statement: **/* modes to read */** must be the same as the number of proceeding lines that specify the data to write out. For example, the following code segment will force an issue (not necessarily a run-time error):

```
0              /* write out all modes */
0              /* write out range of modes */
2              /* modes to read */
1              /* write out mode 1 */
128        /* write out mode 128 */
256        /* write out mode 256 */
```

This segment will result in reading in 256 as the next input (**writeVACF**) which will alter the correct input sequence and cause issues. Similarly, the following will force a different issue:

```
0              /* write out all modes */
```

```
0               /* write out range of modes */
4               /* modes to read */
1               /* write out mode 1 */
128         /* write out mode 128 */
256         /* write out mode 256 */
```

This segment will look at the input for **writeVACF** as a mode number to output, again altering the correct input sequence. The input file also specifies to read in the initial condition (on the phase space trajectory values) from the input file **sampleInput.bin**.

Write Out a Range of Modes

inputSample2.dat

```
0                /* tumble? */
1.0                /* beta */
0                /* write Alpha? (1=YES) */
0.001            /* stepsize                 */
500000             /* nsteps                  */
50                /* northo                 */
1000               /* noutput                 */
64              /* number of disks        */
0.5              /* rho                      */
256                /* nlya                     */
1.0       /* aspect ratio            */
0                 /* compute batch average */
100               /* batch size */
0                 /* write out all modes */
1                 /* write out range of modes (and state) */
125               /* lower index of write */
130               /* upper index of write */
0                 /* write velocities for vacf processing */
N64_B1p0.bin           /* output file                  */
0     /* read state from file */
0     /* Store data for, and process, covariant vectors? */
```

This input file sets up the same run as *inputSample1.dat* presented above, but instead of specifying individual mode numbers to write out, a range of modes is specified; modes 125-130 will be written out in addition to the system state.

Matlab® Function to Create Input File

To facilitate fast creation of input files, the following Matlab® script is very useful. Each configuration setting possible can be obtained using different arguments to the **writeDatFile()** function:

```
function outfile = writeDatFile(tmb,beta,writeAlpha,stepSize,↩
    numSteps,nOrtho,nOutput,nDisks,rho,nExp,asp,batchAvg,allModes↩
    ,rangeModes,chooseModes,vacf,msg,initState,procCLV,clvTime,↩
    botIdx,topIdx,dtTmbl,outModes,settleTime,modVacf,captureTime)

if (~exist('initState','var')); initState = 0; end
if (~exist('procCLV','var')); procCLV = 0; end
if (~exist('dtTmbl','var')); dtTmbl = .5; end
if (~exist('outModes','var')); outModes = 0; end
if (~exist('settleTime','var')); settleTime = 100; end
if (~exist('modVacf','var')); modVacf = 20; end
if (~exist('captureTime','var')); captureTime = 100; end
if (~exist('botIdx','var')); botIdx = 1; end
if (~exist('topIdx','var')); topIdx = 1; end
if (~exist('clvTime','var')); clvTime = 1000; end

if (beta > .999)
    strb = '1p0';
elseif (beta < .099)
    strb = ['0p0' num2str(round(100*beta))];
else
    strb = ['0p' num2str(round(100*beta))];

end
if (rho > .999)
    strr = '1p0';
else
    strr = ['0p' num2str(floor(100*rho))];
end
if (asp > .999)
    stra = '1p0';
else
    stra = ['0p' num2str(floor(100*asp))];
end
outfile = ['paraTest_N' num2str(nDisks) '_rho' strr '_asp' stra ↩
    '_beta' strb '_' msg '.dat'];
```

```
fid = fopen(outfile,'w');

fprintf(fid,'%d \t /* tumble? */\n',tmb);
if tmb
    fprintf(fid,'%f \t /* time interval between tumbles */\n',dtTmbl);
end
fprintf(fid,'%f \t /* beta */\n',beta);
fprintf(fid,'%f \t /* 1=writeAlpha */\n',writeAlpha);
fprintf(fid,'%f \t /* stepsize */\n',stepSize);
fprintf(fid,'%d \t /* nsteps */\n',numSteps);
fprintf(fid,'%d \t /* steps between ortho */\n',nOrtho);
fprintf(fid,'%d \t /* nOutput */\n',nOutput);
fprintf(fid,'%d \t /* ndisks */\n',nDisks);
fprintf(fid,'%f \t /* density */\n',rho);
fprintf(fid,'%d \t /* num exps to compute */\n',nExp);
fprintf(fid,'%f \t /* aspect ratio */\n',asp);
fprintf(fid,'%d \t /* compute batch avg */\n',batchAvg);
fprintf(fid,'%d \t /* batch size */\n',100);
fprintf(fid,'%d \t /* all modes */\n',allModes);
if ~allModes
    fprintf(fid,'%d \t /* range modes */\n',rangeModes);
    if rangeModes
        fprintf(fid,'%d \t /* range bottom index */\n',botIdx);
        fprintf(fid,'%d \t /* range top index */\n',topIdx);
    else
        fprintf(fid,'%d \t /* select modes to output */\n',chooseModes);
        if chooseModes
            for i = 1:length(outModes)
                fprintf(fid,'%d \t\t /* mode no. %d to write out */\n',outModes,i);
            end
        end
    end
end

fprintf(fid,'%d \t /* write data for vacf? */\n',vacf);
if vacf
    fprintf(fid,'%d \t\t /* settle time? */\n',settleTime);
    fprintf(fid,'%d \t\t /* mod steps */\n',modVacf);
    fprintf(fid,'%d \t\t /* capture time */\n',captureTime);
end
fprintf(fid,'%s \t /* output file */\n',['N' num2str(nDisks) '↩
_rho' strr '_asp' stra '_' msg '_' strb '.bin']);
fprintf(fid,'%d \t /* read state from file */\n',initState);

if (initState)
    %fprintf(fid,'%s \t /* input file with initial state */\n',['input_N' num2str(nDisks) '_rho' strr '_asp' stra '_' msg '_' strb '.bin']);
    fprintf(fid,'%s \t /* input file with initial state */\n',['samp' num2str(nDisks) '_' strr '.bin']);
end

fprintf(fid,'%d \t /* Store data for, and process, covariant vectors? */\n',procCLV);

if (procCLV)
    fprintf(fid,'%f \t /* Time before Ginelli method */\n',clvTime);
end

fclose(fid);
```

It is important to note here that, if reading the initial data from an input file, the filename must match the name of an appropriately-formatted input file expected in the **initialize()** routine (see *initialize*).

B.7.2 Reading the Data in Matlab®

The data is written to a binary file (to save disk space). To unpack the binary file and store to a Matlab® **.mat** file, the function **readDiskFile** is used. This function uses Matlab® standard routines to read-in and plot data from the output file written by **diskDynamics**. The source code is:

```
function readDiskFile(fname,saveFigs,recordStartTime,readModes)

if ~exist('fname','var') fname = 'paraTest.dat'; end %default: try standard filename
if ~exist('saveFigs','var') saveFigs = 0; end %default: don't save figures
if ~exist('recordStartTime','var') recordStartTime = 0; end %default: store from beginning of file
if ~exist('readModes','var') readModes = 0; end %default: don't store modes

eval(['a = textread(''' fname ''',''%s'');']);

if (isempty(a))
    fprintf('There is no input data in the file given.  Exiting...\n');
    return
end

indFlag = strfind(a,'/*');

j = 0;
for i = 1:length(indFlag)
    if ~isempty(indFlag{i})
        j = j+1;
        dataPt{j} = a{i-1}; %parameters of interest are just before this expression
    end
end

if ( strcmp(dataPt{j-1},'1'))
    upperInd = j-4;
elseif ( strcmp(class(dataPt{j}),'0'))
    upperInd = j-3;
else
    upperInd = j-3;
end
%upperInd = j-4; %j-3 when using input initial conditions
%upperInd = j-2; %j-2 when not using input initial conditions

for i = 1:upperInd %exclude second to last point b/c it is non-numeric
    data(i) = str2num(dataPt{i});
end
%mapping of input data to parameters of interest for reading output
if (data(1) == 0)
    numTimeSteps = data(5)/data(7)-1;
    numExp       = data(10);
    numDisks     = data(8);
    writeAllModes = data(14);
    if (writeAllModes == 1)
        numStateVecs = numExp + 1;
        fprintf('Reading in data for state and all modes...\n');
    else
        writeRangeModes = data(15);
        if (writeRangeModes)
            bottomIndx = data(16);
            topIndx    = data(17);
            numStateVecs = topIndx - bottomIndx + 2;
            ind(1) = 1;
            ind(2:numStateVecs) = bottomIndx:topIndx;
            fprintf('Reading in data for state and modes %d to %d...\n',bottomIndx,topIndx);
        else
            numStateVecs = data(16)+1;
            ind(1) = 1;
            fprintf('Reading in data for state and \n');
            for i = 2:numStateVecs
                ind(i) = data(16+i-1);
                if (i ~= numStateVecs)
                    fprintf('\t Mode %d\n',ind(i));
                else
                    fprintf('\t Mode %d...\n',ind(i));
                end
            end
        end
    end
    betak        = data(2);
else % +1 is the index offset from tumble inputs
    numTimeSteps = data(5+1)/data(7+1)-1;
    numExp       = data(10+1);
    numDisks     = data(8+1);
    writeAllModes = data(14+1);
    if (writeAllModes == 1)
        numStateVecs = numExp + 1;
        fprintf('Reading in data for state and all modes...\n');
    else
        writeRangeModes = data(15+1);
        if (writeRangeModes)
            bottomIndx = data(16+1);
            topIndx    = data(17+1);
```

```
                numStateVecs = topIndx - bottomIndx + 2;
                ind(1) = 1;
                ind(2:numStateVecs) = bottomIndx:topIndx;
                fprintf('Reading in data for state and modes %d to %↩
                    d...\n',bottomIndx,topIndx);
            else
                numStateVecs = data(16+1)+1;
                ind(1) = 1;
                fprintf('Reading in data for state and \n');
                for i = 2:numStateVecs
                    ind(i) = data(16+1+i-1);
                    if (i ~= numStateVecs)
                        fprintf('\t Mode %d\n',ind(i));
                    else
                        fprintf('\t Mode %d...\n',ind(i));
                    end
                end
            end
        end
        betak        = data(2+1);
end

dim            = 2;
aspRatio       = data(11);
density        = data(9);
Lx = sqrt(numDisks * pi / 4.0 / density / aspRatio);
Ly = Lx * aspRatio;

fileInd = upperInd+1; %when using input initial conditions
%fileInd = j-1: %when not
eval(['fid = fopen(''' dataPt{fileInd} ''',''r'');']);
cntr = 0;
cntrSave = 1;
while (cntr <= (numTimeSteps-99))%100 is time before writing out↩
     starts
    try
        cntr = cntr+1;
        t(cntrSave) = fread(fid,1,'double');

        for i = 1:numStateVecs
            tmp = fread(fid,2*dim*numDisks,'double');
            if ( (i == 1) || (i >= 2 && readModes) )
                y(cntrSave,i,:) = tmp;
            end
        end
        lya(cntrSave,:) = fread(fid,numExp,'double');
        icoll(cntrSave) = fread(fid,1,'int');
        if (t(cntrSave) >= recordStartTime)
            cntrSave = cntrSave + 1;
        end

    catch
        cntr = cntr-1;
        break
    end
end
fprintf('End time in data file: t = %f\n',t(cntrSave-1));

jj = j;
%read-in localization measure stuff
locMeas = fread(fid,numExp,'double');
%read-in average cosine angle stuff
avgCos = fread(fid,numExp,'double');
eval(['save vacf_' dataPt{j}(1:end-4) ' -v7.3']);
```

References

[1] BJ Alder and TE Wainwright. Phase transition for a hard sphere system. *The Journal of Chemical Physics*, 27(5):1208–1209, 1957.

[2] BJ Alder and TE Wainwright. Decay of the velocity autocorrelation function. *Physical review A*, 1(1):18, 1970.

[3] Vladimir I. Arnold. *Ordinary Differential Equations*. Springer, 2006.

[4] John A Barker and Douglas Henderson. Perturbation theory and equation of state for fluids. ii. a successful theory of liquids. *The Journal of Chemical Physics*, 47:4714, 1967.

[5] 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 1: Theory. *Meccanica*, 15(1):9–20, 1980.

[6] Jairo Bochi and Marcelo Viana. The lyapunov exponents of generic volume-preserving and symplectic maps. *Annals of mathematics*, pages 1423–1485, 2005.

[7] Ludwig Boltzmann. Zur theorie der elastischen nachwirkung. *Annalen der Physik*, 241(11):430–432, 1878.

[8] Hadrien Bosetti and Harald A Posch. Covariant lyapunov vectors for rigid disk systems. *Chemical physics*, 375(2):296–308, 2010.

[9] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[10] Leonid Abramovich Bunimovich, Yakov Grigor'evich Sinai, and Nikolai Ivanovich Chernov. Statistical properties of two-dimensional hyperbolic billiards. *Russian Mathematical Surveys*, 46(4):47–106, 1991.

[11] Nikolai Ivanovich Chernov. New proof of sinai's formula for the entropy of hyperbolic billiard systems. application to lorentz gases and bunimovich stadiums. *Functional Analysis and Its Applications*, 25(3):204–219, 1991.

[12] Geon Ho Choe. *Computational Ergodic Theory*, volume 13 of *Algorithms and Computation in Mathematics*. Springer, 2005.

[13] Tony Chung, Daniel Truant, and Gary P Morriss. Lyapunov modes as fields. *Physical Review E*, 83(4):046216, 2011.

[14] Pierre Collet and J-P Eckmann. Liapunov multipliers and decay of correlations in dynamical systems. *Journal of statistical physics*, 115(1-2):217–254, 2004.

[15] C. Dellago. Private communication, 2011.

[16] Ch. Dellago, H. A. Posch, and W. G. Hoover. Lyapunov instability in a system of hard disks in equilibrium and nonequilibrium steady states. *Phys. Rev. E*, 53:1485–1501, Feb 1996.

[17] CP Dettmann and GP Morriss. Proof of lyapunov exponent pairing for systems at constant kinetic energy. *Physical Review-Section E-Statistical Physics Plasma Fluids Related Interdiscpl Topics*, 53(6):R5545, 1996.

[18] J.R. Dorfman. *An Introduction to Chaos in Nonequilibrium Statistical Mechanics*, volume 14 of *Cambridge Lecture Notes in Physics*. Cambridge University Press, 1999.

[19] JR Dorfman and EGD Cohen. Velocity correlation functions in two and three dimensions. *Physical Review Letters*, 25(18):1257, 1970.

[20] J-P Eckmann and David Ruelle. Ergodic theory of chaos and strange attractors. *Reviews of modern physics*, 57(3):617, 1985.

[21] Jean-Pierre Eckmann, Christina Forster, HaraldA. Posch, and Emmanuel Zabey. Lyapunov modes in hard-disk systems. *Journal of Statistical Physics*, 118(5-6):813–847, 2005.

[22] Jean-Pierre Eckmann and Omri Gat. Hydrodynamic lyapunov modes in translation-invariant systems. *Journal of Statistical Physics*, 98(3-4):775–798, 2000.

[23] Jerome J Erpenbeck. Transport coefficients of hard-sphere mixtures: Theory and monte carlo molecular-dynamics calculations for an isotopic mixture. *Physical Review A*, 39(9):4718, 1989.

[24] Jerome J Erpenbeck and William W Wood. Molecular-dynamics calculations of the velocity-autocorrelation function. methods, hard-disk results. *Physical Review A*, 26(3):1648, 1982.

[25] Jerome J Erpenbeck and William W Wood. Self-diffusion coefficient for the hard-sphere fluid. *Physical Review A*, 43(8):4254, 1991.

[26] Sergey V Ershov and Alexei B Potapov. On the concept of stationary lyapunov basis. *Physica D: Nonlinear Phenomena*, 118(3):167–198, 1998.

[27] Denis J Evans, EGD Cohen, and Gary P Morriss. Viscosity of a simple fluid from its maximal lyapunov exponents. *Physical Review A*, 42(10):5990, 1990.

[28] RM Everson. Chaotic dynamics of a bouncing ball. *Physica D: Nonlinear Phenomena*, 19(3):355–383, 1986.

[29] Gerald B Folland and GB Folland. *Real analysis: modern techniques and their applications*, volume 361. Wiley New York, 1999.

[30] C. Forster. Lyapunov Instability of Two-Dimensional Fluids. Master's thesis, University of Vienna, 2002.

[31] Christina Forster and Harald A Posch. Lyapunov modes in soft-disk fluids. *New Journal of Physics*, 7(1):32, 2005.

[32] Gary Froyland, Thorsten Hüls, Gary P Morriss, and Thomas M Watson. Computing covariant lyapunov vectors, oseledets vectors, and dichotomy projectors: A comparative numerical study. *Physica D: Nonlinear Phenomena*, 2012.

[33] Gary Froyland, Simon Lloyd, and Anthony Quas. Coherent structures and isolated spectrum for perron-frobenius cocycles. *Ergodic Theory and Dynamical Systems*, 30(03):729–756, 2010.

[34] Jan Froyland and Knut H Alfsen. Liapunov-exponent spectra for the lorenz model. *Physical Review A*, 29:2928–2931, 1984.

[35] Pierre Gaspard. *Chaos, Scattering and Statistical Mechanics*, volume 9 of *Cambridge Nonlinear Science Series*. Cambridge University Press, 1998.

[36] F Ginelli, P Poggi, A Turchi, H Chaté, R Livi, and A Politi. Characterizing dynamics with covariant lyapunov vectors. *Physical review letters*, 99(13):130601, 2007.

[37] Francesco Ginelli, Hugues Chaté, Roberto Livi, and Antonio Politi. Covariant lyapunov vectors. *Journal of Physics A: Mathematical and Theoretical*, 46(25):254005, 2013.

[38] Rafal Goebel, Ricardo G Sanfelice, and Andrew R Teel. *Hybrid Dynamical Systems: modeling, stability, and robustness*. Princeton University Press, 2012.

[39] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[40] John Guckenheimer and Philip Holmes. Nonlinear oscillations, dynamical systems, and bifurcations of vector fields. 1983.

[41] James M Haile. *Molecular dynamics simulation: elementary methods*. John Wiley & Sons, Inc., 1992.

[42] Jean-Pierre Hansen and Ian R McDonald. *Theory of simple liquids*. Access Online via Elsevier, 1990.

[43] Li He, Xiangwei Liu, and Gilbert Strang. Laplacian eigenvalues of growing trees. In *Conf. on Math. Theory of Networks and Systems*. Citeseer, 2000.

[44] Li He, Xiangwei Liu, and Gilbert Strang. Trees with cantor eigenvalue distribution. *Studies in Applied Mathematics*, 110(2):123–138, 2003.

[45] Michel Hénon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50(1):69–77, 1976.

[46] Morris W. Hirsch, Stephen Smale, and Robert L. Devaney. *Differential Equations, Dynamical Systems, and an Introduction to Chaos*. Elsevier, 2013.

[47] Wm.G. Hoover, HaraldA. Posch, Christina Forster, Christoph Dellago, and Mary Zhou. Lyapunov modes of two-dimensional many-body systems; soft disks, hard disks, and rotors. *Journal of Statistical Physics*, 109(3-4):765–776, 2002.

[48] Thorsten Hüls. Numerical computation of dichotomy rates and projectors in discrete time. *Discrete Contin. Dyn. Syst. Ser. B*, 12(1):109–131, 2009.

[49] Thorsten Hüls. Computing sacker-sell spectra in discrete time dynamical systems. *SIAM Journal on Numerical Analysis*, 48(6):2043–2064, 2010.

[50] Dongchul Ihm, Young-Han Shin, Jae-Weon Lee, and Eok Kyun Lee. Correlation between kolmogorov-sinai entropy and self-diffusion coefficient in simple fluids. *Physical Review E*, 67(2):027205, 2003.

[51] James L Kaplan and James A Yorke. Chaotic behavior of multidimensional difference equations. In *Functional differential equations and approximation of fixed points*, pages 204–227. Springer, 1979.

[52] Mehran Kardar. *Statistical physics of particles*. Cambridge University Press, 2007.

[53] Anatole Katok and Boris Hasselblatt. *Modern Theory of Dynamical Systems*, volume 54 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 1997.

[54] John Frank Charles Kingman. Subadditive ergodic theory. *The annals of Probability*, pages 883–899, 1973.

[55] Leonid B Koralov, Yakov G Sinai, et al. *Theory of probability and random processes*. Springer-Verlag Berlin Heidelberg, 2007.

[56] Ryogo Kubo. *Statistical Mechanics*. Elsevier, 1965.

[57] John M Lee. *Riemannian manifolds: an introduction to curvature*, volume 176. Springer, 1997.

[58] John M Lee. *Introduction to smooth manifolds*, volume 218. Springer, 2012.

[59] Joceline Lega. Collective behaviors in two-dimensional systems of interacting particles. *SIAM Journal on Applied Dynamical Systems*, 10(4):1213–1231, 2011.

[60] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[61] Michel Mareschal and Sean McNamara. Lyapunov hydrodynamics in the dilute limit. *Physica D: Nonlinear Phenomena*, 187(1):311–325, 2004.

[62] Sean McNamara and Michel Mareschal. Origin of the hydrodynamic lyapunov modes. *Physical Review E*, 64(5):051103, 2001.

[63] Neil H Mendelson, Adrienne Bourque, Kathryn Wilkening, Kevin R Anderson, and Joseph C Watkins. Organized cell swimming motions in bacillus subtilis colonies: patterns of short-lived whirls and jets. *Journal of bacteriology*, 181(2):600–609, 1999.

[64] J.L. Meriam. *Dynamics*. Wiley, 1975.

[65] L Morino. Helmholtz decomposition revisited: vorticity generation and trailing edge condition. *Computational Mechanics*, 1(1):65–90, 1986.

[66] Gary Morriss and Daniel Truant. Finite-size scaling of lyapunov spectra for quasi-one-dimensional hard discs. *Molecular Simulation*, 37(4):277–283, 2011.

[67] Gary P Morriss and Daniel Truant. Characterization of the lyapunov modes for an equilibrium quasi-one-dimensional system. *Journal of Statistical Mechanics: Theory and Experiment*, 2009(02):P02029, 2009.

[68] Gary P Morriss and Daniel P Truant. A review of the hydrodynamic lyapunov modes of hard disk systems. *Journal of Physics A: Mathematical and Theoretical*, 46(25):254010, 2013.

[69] GP Morriss. Localization properties of covariant lyapunov vectors for quasi-one-dimensional hard disks. *Physical Review E*, 85(5):056219, 2012.

[70] Valery Iustinovich Oseledec. A multiplicative ergodic theorem. lyapunov characteristic numbers for dynamical systems. *Trans. Moscow Math. Soc*, 19(2):197–231, 1968.

[71] E. Ott. *Chaos in Dynamical Systems*. Cambridge University Press, 2002.

[72] Tao Pang. *An introduction to computational physics*. Cambridge University Press, 1997.

[73] Yakov Borisovich Pesin. Characteristic lyapunov exponents and smooth ergodic theory. *Russian Mathematical Surveys*, 32(4):55–114, 1977.

[74] Marco Pettini. *Geometry and Topology in Hamiltonian Dynamics and Statistical Mechanics*, volume 33 of *Interdisciplinary Applied Mathematics*. Springer, 2007.

[75] Jim Pitman and Songhwai Oh. Lecture 12: Subadditive ergodic theorem. http://stat-www.berkeley.edu/users/pitman/s205s03/lecture12.pdf, 2003. Lecture notes from Statistics 205b: Probability Theory.

[76] Dennis C Rapaport. *The art of molecular dynamics simulation*. Cambridge university press, 2004.

[77] David Ruelle. Ergodic theory of differentiable dynamical systems. *Publications Mathématiques de l'Institut des Hautes Études Scientifiques*, 50(1):27–58, 1979.

[78] David Ruelle. *Chaotic evolution and strange attractors*, volume 1. Cambridge University Press, 1989.

[79] David Saintillan and Michael J Shelley. Orientational order and instabilities in suspensions of self-locomoting rods. *Physical review letters*, 99(5):058102, 2007.

[80] David Saintillan and Michael J Shelley. Emergence of coherent structures and large-scale flows in motile suspensions. *Journal of The Royal Society Interface*, 9(68):571–585, 2012.

[81] Ricardo Sanfelice, David Copp, and Pablo Nanez. A toolbox for simulation of hybrid systems in matlab/simulink: hybrid equations (hyeq) toolbox. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 101–106. ACM, 2013.

[82] Ippei Shimada and Tomomasa Nagashima. A numerical approach to ergodic problem of dissipative dynamical systems. *Progress of Theoretical Physics*, 61(6):1605–1616, 1979.

[83] Yakov G Sinai. Dynamical systems with elastic reflections. *Russian Mathematical Surveys*, 25(2):137, 1970.

[84] Yakov Grigor'evich Sinai and Nikolai Ivanovich Chernov. Ergodic properties of certain systems of two-dimensional discs and three-dimensional balls. *Russian Mathematical Surveys*, 42(3):181–207, 1987.

[85] Eduardo D Sontag. *Mathematical control theory: deterministic finite dimensional systems*, volume 6. Springer, 1998.

[86] J Michael Steele. Kingmans subadditive ergodic theorem. *Ann. Inst. H. Poincaré*, 25:93–98, 1989.

[87] Steven H. Strogatz. *Nonlinear Dynamics and Chaos*. Westview, 1994.

[88] Domokos Szász and LA Bunimovich. *Hard ball systems and the Lorentz gas*, volume 101. Springer Berlin, 2000.

[89] Tooru Taniguchi, Carl P Dettmann, and Gary P Morriss. Lyapunov spectra of periodic orbits for a many-particle system. *Journal of Statistical Physics*, 109(3-4):747–764, 2002.

[90] Tooru Taniguchi and Gary P Morriss. Stepwise structure of lyapunov spectra for many-particle systems using a random matrix dynamics. *Physical Review E*, 65(5):056202, 2002.

[91] Tooru Taniguchi and Gary P Morriss. Localized behavior in the lyapunov vectors for quasi-one-dimensional many-hard-disk systems. *Physical Review E*, 68(4):046203, 2003.

[92] Tooru Taniguchi and Gary P Morriss. Dynamics of strongly localized lyapunov vectors in many-hard-disk systems. *Physical Review E*, 73(3):036208, 2006.

[93] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*. Number 50. Siam, 1997.

[94] Loring W Tu. *An introduction to manifolds*. Springer, 2011.

[95] Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford University Press, 2010.

[96] Peter Walters. *An Introduction to Ergodic Theory*, volume 79 of *Graduate Texts in Mathematics*. Springer, 2000.

[97] Stephen Wiggins. *Introduction to applied nonlinear dynamical systems and chaos*, volume 2. Springer, 2003.

[98] Alan Wolf, Jack B Swift, Harry L Swinney, and John A Vastano. Determining lyapunov exponents from a time series. *Physica D: Nonlinear Phenomena*, 16(3):285–317, 1985.

[99] Christopher L Wolfe and Roger M Samelson. An efficient method for recovering lyapunov vectors from singular vectors. *Tellus A*, 59(3):355–366, 2007.

[100] Hong-liu Yang and Günter Radons. When can one observe good hydrodynamic lyapunov modes? *Physical review letters*, 100(2):024101, 2008.

[101] Hong-liu Yang and Günter Radons. Comparison between covariant and orthogonal lyapunov vectors. *Physical Review E*, 82(4):046204, 2010.

[102] Lai-Sang Young. Ergodic theory of differentiable dynamical systems. In *Real and complex dynamical systems*, pages 293–336. Springer, 1995.

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

[104] Andrea R Zeni and Jason AC Gallas. Lyapunov exponents for a duffing oscillator. *Physica D: Nonlinear Phenomena*, 89(1):71–82, 1995.