INTRODUCTION TO THE MODELING AND ANALYSIS OF COMPLEX SYSTEMS



Hiroki Sayama
Binghamton University, State University of New York

LibreTexts™

Binghamton University, State University of New York

Introduction to the Modeling and Analysis of Complex Systems

Hiroki Sayama

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

This text was compiled on 10/13/2025

TABLE OF CONTENTS

Licensing

1: Introduction to Modeling and Analysis

- 1.1: Complex Systems in a Nutshell
- 1.2: Topical Clusters

2: Fundamentals of Modeling

- 2.1: Models in Science and Engineering
- 2.2: How to Create a Model
- 2.3: Modeling Complex Systems
- 2.4: What Are Good Models?
- 2.5: A Historical Perspective

3: Basics of Dynamical Systems

- 3.1: What are Dynamical Systems?
- 3.2: Phase Space
- 3.3: What Can We Learn?

4: Discrete-Time Models I - Modeling

- 4.1: Discrete-Time Models with Difference Equations
- 4.2: Classifications of Model Equations
- 4.3: Simulating Discrete-Time Models with One Variable
- 4.4: Simulating Discrete-Time Models with Multiple Variables
- 4.5: Building Your Own Model Equation
- 4.6: Building Your Own Model Equations with Multiple Variables

5: Discrete-Time Models II - Analysis

- 5.1: Finding Equilibrium Points
- 5.2: 5.2 Phase Space Visualization of Continuous-State Discrete-Time Models
- 5.3: 5.3 Cobweb Plots for One-Dimensional Iterative Maps
- 5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model
- 5.5: Variable Rescaling of Discrete-Time Models
- 5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems
- 5.7: 5.7 Linear Stability Analysis of Discrete-Time Nonlinear Dynamical Systems

6: Continuous-Time Models I - Modeling

- 6.1: Continuous-Time Models with Differential Equations
- 6.2: Classifications of Model Equation
- 6.3: Connecting Continuous - Time Models with DiscreteTime Models
- 6.4: Simulating Continuous-Time Models
- 6.5: Building Your Own Model Equation

7: Continuous-Time Models II - Analysis

- 7.1: Finding Equilibrium Points
- 7.2: Phase Space Visualization
- 7.3: Variable Rescaling of Continuous-Time Models
- 7.4: Asymptotic Behavior of Continuous-Time Linear Dynamical Systems
- 7.5: Linear Stability Analysis of Nonlinear Dynamical Systems

8: Bifurcations

- 8.1: 8.1 What are Bifurcations?
- 8.2: Bifurcations in 1-D Continuous-Time Models
- 8.3: Hopf Bifurcations in 2-D Continuous-Time Models
- 8.4: Bifurcations in Discrete-Time Models

9: Chaos

- 9.1: Chaos in Discrete-Time Models
- 9.2: Characteristics of Chaos
- 9.3: Lyapunov Exponent
- 9.4: Chaos in Continuous-Time Model

10: Interactive Simulations of Complex Systems

- 10.1: Simulation of Systems with a Large Number of Variables
- 10.2: Interactive Simulation with PyCX
- 10.3: Interactive Parameter Control in PyCX
- 10.4: Simulation without PyCX

11: Cellular Automata I - Modeling

- 11.1: Definition of Cellular Automata
- 11.2: Examples of Simple Binary Cellular Automata Rules
- 11.3: Simulating Cellular Automata
- 11.4: Extensions of Cellular Automata
- 11.5: Examples of Biological Cellular Automata Models

12: Cellular Automata II - Analysis

- 12.1: Sizes of Rule Space and Phase Space
- 12.2: Phase Space Visualization
- 12.3: Mean-Field Approximation
- 12.4: Renormalization Group Analysis to Predict Percolation Thresholds

13: Continuous Field Models I - Modeling

- 13.1: Continuous Field Models with Partial Differential Equations
- 13.2: Fundamentals of Vector Calculus
- 13.3: Visualizing Two-Dimensional Scalar and Vector Field
- 13.4: Modeling Spatial Movement
- 13.5: Simulation of Continuous Field Models
- 13.6: Reaction-Diffusion Systems

14: Continuous Field Models II - Analysis

- 14.1: Finding Equilibrium States
- 14.2: Variable Rescaling
- 14.3: Linear Stability Analysis of Continuous Field Models
- 14.4: Linear Stability Analysis of Reaction-Diffusion Systems
- Index

15: Basics of Networks

- 15.1: Network Models
- 15.2: Terminologies of Graph Theory
- 15.3: Constructing Network Models with NetworkX
- 15.4: Visualizing Networks with NetworkX
- 15.5: Importing/Exporting Network Data
- 15.6: Generating Random Graphs

16: Dynamical Networks I - Modeling

- 16.1: Dynamical Network Models
- 16.2: Simulating Dynamics on Networks
- 16.3: Simulating Dynamics of Networks
- 16.4: Simulating Adaptive Networks

17: Dynamical Networks II - Analysis of Network Topologies

- 17.1: Network Size, Density, and Percolation
- 17.2: Shortest Path Length
- 17.3: Centralities and Coreness
- 17.4: Clustering
- 17.5: Degree Distribution
- 17.6: Assortativity
- 17.7: Community Structure and Modularity

18: Dynamical Networks III - Analysis of Network Dynamics

- 18.1: Dynamics of Continuous-State Networks
- 18.2: Diffusion on Networks
- 18.3: Synchronizability
- 18.4: Mean-Field Approximation of Discrete-State Networks
- 18.5: Mean-Field Approximation on Random Networks
- 18.6: Mean-Field Approximation on Scale-Free Networks

19: Agent-Based Models

- 19.1: What Are Agent-Based Models?
- 19.2: Building an Agent-Based Model
- 19.3: Agent-Environment Interaction
- 19.4: Ecological and Evolutionary Models

Index

Glossary

Detailed Licensing

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Introduction to Modeling and Analysis

1.1: Complex Systems in a Nutshell
1.2: Topical Clusters

This page titled 1: Introduction to Modeling and Analysis is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

1.1: Complex Systems in a Nutshell

It may be rather unusual to begin a textbook with an outright definition of a topic, but anyway, here is what we mean by complex systems in this textbook[1]:

Definition: Complex Systems

Complex systems are networks made of a number of components that interact with each other, typically in a nonlinear fashion. Complex systems may arise and evolve through self-organization, such that they are neither completely regular nor completely random, permitting the development of emergent behavior at macroscopic scales.

These properties can be found in many real-world systems, e.g., gene regulatory networks within a cell, physiological systems of an organism, brains and other neural systems, food webs, the global climate, stock markets, the Internet, social media, national and international economies, and even human cultures and civilizations. To better understand what complex systems are, it might help to know what they are not. One example of systems that are not complex is a collection of independent components, such as an ideal gas (as discussed in thermodynamics) and random coin tosses (as discussed in probability theory). This class of systems was called "*problems of disorganized complexity*" by American mathematician and systems scientist Warren Weaver [2]. Conventional statistics works perfectly when handling such independent entities. Another example, which is at the other extreme, is a collection of strongly coupled components, such as rigid bodies (as discussed in classical mechanics) and fixed coin tosses (I'm not sure which discipline studies this). Weaver called this class of systems "*problems of simplicity*" [2]. In this class, the components of a system are tightly coupled to each other with only a few or no degrees of freedom left within the system, so one can describe the collection as a single entity with a small number of variables. There are very well-developed theories and tools available to handle either case. Unfortunately, however, most real-world systems are somewhere in between.

Complex systems science is a rapidly growing scientific research area that fills the huge gap between the two traditional views that consider systems made of either completely independent or completely coupled components. This is the gap where what Weaver called "*problems of organized complexity*" exist [2]. Complex systems science develops conceptual, mathematical, and computational tools to describe systems made of interdependent components. It studies the structural and dynamical properties of various systems to obtain general, cross-disciplinary implications and applications.

Complex systems science has multiple historical roots and topical clusters of concepts, as illustrated in Fig. 1.1.1. There are two core concepts that go across almost all subareas of complex systems: emergence and self-organization. The idea of emergence was originally discussed in philosophy more than a century ago. There are many natural phenomena where some property of a system observed at macroscopic scales simply can't be reduced to microscopic physical rules that drive the system's behavior. For example, you can easily tell that a dog wagging its tail is alive, but it is extremely difficult to explain what kind of microscopic physical/chemical processes going on in its body are making this organism "alive." Another typical example is your consciousness. You know you are conscious, but it is hard to describe what kind of neurophysiological processes make you a "conscious" entity. Those macroscopic properties (livingness, consciousness) are called emergent properties of the systems.

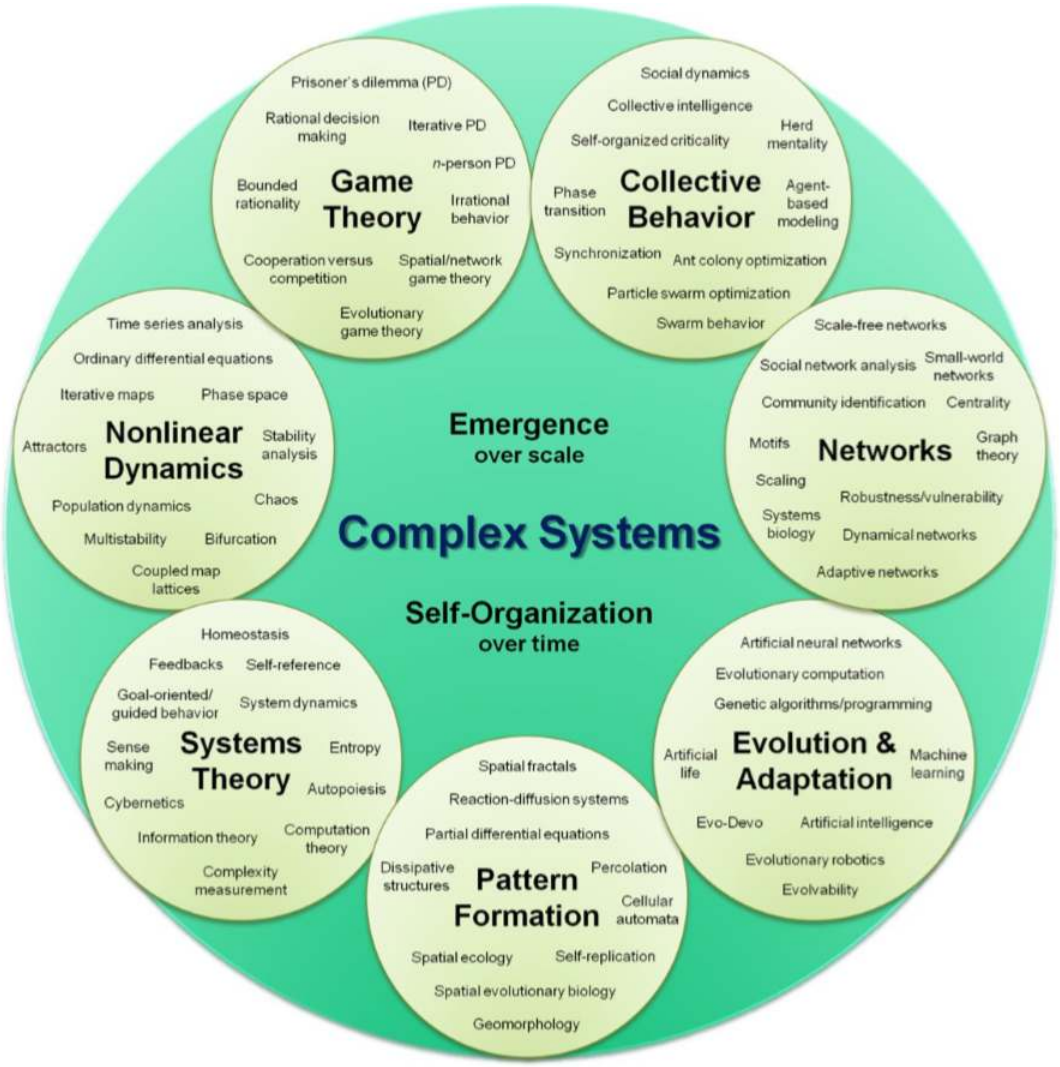


Figure **1.1.1**: *Visual, organizational map of complex systems science broken into seven topical areas. The three circles on the left (Nonlinear Dynamics, Systems Theory, and Game Theory) are the historical roots of complex systems science, while the other four circles (Pattern Formation, Evolution and Adaptation, Networks, and Collective Behavior) are the more recently studied topical areas. Created by the author and available from Wikipedia [3].*

Despite its long history of discussion and debate, there are still a number of different definitions for the concept of emergence in complex systems science. However, the one thing that is common in most of the proposed definitions is that the emergence is about the system's properties at different *scales*. If you observe a property at a macroscopic scale that is fundamentally different from what you would naturally expect from microscopic rules, then you are witnessing emergence. More concisely, emergence is a nontrivial relationship between the system's properties at different scales. This definition was proposed by complex systems scientist Yaneer Bar-Yam[4]. I will adopt this definition since it is simple and consistent with most of the definitions proposed in the literature.

Definition: Emergence

Emergence is a nontrivial relationship between the properties of a system at microscopic and macroscopic scales. Macroscopic properties are called emergent when it is hard to explain them simply from microscopic properties.

Another key idea of complex systems science is *self-organization*, which is sometimes confused with emergence. Some researchers even use these terms almost interchangeably. One clear distinction, though, is that, while emergence is about scale, self-organization is about time (in addition to scale). Namely, you call something *self-organizing* when you observe that the system spontaneously organizes itself to produce a nontrivial macroscopic structure and/or behavior (or "order," if you will) as time

progresses. In other words, self-organization is a dynamical process that looks as if it were going against the second law of thermodynamics (which states that entropy of a closed system increases monotonically over time). Many physical, biological, and social systems show self-organizing behavior, which could appear mysterious when people were not aware of the possibility of self-organization. Of course, these systems are not truly going against the law of thermodynamics, because they are open systems that are driven by energy flow coming from and going to the outside of the system. In a sense, the idea of self-organization gives a dynamical explanation for emergent properties of complex systems.

Definition: Self-organization

Self-organization is a dynamical process by which a system spontaneously forms nontrivial macroscopic structures and/or behaviors over time.

Around these two key ideas, there are several topical clusters, which are illustrated in Fig. 1.1.1. Let's quickly review them.

[1] In fact, the first sentence of this definition is just a bit wordier version of Herbert Simon's famous definition in his 1962 paper [1]: "[A] complex system [is a system] made up of a large number of parts that interact in a non-simple way."

This page titled 1.1: Complex Systems in a Nutshell is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

1.2: Topical Clusters

Nonlinear dynamics is probably the topical cluster that has the longest history, at least from as far back as the 17th century when Isaac Newton and Gottfried Wilhelm Leibniz invented calculus and differential equations. But it was found only in the 20th century that systems that include nonlinearity in their dynamics could show some weird behaviors, such as *chaos* [5, 6] (which will be discussed later). Here, *nonlinearity* means that the outputs of a system are not given by a linear combination of the inputs. In the context of system behavior, the inputs and outputs can be the current and next states of the system, and if their relationship is not linear, the system is called a *nonlinear system*. The possibility of chaotic behavior in such nonlinear systems implies that there will be no analytical solutions generally available for them. This constitutes one of the several origins of the idea of complexity.

Systems theory is another important root of complex systems science. It rapidly developed during and after World War II, when there was a huge demand for mathematical theories to formulate systems that could perform computation, control, and/or communication. This category includes several ground-breaking accomplishments in the last century, such as Alan Turing's foundational work on theoretical computer science [7], Norbert Wiener's cybernetics [8], and Claude Shannon's information and communication theories [9]. A common feature shared by those theories is that they all originated from some engineering discipline, where engineers were facing real-world complex problems and had to come up with tools to meet societal demands. Many innovative ideas of systems thinking were invented in this field, which still form the key components of today's complex systems science.

Game theory also has an interesting societal background. It is a mathematical theory, established by John von Neumann and Oskar Morgenstern [10], which formulates the decisions and behaviors of people playing games with each other. It was developed during the Cold War, when there was a need to seek a balance between the two mega powers that dominated the world at that time. The rationality of the game players was typically assumed in many game theory models, which made it possible to formulate the decision making process as a kind of deterministic dynamical system (in which either decisions themselves or their probabilities could be modeled deterministically). In this sense, game theory is linked to nonlinear dynamics. One of the many contributions game theory has made to science in general is that it demonstrated ways to model and analyze human behavior with great rigor, which has made huge influences on economics, political science, psychology, and other areas of social sciences, as well as contributing to ecology and evolutionary biology.

Later in the 20th century, it became clearly recognized that various innovative ideas and tools arising in those research areas were all developed to understand the behavior of systems made of multiple interactive components whose macroscopic behaviors were often hard to predict from the microscopic rules or laws that govern their dynamics. In the 1980s, those systems began to be the subject of widespread interdisciplinary discussions under the unified moniker of "complex systems." The research area of complex systems science is therefore inherently interdisciplinary, which has remained unchanged since the inception of the field. The recent developments of complex systems research may be roughly categorized into four topical clusters: pattern formation, evolution and adaptation, networks, and collective behavior.

Pattern formation is a self-organizing process that involves space as well as time. A system is made of a large number of components that are distributed over a spatial domain, and their interactions (typically local ones) create an interesting spatial pattern over time. *Cellular automata*, developed by John von Neumann and Stanisław Ulam in the 1940s [11], are a well-known example of mathematical models that address pattern formation. Another modeling framework is *partial differential equations* (PDEs) that describe spatial changes of functions in addition to their temporal changes. We will discuss these modeling frameworks later in this textbook.

Evolution and adaptation have been discussed in several different contexts. One context is obviously evolutionary biology, which can be traced back to Charles Darwin's evolutionary theory. But another, which is often discussed more closely to complex systems, is developed in the "*complex adaptive systems*" context, which involves *evolutionary computation*, *artificial neural networks*, and other frameworks of man-made adaptive systems that are inspired by biological and neurological processes. Called *soft computing*, *machine learning*, or *computational intelligence*, nowadays, these frameworks began their rapid development in the 1980s, at the same time when complex systems science was about to arise, and thus they were strongly coupled—conceptually as well as in the literature. In complex systems science, evolution and adaptation are often considered to be general mechanisms that can not only explain biological processes, but also create non-biological processes that have dynamic learning and creative abilities. This goes well beyond what a typical biological study covers.

Finally, *networks* and *collective behavior* are probably the most current research fronts of complex systems science (as of 2015). Each has a relatively long history of its own. In particular, the study of networks was long known as *graph theory* in mathematics, which was started by Leonhard Euler back in the 18th century. In the meantime,the recent boom of network and collective behavior research has been largely driven by the availability of increasingly large amounts of data. This is obviously caused by the explosion of the Internet and the WWW, and especially the rise of mobile phones and social media over the last decade. With these information technology infrastructures, researchers are now able to obtain high-resolution, high-throughput data about how people are connected to each other, how they are communicating with each other, how they are moving geographically, what they are interested in, what they buy, how they form opinions or preferences, how they respond to disastrous events, and the list goes on and on. This allows scientists to analyze the structure of networks at multiple scales and also to develop dynamical models of how the collectives behave. Similar data-driven movements are also seen in biology and medicine (e.g., behavioral ecology, systems biology, epidemiology), neuroscience (e.g., the Human Connectome Project [12]), and other areas. It is expected that these topical areas will expand further in the coming decades as the understanding of the collective dynamics of complex systems will increase their relevance in our everyday lives.

Here, I should note that these seven topical clusters are based on my own view of the field, and they are by no means well defined or well accepted by the community. There must be many other ways to categorize diverse complex systems related topics. These clusters are more or less categorized based on research communities and subject areas, while the methodologies of modeling and analysis traverse across many of those clusters. Therefore, the following chapters of this textbook are organized based on the methodologies of modeling and analysis, and they are not based on specific subjects to be modeled or analyzed. In this way, I hope you will be able to learn the "how-to" skills systematically in the most generalizable way, so that you can apply them to various subjects of your own interest.

? Exercise 1.2.1

Choose a few concepts of your own interest from Fig. 1.1.1. Do a quick online literature search for those words, using Google Scholar (http://scholar.google.com/), arXiv (http://arxiv.org/), etc., to find out more about their meaning, when and how frequently they are used in the literature, and in what context.

? Exercise 1.2.2

Conduct an online search to find visual examples or illustrations of some of the concepts shown in Fig. 1.1.1. Discuss which example(s) and/or illustration(s) are most effective in conveying the key idea of each concept. Then create a short presentation of complex systems science using the visual materials you selected.

? Exercise 1.2.3

Think of other ways to organize the concepts shown in Fig.1.1.1 (and any other relevant concepts you want to include). Then create your own version of a map of complex systems science.

Now we are ready to move on. Let's begin our journey of complex systems modeling and analysis.

This page titled 1.2: Topical Clusters is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

2: Fundamentals of Modeling

2.1: Models in Science and Engineering

2.2: How to Create a Model

2.3: Modeling Complex Systems

2.4: What Are Good Models?

2.5: A Historical Perspective

This page titled 2: Fundamentals of Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

2.1: Models in Science and Engineering

Science is an endeavor to try to understand the world around us by discovering fundamental laws that describe how it works. Such laws include Newton's law of motion, the ideal gas law, Ohm's law in electrical circuits, the conservation law of energy, and so on, some of which you may have learned already. A typical cycle of scientific effort by which scientists discover these fundamental laws may look something like this:

1. Observe nature.
2. Develop a hypothesis that could explain your observations.
3. From your hypothesis, make some predictions that are testable through an experiment.
4. Carry out the experiment to see if your predictions are actually true.

    • Yes → Your hypothesis is proven, congratulations. Uncork a champagne bottle and publish a paper.

    • No → Your hypothesis was wrong, unfortunately. Go back to the labor the field, get more data, and develop another hypothesis.

Many people think this is how science works. But there is at least one thing that is not quite right in the list above. What is it? Can you figure it out?

As some of you may know already, the problem exists in the last part, i.e., when the experiment produced a result that matched your predictions. Let's do some logic to better understand what the problem really is. Assume that you observed a phenomenon P in nature and came up with a hypothesis H that can explain P. This means that a logical statement $H \rightarrow P$ is always true (because you chose H that way). To prove H, you also derived a prediction Q from H , i.e., another logical statement $H \rightarrow Q$ is always true, too. Then you conduct experiments to see if Q can be actually observed. What if Q is actually observed? Or, what if "*not* Q" is observed instead?

If "*not* Q" is observed, things are easy. Logically speaking, $(H \rightarrow Q)$ is equivalent to (*not* $Q \rightarrow$ *not* H) because they are contrapositions of each other, i.e., logically identical statements that can be converted from one to another by negating both the condition and the consequence and then flipping their order. This means that, if not Q is true, then it logically proves that not H is also true, i.e., your hypothesis is wrong. This argument is clear, and there is no problem with it (aside from the fact that you will probably have to redo your hypothesis building and testing).

The real problem occurs when your experiment gives you the desired result, Q. Logically speaking, "$(H \rightarrow Q)$ and Q" *doesn't tell you anything about whether H is true or not!* There are many ways your hypothesis could be wrong or insufficient even if the predicted outcome was obtained in the experiment. For example, maybe another alternative hypothesis R could be the right one ($R \rightarrow P$, $R \rightarrow Q$), or maybe H would need an additional condition K to predict P and Q (H and $K \rightarrow P$, H and $K \rightarrow Q$) but you were not aware of the existence of K.

Let me give you a concrete example. One morning, you looked outside and found that your lawn was wet (observation P). You hypothesized that it must have rained while you were asleep (hypothesis H), which perfectly explains your observation ($H \rightarrow P$). Then you predicted that, if it rained overnight, the driveway next door must also be wet (prediction Q that satisfies $H \rightarrow Q$). You went out to look and, indeed, it was also wet (if not, H would be clearly wrong). Now, think about whether this new observation really proves your hypothesis that it rained overnight. If you think critically, you should be able to come up with other scenarios in which both your lawn and the driveway next door could be wet without having a rainy night. Maybe the humidity in the air was unusually high, so the condensation in the early morning made the ground wet everywhere. Or maybe a fire hydrant by the street got hit by a car early that morning and it burst open, wetting the nearby area. There could be many other potential explanations for your observation.

In sum, obtaining supportive evidence from experiments doesn't prove your hypothesis in a logical sense. It only means that you have failed to disprove your hypothesis. However, many people still believe that science can prove things in an absolute way. *It can't.* There is no logical way for us to reach the ground truth of nature[1].

This means that all the "laws of nature," including those listed previously, are no more than well-tested hypotheses at best. Scientists have repeatedly failed to disprove them, so we give them more credibility than we do to other hypotheses. But there is

absolutely no guarantee of their universal, permanent correctness. There is always room for other alternative theories to better explain nature.

In this sense, all science can do is just build *models* of nature. All of the laws of nature mentioned earlier are also models, not scientific facts, strictly speaking. This is something every single person working on scientific research should always keep in mind. I have used the word “model” many times already in this book without giving it a definition. So here is an informal definition:

> A *model* is a simplified representation of a system. It can be conceptual, verbal, diagrammatic, physical, or formal (mathematical).

As a cognitive entity interacting with the external world, you are always creating a model of something in your mind. For example, at this very moment as you are reading this textbook, you are probably creating a model of what is written in this book. *Modeling* is a fundamental part of our daily cognition and decision making; it is not limited only to science.

With this understanding of models in mind, we can say that science is an endless effort to create models of nature, because, after all, modeling is the one and only rational approach to the unreachable reality. And similarly, engineering is an endless effort to control or influence nature to make something desirable happen, by creating and controlling its models. Therefore, modeling occupies the most essential part in any endeavor in science and engineering.

? Exercise 2.1.1

In the “wet lawn” scenario discussed above, come up with a few more alternative hypotheses that could explain both the wet lawn and the wet driveway without assuming that it rained. Then think of ways to find out which hypothesis is most likely to be the real cause.

? Exercise 2.1.2

Name a couple of scientific models that are extensively used in today’s scientific/engineering fields. Then investigate the following:

- How were they developed?
- What made them more useful than earlier models?
- How could they possibly be wrong?

[1]This fact is deeply related to the impossibility of general system identification, including the identification of computational processes.

This page titled 2.1: Models in Science and Engineering is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

2.2: How to Create a Model

There are a number of approaches for scientific model building. My favorite way of classifying various kinds of modeling approaches is to put them into the following two major families:

> **Descriptive modeling** In this approach, researchers try to specify the actual state of a system at a given time point (or at multiple time points) in a descriptive manner. Taking a picture, creating a miniature (this is literally a "model" in the usual sense of the word), and writing a biography of someone, all belong to this family of modeling effort. This can also be done using quantitative methods (e.g., equations, statistics, computational algorithms), such as regression analysis and pattern recognition. They all try to capture "what the system looks like."
>
> **Rule-based modeling** In this approach, researchers try to come up with dynamical rules that can explain the observed behavior of a system. This allows researchers to make predictions of its possible (e.g., future) states. Dynamical equations, theories, and first principles, which describe how the system will change and evolve over time, all belong to this family of modeling effort. This is usually done using quantitative methods, but it can also be achieved at conceptual levels as well (e.g., Charles Darwin's evolutionary theory). They all try to capture "how the system will behave."

Both modeling approaches are equally important in science and engineering. For example, observation of planetary movement using telescopes in the early 17th century generated a lot of descriptive information about how they actually moved. This information was already a model of nature because it constituted a simplified representation of reality.

In the meantime, Newton derived the law of motion to make sense out of observational information, which was a rule-based modeling approach that allowed people to make predictions about how the planets would/could move in the future or in a hypothetical scenario. In other words, descriptive modeling is a process in which descriptions of a system are produced and accumulated, while rule-based modeling is a process in which underlying dynamical explanations are built for those descriptions. These two approaches take turns and form a single cycle of the scientific modeling effort.

In this textbook, we will focus on the latter, the rule-based modeling approach. This is because rule-based modeling plays a particularly important role in complex systems science. More specifically, developing a rule-based model at microscopic scales and studying its macroscopic behaviors through computer simulation and/or mathematical analysis is almost a necessity to understand emergence and self-organization of complex systems. We will discuss how to develop rule-based models and what the challenges are throughout this textbook.

A typical cycle of rule-based modeling effort goes through the following steps (which are similar to the cycle of scientific discoveries we discussed above):

1. Observe the system of your interest.
2. Reflect on the possible rules that might cause the system's characteristics that were seen in the observation.
3. Derive predictions from those rules and compare them with reality.
4. Repeat the above steps to modify the rules until you are satisfied with the model (or you run out of time or funding).

This seems okay, and it doesn't contain the logical problem of "proving a hypothesis" that we had before, because I loosened the termination criteria to be your satisfaction as a researcher. However, there is still one particular step that is fundamentally difficult. Which step do you think it is?

Of course, each of the four steps has its own unique challenges, but as an educator who has been teaching complex systems modeling for many years, I find that the second step ("*Reflect on possible rules that might cause the system's characteristics seen in the observation*.") is particularly challenging to modelers. This is because this step is so deeply interwoven with the modeler's knowledge, experience, and everyday cognitive processes. It is based on who you are, what you know, and how you see the world—it is, ultimately, a personal thinking process, which is very difficult to teach or to learn in a structured way.

Let me give you some examples to illustrate my point. The following figure shows an observation of a system over time. Can you create a mathematical model of this observation?

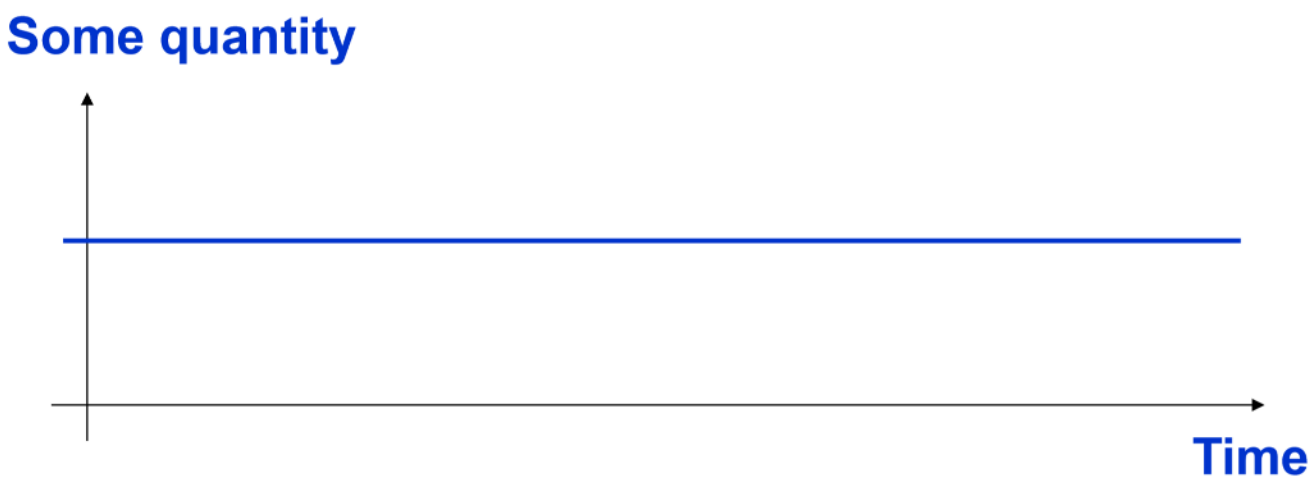


Figure **2.2.1**: *Observation example 1.*

This one should be quite easy, because the observed data show that nothing changed over time. The description "no change" is already a valid model written in English, but if you prefer writing it in mathematical form, you may want to write it as

$$x(t) = C \tag{2.2.1}$$

or, if you use a differential equation,

$$dx/dt = 0. \tag{2.2.2}$$

Coming up with these models is a no brainer, because we have seen this kind of behavior many times in our daily lives. Here is another example. Can you create a mathematical model of this observation?

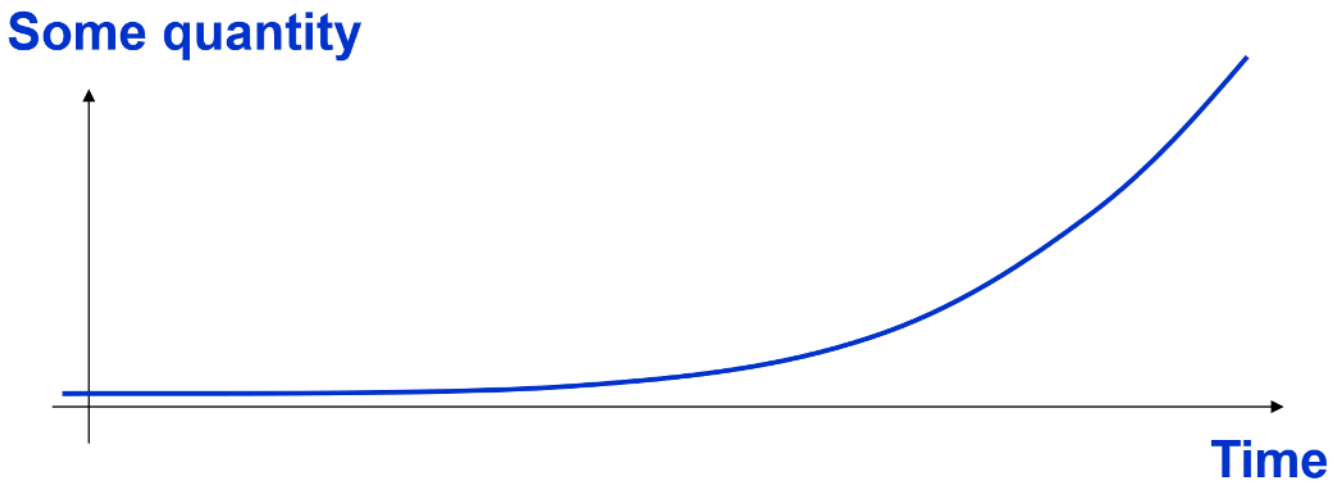


Figure **2.2.2**: *Observation example 2.*

Now we see some changes. It seems the quantity monotonically increased over time. Then your brain must be searching your past memories for a pattern that looks like this curve you are looking at, and you may already have come up with a phrase like "*exponential growth*," or more mathematically, equations like

$$x(t) = ae^{bt} \tag{2.2.3}$$

or

$$dx/dt = bx \tag{2.2.4}$$

This may be easy or hard, depending on how much knowledge you have about such exponential growth models. In the meantime, if you show the same curve to middle school students, they may proudly say that this must be the right half of a flattened parabola that they just learned about last week. And there is nothing fundamentally wrong with that idea. It could be a right half of a parabola, indeed. We never know for sure until we see what the entire curve looks like for $-\infty < t < \infty$.

Let's move on to a more difficult example. Create a mathematical model of the following observation.

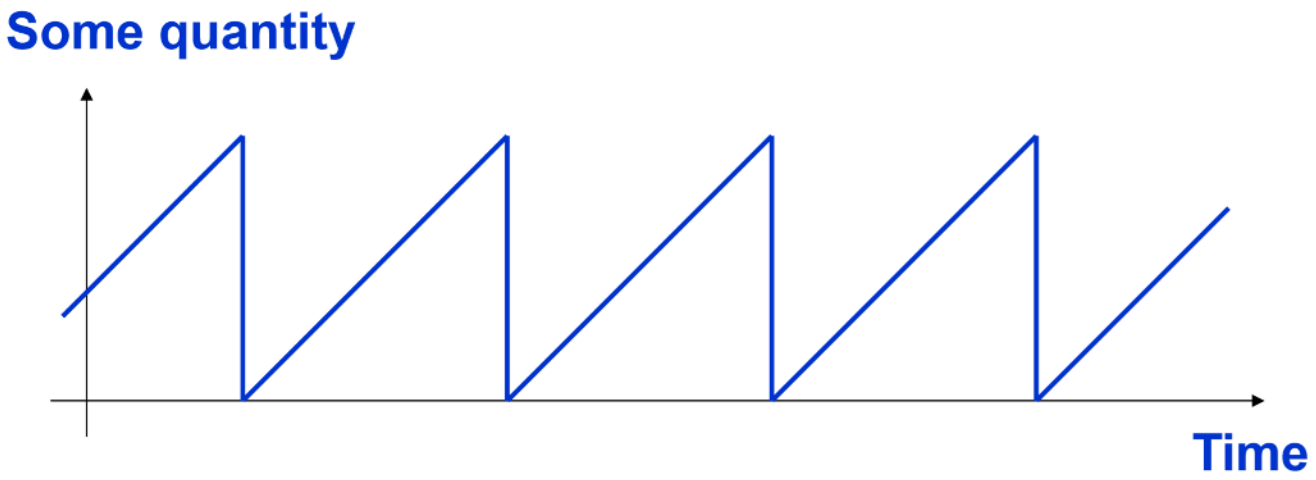


Figure **2.2.3**: *Observation example 3.*

It is getting harder. Unless you have seen this kind of dynamic behavior before, you will have a hard time coming up with any concise explanation of it. An engineer might say, "This is a sawtooth wave," or a computer scientist might say,"This is time mod something." Or, an even more brilliant answer could come from an elementary school kid, saying,"This must be months in a calendar!" (which is equivalent to what the computer scientist said, by the way). In either case, people tend to map a new observation to something they already know in their mind when they create a model.

This last example is the toughest. Create a mathematical model of the following observation.

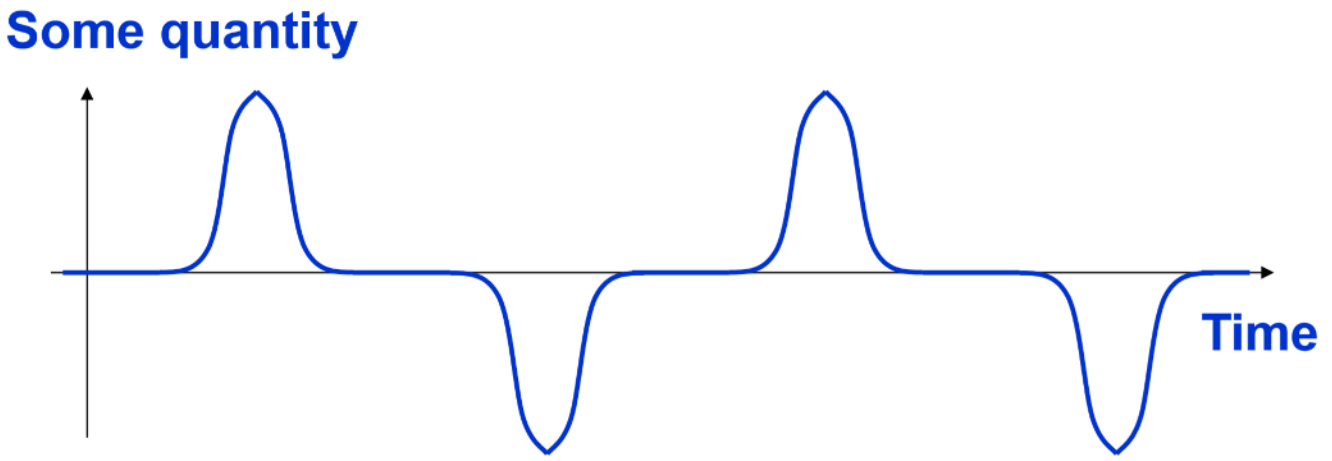


Figure **2.2.4**: *Observation example 4.*

Did you come up with any ideas? I have seen only a few people who were able to make reasonable models of this observation throughout my career. The reason why this example is so hard to model is because we don't see this kind of behavior often in our lives. We are simply not experienced with it. We don't have a good mental template to use to capture the essence of this pattern[2].

I hope that these examples have made my point clear by now. Coming up with a model is inherently a personal process, which depends on your own knowledge, experience, and worldview. There is no single algorithm or procedure you can follow to develop

a good model. The modeling process is a full-scale interaction between the external world and your whole, intellectual self. To become a good modeler, you will need to gain diverse knowledge and experience and develop rich worldviews. This is why I said it would be very difficult to be taught.

? Exercise 2.2.1

Create a few different models for each of the examples shown above. Discuss how those models differ from each other, and what you should do to determine which model is more appropriate as an explanation of the observed behavior.

[2]For those who are curious—this kind of curve could be generated by raising a sine or cosine function of time to an odd number (e.g., $sin3(t)$, $cos5(t)$), but I am not sure if knowing this would ever help you in your future career.

This page titled 2.2: How to Create a Model is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

2.3: Modeling Complex Systems

The challenge in developing a model becomes particularly tough when it comes to the modeling of complex systems, because their unique properties (networks, nonlinearity, emergence, self-organization, etc.) are not what we are familiar with. We usually think about things on a single scale in a step-by-step, linear chain of reasoning, in which causes and effects are clearly distinguished and discussed sequentially. But this approach is not suitable for understanding complex systems where a massive amount of components are interacting with each other interdependently to generate patterns over a broad range of scales. Therefore, the behavior of complex systems often appears to contradict our everyday experiences.

As illustrated in the examples above, it is extremely difficult for us to come up with a reasonable model when we are facing something unfamiliar. And it is even more difficult to come up with a reasonable set of microscopic rules that could explain the observed macroscopic properties of a system. Most of us are simply not experienced enough to make logical connections between things at multiple different scales.

How can we improve our abilities to model complex systems? The answer might be as simple as this: We need to become experienced and familiar with various dynamics of complex systems to become a good modeler of them. How can we become experienced? This is a tricky question, but thanks to the availability of the computers around us, *computational modeling* and *simulation* is becoming a reasonable, practical method for this purpose. You can construct your own model with full details of microscopic rules coded into your computer, and then let it actually show the macroscopic behavior arising from those rules. Such computational modeling and simulation is a very powerful tool that allows you to gain interactive, intuitive (simulated) experiences of various possible dynamics that help you make mental connections between micro- and macroscopic scales. I would say there are virtually no better tools available for studying the dynamics of complex systems in general.

There are a number of pre-built tools available for complex systems modeling and simulation, including NetLogo [13], Repast [14], MASON [15], Golly [16], and so on. You could also build your own model by using general-purpose computer programming languages, including C, C++, Java, Python, R, Mathematica, MATLAB, etc. In this textbook, we choose Python as our modeling tool, specifically Python 2.7, and use *PyCX* [17] to build interactive dynamic simulation models[3]. Python is free and widely used in scientific computing as well as in the information technology industries. More details of the rationale for this choice can be found in [17].

When you create a model of a complex system, you typically need to think about the following:

1. *What are the key questions you want to address?*

2. *To answer those key questions, at what scale should you describe the behaviors of the system's components?* These components will be the "microscopic" components of the system, and you will define dynamical rules for their behaviors.

3. *How is the system structured?* This includes what those microscopic components are, and how they will be interacting with each other.

4. *What are the possible states of the system?* This means describing what kind of dynamical states each component can take.

5. *How does the state of the system change over time?* This includes defining the dynamical rules by which the components' states will change over time via their mutual interaction, as well as defining how the interactions among the components will change over time.

Figuring out the "right" choices for these questions is by no means a trivial task. You will likely need to loop through these questions several times until your model successfully produces behaviors that mimic key aspects of the system you are trying to model. We will practice many examples of these steps throughout this textbook.

? Exercise 2.3.1

Create a schematic model of some real-world system of your choice that is made of many interacting components. Which scale do you choose to describe the microscopic components? What are those components? What states can they take? How are those components connected? How do their states change over time? After answering all of these questions, make a mental prediction about what kind of macroscopic behaviors would arise if you ran a computational simulation of your model

[3]For those who are new to Python programming, see Python's online tutorial at docs.python. org/2/tutorial/index.html. Several pre-packaged Python distributions are available for free, such as Anaconda (available from continuum.io/downloads) and Enthought Canopy (available from enthought.com/products/canopy/). A recommended environment is Anaconda's Python code editor named "Spyder."

This page titled 2.3: Modeling Complex Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

2.4: What Are Good Models?

You can create various kinds of models for a system, but useful ones have several important properties. Here is a very brief summary of what a good model should look like:

> A good model is simple, valid, and robust.

Simplicity of a model is really the key essence of what modeling is all about. The main reason why we want to build a model is that we want to have a shorter, simpler description of reality. As the famous principle of *Occam's razor* says, if you have two models with equal predictive power, you should choose the simpler one. This is not a theorem or any logically proven fact, but it is a commonly accepted practice in science. Parsimony is good because it is economical (e.g., we can store more models within the limited capacity of our brain if they are simpler) and also insightful (e.g., we may find useful patterns or applications in the models if they are simple). If you can eliminate any parameters, variables, or assumptions from your model without losing its characteristic behavior, you should.

V*alidity* of a model is how closely the model's prediction agrees with the observed reality. This is of utmost importance from a practical viewpoint. If your model's prediction doesn't reasonably match the observation, the model is not representing reality and is probably useless. It is also very important to check the validity of not only the predictions of the model but also the assumptions it uses, i.e., whether each of the assumptions used in your model makes sense *at its face value*, in view of the existing knowledge as well as our common sense. Sometimes this "*face validity*" is more important in complex systems modeling, because there are many situations where we simply can't conduct a quantitative comparison between the model prediction and the observational data. Even if this is the case, you should at least check the face validity of your model assumptions based on your understanding about the system and/or the phenomena.

Note that there is often a trade-off between trying to achieve simplicity and validity of a model. If you increase the model complexity, you may be able to achieve a better fit to the observed data, but the model's simplicity is lost and you also have the risk of *overfitting*— that is, the model prediction may become adjusted too closely to a specific observation at the cost of generalizability to other cases. You need to strike the right balance between those two criteria.

Finally, *robustness* of a model is how insensitive the model's prediction is to minor variations of model assumptions and/or parameter settings. This is important because there are always errors when we create assumptions about,or measure parameter values from,the real world. If the prediction made by your model is sensitive to their minor variations, then the conclusion derived from it is probably not reliable. But if your model is robust, the conclusion will hold under minor variations of model assumptions and parameters, therefore it will more likely apply to reality, and we can put more trust in it.

? Exercise 2.4.1

Humanity has created a number of models of the solar system in its history. Some of them are summarized below:

- Ptolemy's geocentric model (which assumes that the Sun and other planets are revolving around the Earth)
- Copernicus 'heliocentric model (which assumes that the Earth and other planets are revolving around the Sun in concentric circular orbits)
- Kepler's laws of planetary motion (which assumes that the Earth and other planets are revolving in elliptic orbits, at one of whose foci is the Sun, and that the area swept by a line connecting a planet and the Sun during a unit time period is always the same)
- Newton's law of gravity (which assumes that a gravitational force between two objects is proportional to their masses and inversely proportional to their distance squared)

Investigate these models, and compare them in terms of simplicity, validity and robustness.

This page titled 2.4: What Are Good Models? is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

https://math.libretexts.org/@go/page/7775

2.5: A Historical Perspective

As the final section in this chapter, I would like to present some historical perspective of how people have been developing modeling methodologies over time, especially those for complex systems (Fig. 2.5.1). Humans have been creating descriptive models (diagrams, pictures, physical models, texts, etc.) and some conceptual rule-based models since ancient times. More quantitative modeling approaches arose as more advanced mathematical tools became available. In the descriptive modeling family, descriptive statistics is among such quantitative modeling approaches. In the rule-based modeling family, dynamical equations (e.g., differential equations, difference equations) began to be used to quantitatively formulate theories that had remained at conceptual levels before.

During the second half of the 20th century, computational tools became available to researchers, which opened up a whole new area of *computational modeling* approaches

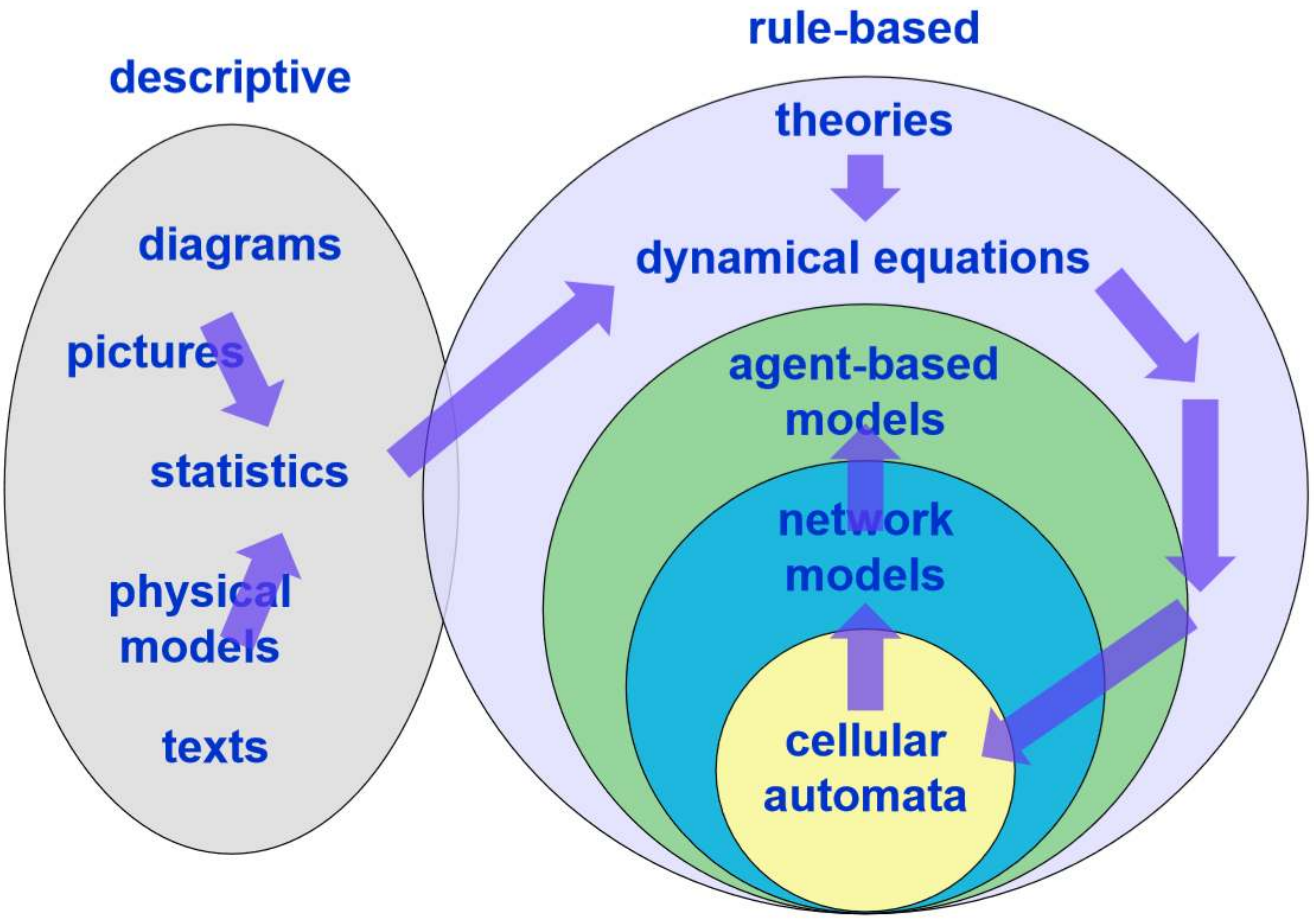


Figure **2.5.1**: *Schematic illustration of how modeling methodologies have developed historically.*

for complex systems modeling. The first of this kind was *cellular automata*, a massive number of identical finite-state machines that are arranged in a regular grid structure and update their states dynamically according to their own and their neighbors' states. Cellular automata were developed by John von Neumann and Stanisław Ulam in the 1940s, initially as a theoretical medium to implement self-reproducing machines [11], but later they became a very popular modeling framework for simulating various interesting emergent behaviors and also for more serious scientific modeling of spatio-temporal dynamics [18]. Cellular automata are a special case of dynamical networks whose topologies are limited to regular grids and whose nodes are usually assumed to be homogeneous and identical.

Dynamical networks formed the next wave of complex systems modeling in the 1970s and 1980s. Their inspiration came from *artificial neural network* research by Warren McCulloch and Walter Pitts[19] as well as by John Hopfield [20,21], and also from

theoretical *gene regulatory network* research by Stuart Kauffman [22]. In this modeling framework, the topologies of systems are no longer constrained to regular grids, and the components and their connections can be heterogeneous with different rules and weights. Therefore, dynamical networks include cellular automata as a special case within them. Dynamical networks have recently merged with another thread of research on topological analysis that originated in graph theory, statistical physics, social sciences, and computational science, to form a new interdisciplinary field of network science [23, 24, 25].

Finally, further generalization was achieved by removing the requirement of explicit network topologies from the models, which is now called *agent-based modeling* (ABM). In ABM, the only requirement is that the system is made of multiple discrete "agents" that interact with each other (and possibly with the environment), whether they are structured into a network or not. Therefore ABM includes network models and cellular automata as its special cases. The use of ABM became gradually popular during the 1980s, 1990s, and 2000s. One of the primary driving forces for it was the application of complex systems modeling to ecological, social, economic, and political processes, in fields like game theory and microeconomics. The surge of *genetic algorithms* and other population-based search/optimization algorithms in computer science also took place at about the same time, which also had synergistic effects on the rise of ABM.

I must be clear that the historical overview presented above is my own personal view, and it hasn't been rigorously evaluated or validated by any science historians (therefore this may not be a valid model!). But I hope that this perspective is useful in putting various modeling frameworks into a unified, chronological picture. The following chapters of this textbook roughly follow the historical path of the models illustrated in this perspective.

? Exercise 2.5.1

Do a quick online literature search to find a few scientific articles that develop or use mathematical/computational models. Read the articles to learn more about their models, and map them to the appropriate locations in Fig. 2.5.

This page titled 2.5: A Historical Perspective is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

3: Basics of Dynamical Systems

A *dynamical system* is a system whose state is uniquely specified by a set of variables and whose behavior is described by predefined rules.

3.1: What are Dynamical Systems?

3.2: Phase Space

3.3: What Can We Learn?

This page titled 3: Basics of Dynamical Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

3.1: What are Dynamical Systems?

Dynamical systems theory is the very foundation of almost any kind of rule-based models of complex systems. It consider show systems change over time, not just static properties of observations. A dynamical system can be informally defined as follows[1]:

Definition: Dynamical System

A *dynamical system* is a system whose state is uniquely specified by a set of variables and whose behavior is described by predefined rules.

Examples of dynamical systems include population growth, a swinging pendulum, the motions of celestial bodies, and the behavior of "rational" individuals playing a negotiation game, to name a few. The first three examples sound legitimate, as those are systems that typically appear in physics textbooks. But what about the last example? Could human behavior be modeled as a deterministic dynamical system? The answer depends on how you formulate the model using relevant assumptions. If you assume that individuals make decisions always perfectly rationally, then the decision making process becomes deterministic, and therefore the interactions among them may be modeled as a deterministic dynamical system. Of course, this doesn't guarantee whether it is a good model or not; the assumption has to be critically evaluated based on the criteria discussed in the previous chapter.

Anyway, dynamical systems can be described over either discrete time steps or a continuous time line. Their general mathematical formulations are as follows:

Definition: Discrete-time dynamical system

$$x_t = F(x_{t-1}, t) \tag{3.1.1}$$

This type of model is called a difference equation, a *recurrence equation*, or an *iterative map* (if there is no t on the right hand side).

Definition: Continuous-time dynamical system

$$\frac{dx}{dt} = F(x, t) \tag{3.1.2}$$

This type of model is called a differential equation.

In either case, x_t or x is the state variable of the system at time t, which may take a scalar or vector value. F is a function that determines the rules by which the system changes its state over time. The formulas given above are first-order versions of dynamical systems (i.e., the equations don't involve x_{t-2}, x_{t-3}, ..., or d^2x/dt^2, d^3x/dt^3, ...). But these first-order forms are general enough to cover all sorts of dynamics that are possible in dynamical systems, as we will discuss later.

Exercise 3.1.1

Have you learned of any models in the natural or social sciences that are formulated as either discrete-time or continuous-time dynamical systems as shown above? If so, what are they? What are the assumptions behind those models?

Exercise 3.1.2

What are some appropriate choices for state variables in the following systems?

- population growth
- swinging pendulum
- motions of celestial bodies
- behavior of "rational" individuals playing a negotiation game

[1]A traditional definition of dynamical systems considers deterministic systems only, but stochastic (i.e., probabilistic) behaviors can also be modeled in a dynamical system by, for example, representing the probability distribution of the system's states as a meta-level state.

This page titled 3.1: What are Dynamical Systems? is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

3.2: Phase Space

Behaviors of a dynamical system can be studied by using the concept of a *phase space*, which is informally defined as follows:

Definition: Phase Space

A phase space of a dynamical system is a theoretical space where every state of the system is mapped to a unique spatial location.

The number of state variables needed to uniquely specify the system's state is called the *degrees of freedom* in the system. You can build a phase space of a system by having an axis for each degree of freedom, i.e., by taking each state variable as one of the orthogonal axes. Therefore, the degrees of freedom of a system equal the dimensions of its phase space. For example, describing the behavior of a ball thrown upward in a frictionless vertical tube can be specified with two scalar variables, the ball's position and velocity, at least until it hits the bottom again. You can thus create its phase space in two dimensions, as shown in Fig. 3.2.1.

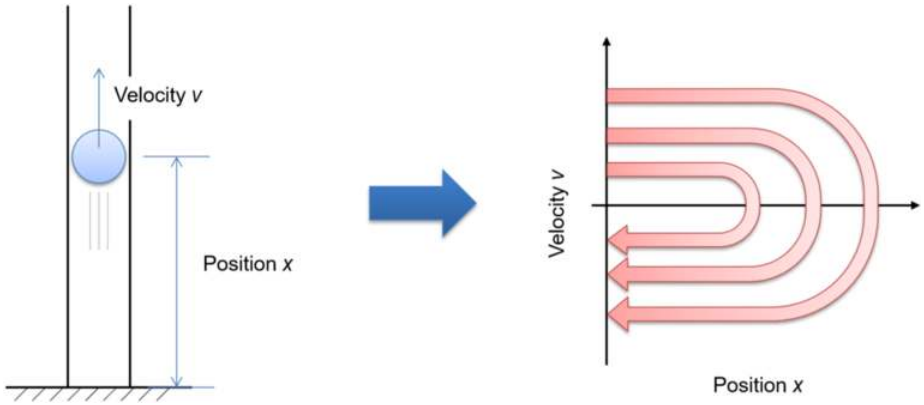


Figure **3.2.1**: A ball is thrown upward in a vertical tube (left) and a scheatic illustration of its phase space (right). The dynamic behavior of the ball can be visualize as a static trajectory in the phase space (red arrows).

One of the benefits of drawing a phase space is that it allows you to visually represent the dynamically changing behavior of a system as a static trajectory in it. This provides a lot of intuitive, geometrical insight into the system's dynamics, which would be hard to infer if you were just looking at algebraic equations.

? Exercise 3.2.1

In the example above, when the ball hits the floor in the vertical tube, its kinetic energy is quickly converted into elastic energy (i.e., deformation of the ball), and then it is converted back into kinetic energy again (i.e., upward velocity). Think about how you can illustrate this process in the phase space. Are the two dimensions enough or do you need to introduce an additional dimension? Then try to illustrate the trajectory of the system going through a bouncing event in the phase space.

This page titled 3.2: Phase Space is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

3.3: What Can We Learn?

There are several important things you can learn from phase space visualizations. First, you can tell from the phase space what will eventually happen to a system's state in the long run. For a deterministic dynamical system, its future state is uniquely determined by its current state (hence, the name "deterministic"). Trajectories of a deterministic dynamical system will never branch off in its phase space (though they could merge), because if they did, that would mean that multiple future states were possible, which would violate the deterministic nature of the system. No branching means that, once you specify an initial state of the system, the trajectory that follows is uniquely determined too. You can visually inspect where the trajectories are going in the phase space visualization. They may diverge to infinity, converge to a certain point, or remain dynamically changing yet stay in a confined region in the phase space from which no outgoing trajectories are running out. Such a converging point or a region is called an *attractor*. The concept of attractors is particularly important for understanding the self-organization of complex systems. Even if it may look magical and mysterious, self-organization of complex systems can be understood as a process whereby the system is simply falling into one of the attractors in a high-dimensional phase space.

Second, you can learn how a system's fate depends on its initial state. For each attractor, you can find the set of all the initial states from which you will eventually end up falling into that attractor. This set is called the *basin of attraction* of that attractor. If you have more than one attractor in the phase space (and/or if the phase space also shows divergence to infinity), you can divide the phase space into several different regions. Such a "map" drawn on the phase space reveals how sensitive the system is to its initial conditions. If one region is dominating in the phase space, the system's fate doesn't depend much on its initial condition. But if there are several regions that are equally represented in the phase space, the system's fate sensitively depends on its initial condition.

? Exercise 3.3.1

Where are the attractor(s) in the phase space of the bouncing ball example created in Exercise 3.2.1? Assume that every time the ball bounces it loses a bit of its kinetic energy.

? Exercise 3.3.2

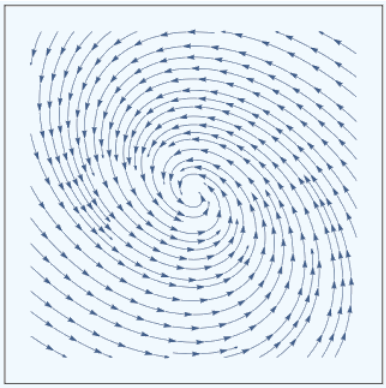
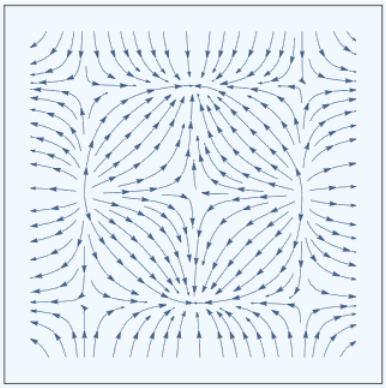
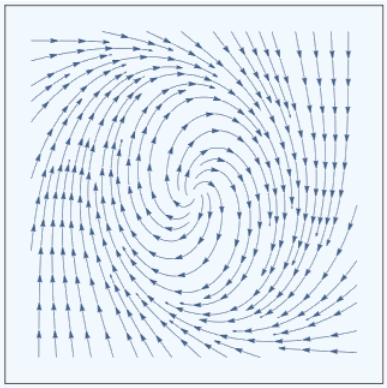
For each attractor obtained in Exercise **3.3.1** above, identify its basin of attraction.

Another thing you can learn from phase space visualizations is the stability of the system's states. If you see that trajectories are converging to a certain point or area in the phase space, that means the system's state is stable in that area. But if you see trajectories are diverging from a certain point or area, that means the system's state is unstable in that area. Knowing system stability is often extremely important to understand, design, and/or control systems in real-world applications. The following chapters will put a particular emphasis on this stability issue.

? Exercise 3.3.3

For each of the phase spaces shown below, identify the following:

- attractor(s)
- basin of attraction for each attractor
- stability of the system's state at several locations in the phase space

? Exercise 3.3.3

Consider a market where two equally good products, A and B, are competing with each other for market share. There is a customer review website where users of each product submit their ratings. Since there is no actual difference in the product quality, the average rating scores are about the same between the two products, but the customers can also see the total number of submitted ratings for each product, which shows how popular the product is in the market. Customers tend to adopt a more popular choice. Answer the following questions:

a. What would the phase space of this system look like?
b. Are there any attractors? Are there any basins of attraction?
c. How does the system's fate depend on its initial state?
d. If you were in charge of marketing product A, what would you do?

This page titled 3.3: What Can We Learn? is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

4: Discrete-Time Models I - Modeling

Discrete-time models are easy to understand, develop and simulate. They are easily implementable for stepwise computer simulations, and they are often suitable for modeling experimental data that are almost always already discrete. Moreover, they can represent abrupt changes in the system's states, and possibly chaotic dynamics, using fewer variables than their continuous-time counterparts.

4.1: Discrete-Time Models with Difference Equations

4.2: Classifications of Model Equations

4.3: Simulating Discrete-Time Models with One Variable

4.4: Simulating Discrete-Time Models with Multiple Variables

4.5: Building Your Own Model Equation

4.6: Building Your Own Model Equations with Multiple Variables

---

This page titled 4: Discrete-Time Models I - Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.1: Discrete-Time Models with Difference Equations

Discrete-time models are easy to understand, develop and simulate. They are easily implementable for stepwise computer simulations, and they are often suitable for modeling experimental data that are almost always already discrete. Moreover, they can represent abrupt changes in the system's states, and possibly chaotic dynamics, using fewer variables than their continuous-time counterparts (this will be discussed more in Chapter 9). The discrete-time models of dynamical systems are often called D*ifference Equations*, because you can rewrite any first-order discrete-time dynamical system with a state variable x (Eq. (3.1.1)), i.e.,

$$x_t = F(x_{t-1}, t) \tag{4.1.1}$$

into a "difference" form

$$\Delta x = x_t - x_{t-1} = x_t = F(x_{t-1}, t) - x_{t-1} \tag{4.1.2}$$

which is mathematically more similar to differential equations. But in this book, we mostly stick to the original form that directly specifies the next value of x, which is more straightforward and easier to understand.

Note that Equation 4.1.1 can also be written as

$$x_{t+1} = F(x_t, t) \tag{4.1.3}$$

which is mathematically equivalent to Equation 4.1.1 and perhaps more commonly used in the literature. But we will use the notation with x_t, x_{t-1}, x_{t-2}, etc., in this textbook, because this notation makes it easier to see how many previous steps are needed to calculate the next step (e.g., if the right hand side contains x_{t-1} and x_{t-2}, that means you will need to know the system's state in previous two steps to determine its next state). From a difference equation, you can produce a series of values of the state variable x over time, starting with initial condition x_0:

$$x_0, x_1, x_2, x_3 \ldots. \tag{4.1.4}$$

This is called *time series*. In this case, it is a prediction made using the difference equation model, but in other contexts, time series also means sequential values obtained by empirical observation of real-world systems as well.

Here is a very simple example of a discrete-time, discrete-state dynamical system. The system is made of two interacting components: A and B. Each component takes one of two possible states: Blue or red. Their behaviors are determined by the following rules:

- A tries to stay the same color as B.
- B tries to be the opposite color of A.

These rules are applied to their states simultaneously in discrete time steps

? Exercise 4.1.1

Write the state transition functions $F_A(S_A, S_B)$ and $F_B(S_A, S_B)$ for this system, where S_A and S_B are the states of A and B, respectively.

? Exercise 4.1.2

Produce a time series of S_A, S_B starting with an initial condition with both components in blue, using the model you created. What kind of behavior will arise?

This page titled 4.1: Discrete-Time Models with Difference Equations is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.2: Classifications of Model Equations

There are some technical terminologies I need to introduce before moving on to further discussions:

Linear system A dynamical equation whose rules involve just a linear combination of state variables (a constant times a variable, a constant, or their sum).

Nonlinear system Anything else (e.g., equation involving squares, cubes, radicals, trigonometric functions, etc., of state variables).

First-order system A difference equation whose rules involve state variables of the immediate past (at time $t-1$) only[a].

Higher-order system Anything else.

[a]Note that the meaning of "order" in this context is different from the order of terms in polynomials.

Autonomous system A dynamical equation whose rules don't explicitly include time t or any other external variables.

Non-autonomous system A dynamical equation whose rules do include time t or other external variables explicitly.

? Exercise 4.2.1

Decide whether each of the following examples is (1) linear or nonlinear, (2) first-order or higher-order, and (3) autonomous or non-autonomous

1. $x_t = ax_{t-1} + b$
2. $x_t = ax_{t-1} + bx_{t-2} + cx_{t-3}$
3. $x_t = ax_{t-1}(1 - x_{t-1})$
4. $x_t = ax_{t-1} + bxt - 2^2 + \sqrt[]{x_{t-1}x_{t-3}}$
5. $x_t = ax_{t-1}x_{t-2} + bx_{t-3} + sin(t)$
6. $x_t = ax_{t-1} + by_{t-1}, y_t = cx_{t-1} + dy_{t-1}$

Also, there are some useful things that you should know about these classifications:

Non-autonomous, higher-order difference equations can always be converted into autonomous, first-order forms, by introducing additional state variables.

For example, the second-order difference equation

$$x_t = x_{t-1} + x_{t-2} \tag{4.2.1}$$

(which is called the ***Fibonacci sequence*** can be converted into a first-order form by introducing a "memory" variable y as follows:

$$y_t = x_{t-1} \tag{4.2.2}$$

Using this, x_{t-2} can be rewritten as y_{t-1}. Therefore the equation can be rewritten as follows:

$$x_t = x_{t-1} + y_{t-1} \tag{4.2.3}$$

$$y_t = x_{t-1} \tag{4.2.4}$$

This is now first-order. This conversion technique works for third-order or any higher-order equations as well, as long as the historical dependency is finite. Similarly, a non-autonomous equation

$$x_t = x_{t-1} + t \tag{4.2.5}$$

can be converted into an autonomous form by introducing a "clock" variable z as follows:

$$z_t = z_{t-1} + 1, z_0 = 1 \tag{4.2.6}$$

This definition guarantees $z_{t-1} = t$. Using this, the equation can be rewritten as

$$x_t = x_{t-1} + z_{t-1}, \tag{4.2.7}$$

which is now autonomous. These mathematical tricks might look like some kind of cheating, but they really aren't. The take-home message on this is that autonomous first-order equations can cover all the dynamics of any non-autonomous, higher-order equations. This gives us confidence that we can safely focus on autonomous first-order equations without missing anything fundamental. This is probably why autonomous first-order difference equations are called by a particular name: *iterative maps*.

? Exercise 4.2.2

Convert the following difference equations into an autonomous, first-order form.

1. $x_t = x_{t-1}(1 - x_{t-1})sint$
2. $x_t = x_{t-1} + x_{t-2} - x_{t-3}$

Another important thing about dynamical equations is the following distinction between linear and nonlinear systems:

> Linear equations are always analytically solvable, while nonlinear equations don't have analytical solutions in general.

Here, an *analytical solution* means a solution written in the form of $x_t = f(t)$ without using state variables on the right hand side. This kind of solution is also called a closed form solution because the right hand side is "closed," i.e., it only needs t and doesn't need x. Obtaining a closed-form solution is helpful because it gives you a way to calculate (i.e., predict) the system's state directly from t at any point in time in the future, without actually simulating the whole history of its behavior. Unfortunately this is not possible for nonlinear systems in most cases.

This page titled 4.2: Classifications of Model Equations is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.3: Simulating Discrete-Time Models with One Variable

Now is the time to do our very first exercise of *computer simulation* of discrete-time models in *Python*. Let's begin with this very simple linear difference equation model of a scalar variable x:

$$x_t = ax_{t-1} \tag{4.3.1}$$

Here, a is a model parameter that specifies the ratio between the current state and the next state. Our objective is to find out what kind of behavior this model will show through computer simulation.

When you want to conduct a computer simulation of any sort, there are at least three essential things you will need to program, as follows:

Three essential components of computer simulation

Initialize. You will need to set up the initial values for all the state variables of the system.
Observe. You will need to define how you are going to monitor the state of the system. This could be done by just printing out some variables, collecting measurements in a list structure, or visualizing the state of the system.
Update. You will need to define how you are going to update the values of those state variables in every time step. This part will be defined as a function, and it will be executed repeatedly.

We will keep using this three-part architecture of simulation codes throughout this textbook. All the sample codes are available from the textbook's website at http://bingweb.binghamton.edu/~sayama/textbook/, directly linked from each Code example if you are reading this electronically.

To write the initialization part, you will need to decide how you are going to represent the system's state in your computer code. This will become a challenging task when we work on simulations of more complex systems, but at this point, this is fairly easy. Since we have just one variable, we can decide to use a symbol x to represent the state of the system. So here is a sample code for initialization:

Code 4.1:

```
def initialize():
    global x
    x = 1.
```

In this example, we defined the initialization as a Python function that initializes the global variable x from inside the function itself. This is why we need to declare that x is global at the beginning. While such use of global variables is not welcomed in mainstream computer science and software engineering, I have found that it actually makes simulation codes much easier to understand and write for the majority of people who don't have much experience in computer programming. Therefore,we frequently use global variables throughout this textbook.

Next, we need to code the observation part. There are many different ways to keep track of the state variables in a computer simulation, but here let's use a very simple approach to create a time series list. This can be done by first initializing the list with the initial state of the system, and then appending the current state to the list each time the observation is made. We can define the observation part as a function to make it easier to read. Here is the updated code:

Code 4.2:

```
def initialize():
    global x, result
    x = 1.
    result = [x]

def observe():
    global x, result
    result.append(x)
```

Finally, we need to program the updating part. This is where the actual simulation occurs. This can be implemented in another function:

Code 4.3:

```
def update():
    global x, result
    x = a * x
```

Note that the last line directly overwrites the content of symbol x. There is no distinction between x_{t-1} and x_t,but this is okay, because the past values of x are stored in the results list in the observe function. Now we have all the three crucial components implemented. We also need to add parameter settings to the beginning, and the iterative loop of updating at the end. Here, let's let $a = 1.1$ so that the value of x increases by 10% in each time step. The completed simulation code looks like this:

Code 4.4:

```
a = 1.1

def initialize():
    global x, result
    x = 1.
    result = [x]

def observe():
    global x, result
    result.append(x)

def update():
    global x, result
    x = a * x

initialize()
for t in xrange(30):
    update()
    observe()
```

Here we simulate the behavior of the system for 30 time steps. Try running this code inyour Python and make sure you don't get any errors. If so, congratulations! You have just completed the first computer simulation of a dynamical system. Of course, this code doesn't produce any output, so you can't be sure if the simulation ran correctly or not. An easy way to see the result is to add the following line to the end of the code:

Code 4.5:

```
print result
```

If you run the code again, you will probably see something like this:

Code 4.6:

```
[1.0, 1.1, 1.2100000000000002, 1.3310000000000004, 1.4641000000000006,
 1.6105100000000008, 1.771561000000001, 1.9487171000000014,
 2.1435888100000016, 2.357947691000002, 2.5937424601000023,
 2.853116706110003, 3.1384283767210035, 3.4522712143931042,
 3.797498335832415, 4.177248169415656, 4.594972986357222,
 5.054470284992944, 5.559917313492239, 6.115909044841463,
 6.72749994932561, 7.400249944258172, 8.140274938683989,
 8.954302432552389, 9.849732675807628, 10.834705943388391,
 11.91817653772723, 13.109994191499954, 14.420993610649951,
 15.863092971714948, 17.449402268886445]
```

We see that the value of x certainly grew. But just staring at those numbers won't give us much information about how it grew. We should visualize these numbers to observe the growth process in a more intuitive manner. To create a visual plot, we will need to use the *matplotlib* library[1]. Here we use its *pylab* environment included in matplotlib. Pylab provides a MATLAB-like working environment by bundling matplotlib's plotting functions together with a number of frequently used mathematical/computational functions (e.g., trigonometric functions, random number generators, etc.). To use pylab, you can add the following line to the beginning of your code[2]:

Code 4.7:

```
from pylab import *
```

And then you can add the following lines to the end of your code:

Code 4.8:

```
plot(result)
show()
```

The completed code as a whole is as follows (you can download the actual Python code by clicking on the name of the code in this textbook):

Code 4.9: exponential-growth.py

```
from pylab import *

a = 1.1

def initialize():
    global x, result
    x = 1.
    result = [x]

def observe():
    global x, result
    result.append(x)

def update():
    global x, result
    x = a * x

initialize()
for t in xrange(30):
    update()
    observe()

plot(result)
show()
```

Run this code and you should get a new window that looks like Fig. 4.3.1. If not, check your code carefully and correct any mistakes. Once you get a successfully visualized plot, you can clearly see an *exponential growth* process in it. Indeed, Equation 4.3.1 is a typical mathematical model of exponential growth or exponential decay. You can obtain several distinct behaviors by varying the value of a.

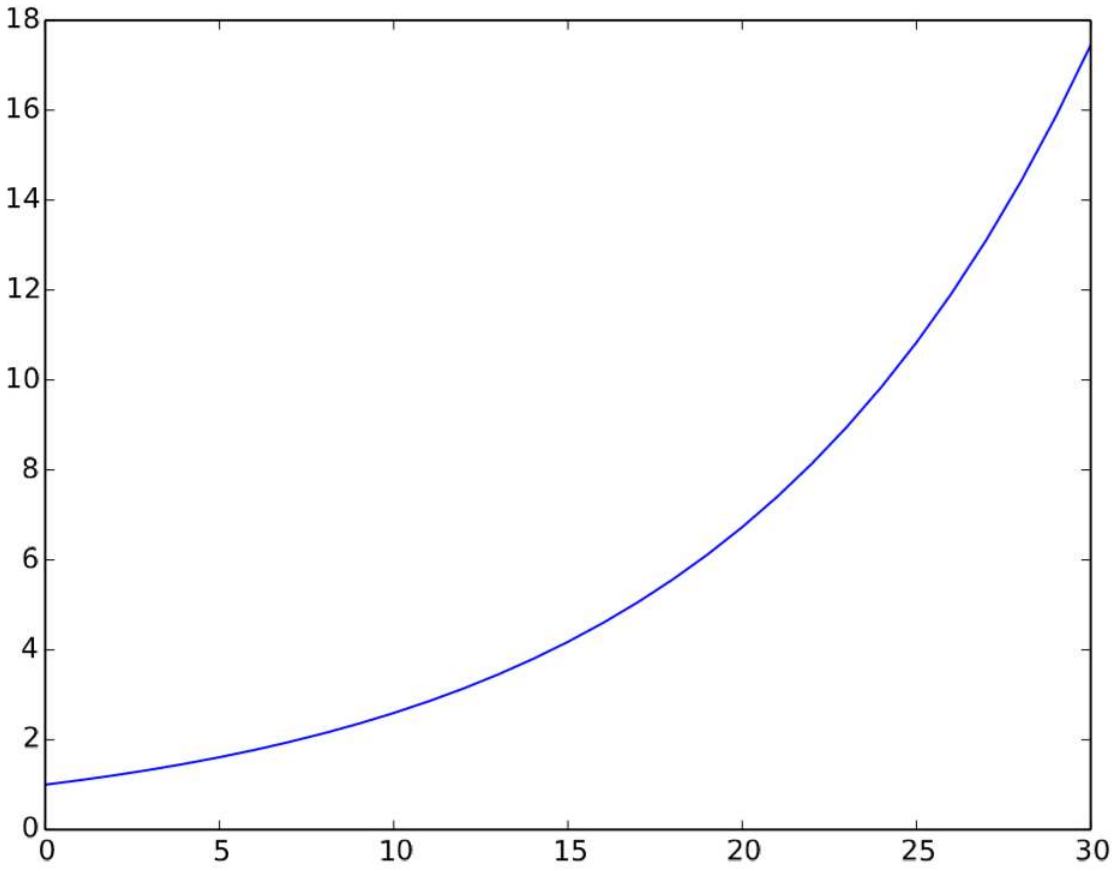


Figure 4.3.1: Visual output of Code 4.9.

? Exercise 4.3.1

Conduct simulations of this model for various values of parameter a to see what kind of behaviors are possible in this model and how the value of a determines the resulting behavior.

For your information, there are a number of options you can specify in the plot function, such as adding a title, labeling axes, changing color, changing plot ranges, etc. Also, you can manipulate the result of the plotting interactively by using the icons located at the bottom of the plot window. Check out matplotlib's website (http://matplotlib.org/) to learn more about those additional features yourself. In the visualization above, the horizontal axis is automatically filled in by integers starting with 0. But if you want to give your own time values (e.g., at intervals of 0.1), you can simulate the progress of time as well, as follows (revised parts are marked with ###):

Code 4.10: exponential-growth-time.py

```
from pylab import *

a = 1.1

def initialize():
    global x, result, t, timesteps ###
    x = 1.
    result = [x]
    t = 0. ###
    timesteps = [t] ###

def observe():
    global x, result, t, timesteps ###
    result.append(x)
    timesteps.append(t) ###

def update():
    global x, result, t, timesteps ###
    x = a * x
    t = t + 0.1 ###

initialize()
while t < 3.: ###
    update()
    observe()

plot(timesteps, result) ###
show()
```

? Exercise 4.3.2

Implement a simulation code of the following difference equation:

$$x_t = ax_{t-1} + b, x_0 = 1 \tag{4.3.2}$$

This equation is still linear, but now it has a constant term in addition to ax_{t-1}. Some real-world examples that can be modeled in this equation include fish population growth with constant removals due to fishing, and growth of credit card balances with constant monthly payments (both with negative b). Change the values of a and b, and see if the system's behaviors are the same as those of the simple exponential growth/decay model

[1]It is already included in Anaconda and Enthought Canopy. If you are using a different distribution of Python, matplotlib is freely available from http://matplotlib.org/.

[2]Another way to import pylab is to write " `import pylab` " instead, which is recommended by more programming-savvy people. If you do this, however, pylab's functions have to have the prefix `pylab` added to them, such as `pylab.plot(result)` . For simplicity, we use " `from pylab import *`" throughout this textbook

This page titled 4.3: Simulating Discrete-Time Models with One Variable is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.4: Simulating Discrete-Time Models with Multiple Variables

Now we are making a first step to complex systems simulation. Let's increase the number of variables from one to two. Consider the following difference equations:

$$x_t = 0.5_{t-1} + y_{t-1} \tag{4.4.1}$$

$$y_t = -0.5x_{t-1} + y_{t-1} \tag{4.4.2}$$

with

$$x_0 = 1, y_0 = 1 \tag{4.4.3}$$

These equations can also be written using vectors and a matrix as follows:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0.5 & 1 \\ -0.5 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}_{t-1} \tag{4.4.4}$$

Try implementing the simulation code of the equation above. This may seem fairly straightforward, requiring only minor changes to what we had before. Namely, you just need to simulate two difference equations simultaneously. Your new code may look like this:

Code 4.11: oscillation-wrong.py

```
from pylab import *

def initialize():
    global x, y, xresult, yresult
    x = 1.
    y = 1.
    xresult = [x]
    yresult = [y]

def observe():
```

```
    global x, y, xresult, yresult
    xresult.append(x)
    yresult.append(y)

def update():
    global x, y, xresult, yresult
    x = 0.5 * x + y
    y = -0.5 * x + y

initialize()
for t in xrange(30):
    update()
    observe()

plot(xresult, 'b-')
plot(yresult, 'g--')
show()
```

What I did in this code is essentially to repeat things for both `x` and `y`. Executing two `plot` commands at the end produces two curves in a single chart. I added the `'b-'` and `'g--'` options to the `plot` functions to draw `xresult` in a blue solid curve and y result in a green dashed curve. If you run this code, it produces a decent result (Fig. 4.4.1), so things might look okay. However, there is one critical mistake I made in the code above. Can you spot it? This is a rather fundamental issue in complex systems simulation in general, so we'd better notice and fix it earlier than later. Read the code carefully again, and try to find where and how I did it wrong.

Did you find it? The answer is that I did not do a good job in updating `x` and `y` simultaneously. Look at the following part of the code:

Code 4.12:

```
    x = 0.5 * x + y
    y = -0.5 * x + y
```

Note that, as soon as the first line is executed, the value of `x` is overwritten by its new value. When the second line is executed, the value of `x` on its right hand side is already updated,but it should still be the original value. In order to correctly simulate simultaneous updating of `x` and `y`, you will need to do something like this:

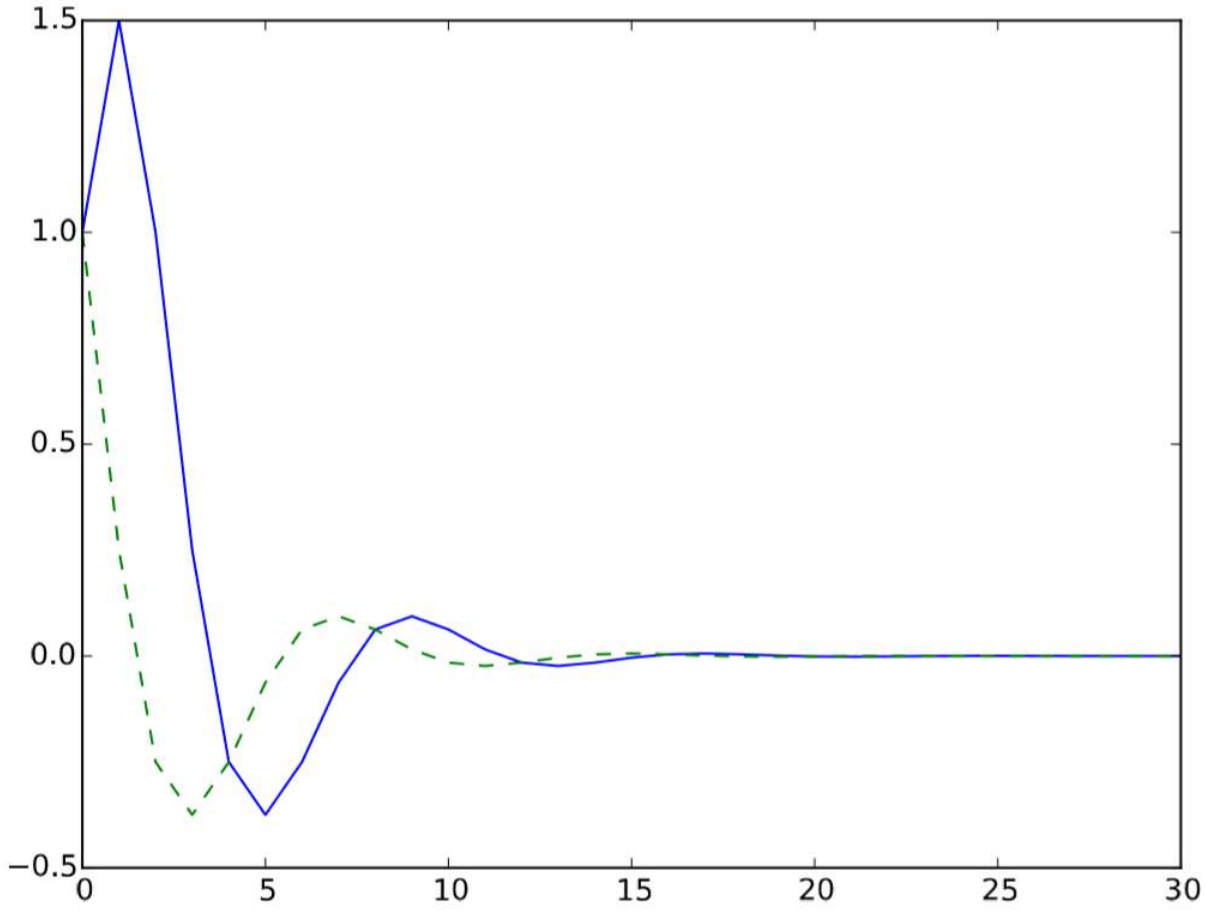


Figure 4.4.1: Visual output of Code 4.11. This result is actually wrong.

Code 4.13: oscillation-correct.py

```
nextx = 0.5 * x + y
nexty = -0.5 * x + y
x, y = nextx, nexty
```

Here we have two sets of state variables, `x`, `y` and `nextx`, `nexty`. We first calculate the next state values and store them in `nextx`, `nexty`, and then copy them to `x`, `y`, which will be used in the next iteration. In this way, we can avoid any interference between the state variables during the updating process. If you apply this change to the code, you will get a correct simulation result (Fig. 4.4.2).

The issue of how to implement *simultaneous updating* of multiple variables is a common technical theme that appears in many complex systems simulation models, as we will discuss more in later chapters. As seen in the example above, a simple solution is to prepare two separate sets of the state variables, one for now and the other for the immediate future, and calculate the updated values of state variables without directly modifying them during the updating. In the visualizations above, we simply plotted the state variables over time, but there is

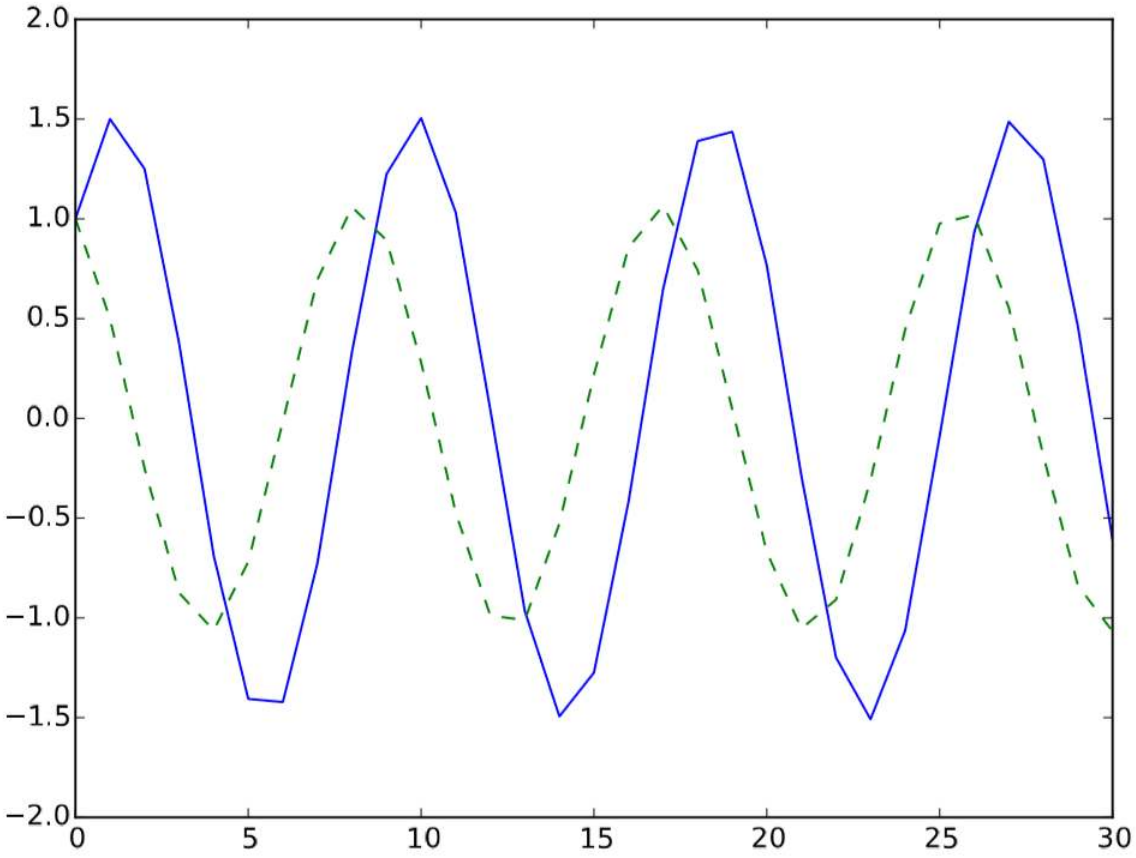


Figure **4.4.2**: Visual output of the simulation result based on a correctly implemented code with Code 4.13.

another way of visualizing simulation results in a phase space. If your model involves just two state variables (or three if you know 3-D plotting), you should try this visualization to see the structure of the phase space. All you need to do is to replace the following part

Code 4.14:

```
plot(xresult)
plot(yresult)
```

with this:

Code 4.15: oscillation-correct-phasespace.py

```
plot(xresult, yresult)
```

This will produce a trajectory of the system state in an $x - y$ phase space, which clearly shows that the system is in an oval, periodic oscillation (Fig. 4.4.3), which is a typical signature of a linear system.

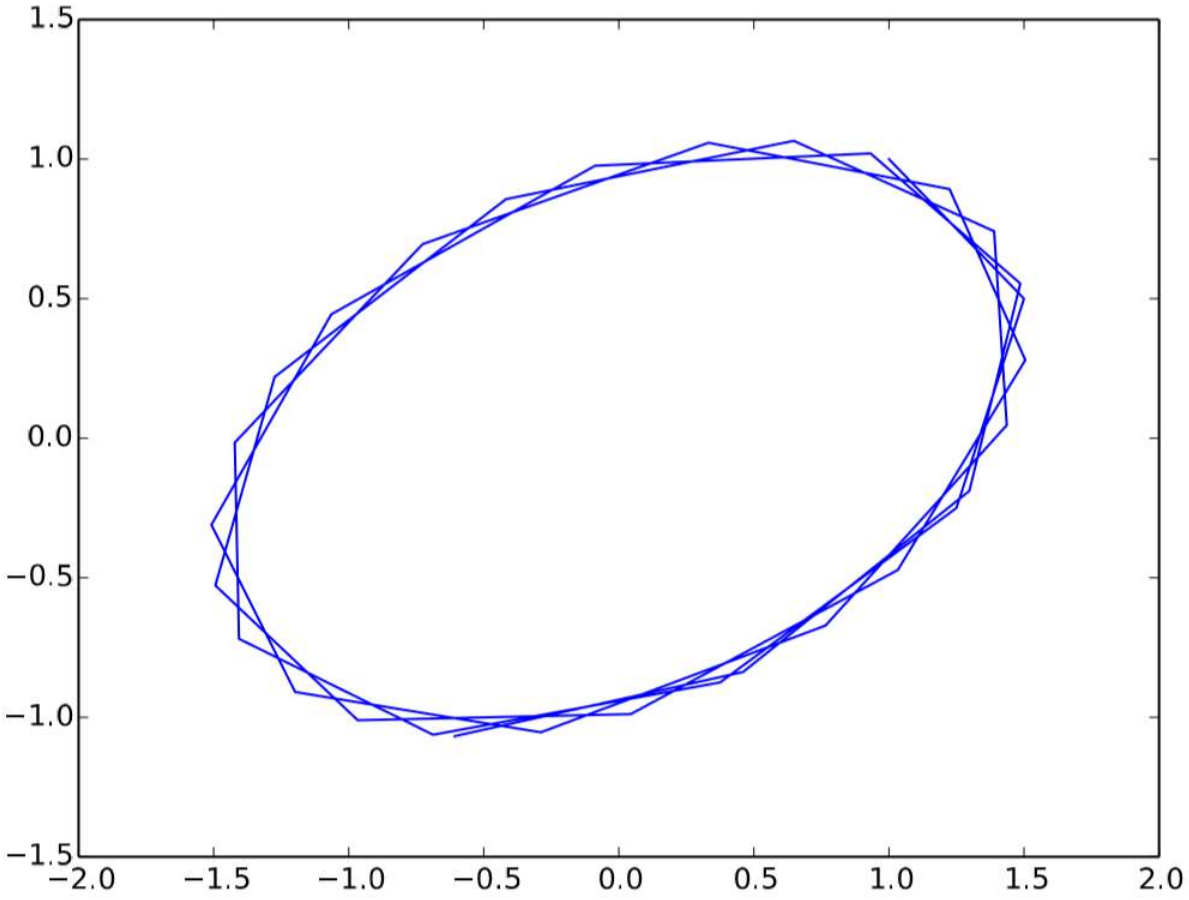


Figure 4.4.3: Visualization of the simulation result in a phase space using Code 4.15.

? Exercise 4.4.1

Simulate the above two-variable system using several different co-efficients in the equations and see what kind of behaviors can arise.

? Exercise 4.4.2

Simulate the behavior of the following *Fibonacci sequence*. You first need to convert it into a two-variable first-order difference equation, and then implement a simulation code for it.

$$x_t = x_{t-1} + x_{t-2}, x_0 = 1, x_1 = 1 \tag{4.4.5}$$

If you play with this simulation model for various coefficient values, you will soon notice that there are only certain kinds of behaviors possible in this system. Sometimes the curves show exponential growth or decay, or sometimes they show more smooth oscillatory behaviors. These two behaviors are often combined to show an exponentially growing oscillation, etc. But that's about it. You don't see any more complex behaviors coming out of this model. This is because the system is linear, i.e., the model equation is composed of a simple linear sum of first-order terms of state variables. So here is an important fact you should keep in mind:

> Linear dynamical systems can show only exponential growth/decay, periodic oscillation, stationary states (no change), or their hybrids (e.g., exponentially growing oscillation)[a].

[a]Sometimes they can also show behaviors that are represented by polynomials (or products of polynomials and exponentials) of time. This occurs when their coefficient matrices are *non-diagonalizable*.

In other words, these behaviors are signatures of linear systems. If you observe such behavior in nature, you may be able to assume that the underlying rules that produced the behavior could be linear.

This page titled 4.4: Simulating Discrete-Time Models with Multiple Variables is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.5: Building Your Own Model Equation

Now that you know how to simulate the dynamics of difference equations, you may want to try building your own model equation and test its behaviors. Then a question arises: How do you build your own model equation?

Mathematics is a language that is used to describe the world. Just like that there is no single correct way to describe an idea in English, there is no single correct way to build a mathematical model equation either. It is highly dependent on your own personal literacy, creativity, and expressiveness in the language of mathematics. Ultimately, you just need to keep "reading" and "writing" math every day, in order to get better in mathematical model building. Having said that, there are some practical tips I can offer to help you build your own model equations. Here they are:

Practical tips for mathematical model building

1. If you aren't sure where to start, just grab an existing model and tweak it.
2. Implement each model assumption one by one. Don't try to reach the final model in one jump.
3. To implement a new assumption, first identify which part of the model equation represents the quantity you are about to change, replace it with an unknown function, and then design the function.
4. Whenever possible, adopt the simplest mathematical form.
5. Once your equation is complete, check if it behaves as you desired. It is often helpful to test its behavior with extreme values assigned to variables and/or parameters.

Let me illustrate each of those tips by going through an example. Consider building another population growth model that can show not just exponential growth but also convergence to a certain population limit. Any ideas about where to start? As the first tip suggests, you could use an existing model that is similar to what you want to model, and then modify it for your needs. Since this example is about population growth, we already know one such model: the exponential growth model. So let's start there:

$$x_t = ax_{t-1} \tag{4.5.1}$$

This model is very simple. It consists of just two components: growth ratio a and population size x_{t-1}.

The second tip says you should take a step-by-step approach. So let's think about what we additionally need to implement in this model. Our new model should show the following two behaviors:

- Exponential growth
- Convergence to a certain population limit

We should check the first one first. The original model already shows exponential growth by itself, so this is already done. So we move on to the second one. Obviously, the original model doesn't show such convergence, so this is what we will need to implement in the model. The third tip says you need to focus on a specific component to be revised. There are many options here. You could revise a, x_{t-1}, or you could even add another term to the right hand side. But in this particular case, the convergence to a certain limit means that the growth ratio a should go to 1 (i.e., no net growth). So, we can focus on the a part, and replace it by an unknown function of the population size $f(x_{t-1})$. The model equation now looks like this:

$$x_t = f(x_{t-1})x_{t-1} \tag{4.5.2}$$

Now your task just got simpler: just to design function $f(x)$. Think about constraints it has to satisfy. $f(x)$ should be close to the original constant a when the population is small, i.e., when there are enough environmental resources, to show exponential growth. In the meantime, $f(x)$ should approach 1 when the population approaches a carrying capacity of the environment(let's call it K for now). Mathematically speaking, these constraints mean that the function $f(x)$ needs to go through the following two points: $(x, f(x)) = (0, a)$ and $(K, 1)$.

And this is where the fourth tip comes in. If you have no additional information about what the model should look like, you should choose the simplest possible form that satisfies the requirements. In this particular case, a straight line that connects the two points above is the simplest one (Fig. 4.5.1), which is given by

$$f(x) = -\frac{a-1}{K}x + a. \tag{4.5.3}$$

You can plug this form into the original equation, to complete a new mathematical equation:

$$x_t = -(\frac{a-1}{K}x_{t-1} + a)x_{t-1} \tag{4.5.4}$$

Now it seems your model building is complete. Following the fifth tip, let's check if the new model behaves the way you intended. As the tip suggests, testing with extreme values often helps find out possible issues in the model. What happens when $x_{t-1} = 0$?

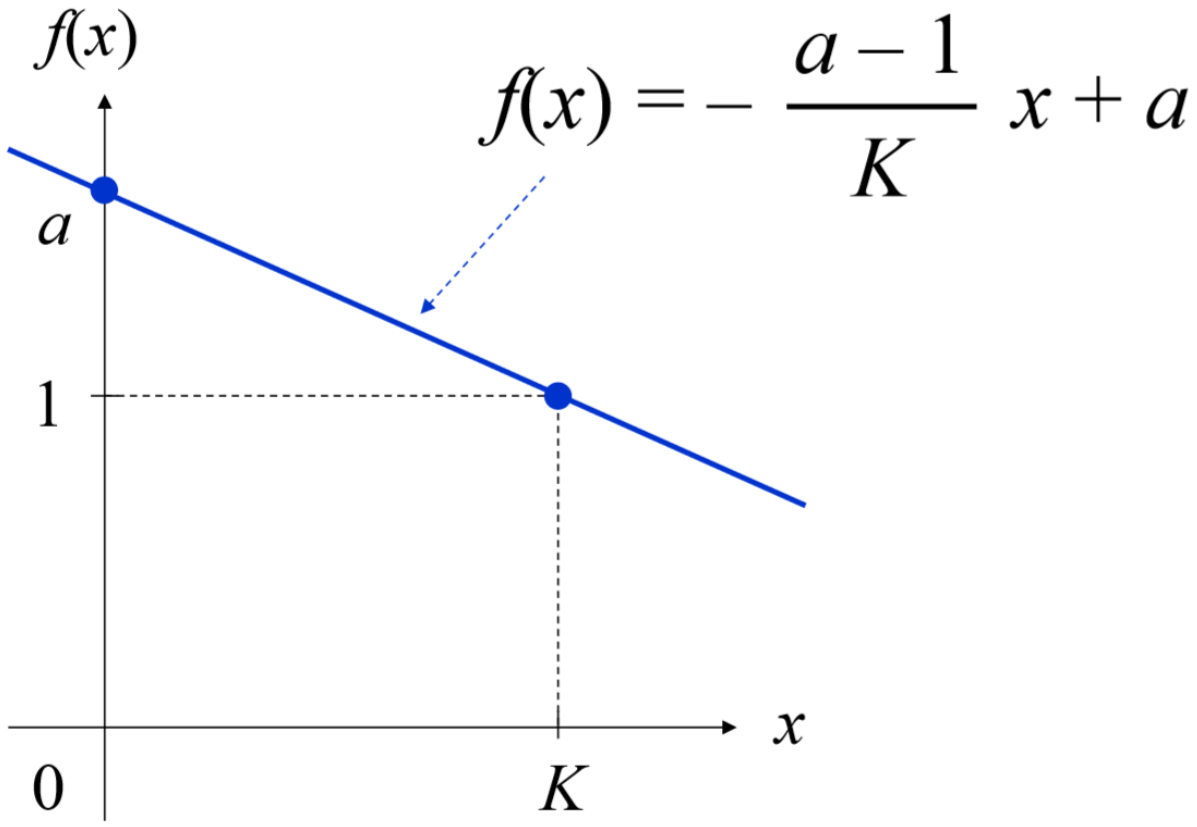


Figure **4.5.1**: *The simplest example of how the growth ratio $a = f(x)$ function of population size should behave as a x.*

In this case, the equation becomes $x_{t=0}$, so there is no growth. This makes perfect sense; if there are no organisms left, there should be no growth. Another extreme case: What happens when $x_{t-1} = K$? In this case, the equation becomes $x_t = x_{t-1}$, i.e., the system maintains the same population size. This is the new convergent behavior you wanted to implement, so this is also good news. Now you can check the behaviors of the new model by computer simulations.

? Exercise 4.5.1

Simulate the behavior of the new population growth model for several different values of parameter a and initial condition x_0 to see what kind of behaviors are possible.

$$x_t = -(\frac{a-1}{K}x_{t-1} + a)x_{t-1} \tag{4.5.5}$$

For your information, the new model equation we have just derived above actually has a particular name; it is called the *logistic growth* model in mathematical biology and several other disciplines. You can apply a parameter substitution $r = a - 1$ to make the equation into a more well-known form:

$$x_t = -(\frac{a-1}{K}x_{t-1} + a)x_{t-1} \tag{4.5.6}$$

$$= -(\frac{r}{K}x_{t-1} + r + 1)x_{t-1} \tag{4.5.7}$$

$$= x_{t-1} + rx_{t-1}(1 - \frac{x_{t-1}}{K}) \tag{4.5.8}$$

This formula has two terms on its right hand side: the current population size (x_{t-1}) and the number of newborns ($rx_{t-1}(\cdots)$). If x is much smaller than K, the value inside the parentheses gets closer to 1, and thus the model is approximated by

$$x_t \approx x_{t-1} + rx_{t-1}. \tag{4.5.9}$$

This means that r times the current population is added to the population at each time step, resulting in exponential growth. But when x comes close to K, inside the parentheses approaches 0, so there will be no net growth.

? Exercise 4.5.2

Create a mathematical model of population growth in which the growth ratio is highest at a certain optimal population size, but it goes down as the population deviates from the optimal size. Then simulate its behavior and see how its behavior differs from that of the logistic growth model.

This page titled 4.5: Building Your Own Model Equation is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

4.6: Building Your Own Model Equations with Multiple Variables

We can take one more step to increase the complexity of the model building, by including more than one variable. Following the theme of population growth, let's consider ecological interactions between two species. A typical scenario would be the *predator-prey interaction*. Let's stick to the population-level description of the system so each species can be described by one variable (say, x and y). The first thing you should consider is each variable's inherent dynamics, i.e., what would happen if there were no influences coming from other variables. If there is always plenty of food available for the prey, we can assume the following:

- Prey grows if there are no predators.
- Predators die if there are no prey.

These assumptions can be diagrammatically illustrated in Fig. 4.6.1.

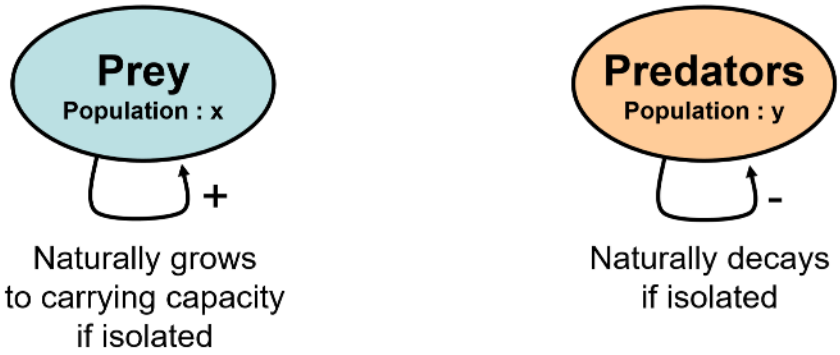


Figure **4.6.1**: Inherent dynamics of each of the prey and predator populations illustrated in a causal loop diagram.

The type of diagrams in Figure **4.6.1** are called a *causal loop diagram* in *System Dynamics* [26]. Each circle, or node, in this diagram represents a state variable in the system. The self-loop arrow attached to each node represents the effect of the variables on itself (e.g., the more prey there are, the faster their growth will be, etc.). The plus/minus signs next to the arrows show whether the effect is positive or negative. We can now consider the interactions between the two variables, i.e., how one variable influences the other and vice versa. Naturally, there should be the following effects:

- The prey's death rate increases as the predator population increases.
- The predators' growth rate increases as the prey population increases.

These interactions can be illustrated as arrows between nodes in the causal loop diagram (Fig. 4.6.2).

Now is the time to translate the structure of the system illustrated in the diagram into mathematical equations. Each arrow in the diagram tells us whether the effect is positive or negative, but they don't give any exact mathematical form, so we will need to create a mathematical representation for each (possibly using the aforementioned tips).

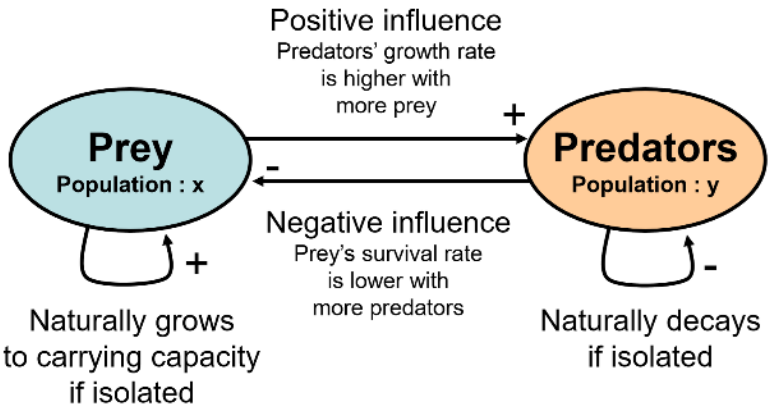


Figure **4.6.2**: Interactions between the prey and predator populations illustrated in a causal loop diagram.

https://math.libretexts.org/@go/page/7787

The inherent dynamics of the two variables are quite straightforward to model. Since we already know how to model growth and decay, we can just borrow those existing models as building components, like this:

$$x_t = x_{t-1} + r_x x_{t-1}(1 - \frac{x_{t-1}}{K}) \tag{4.6.1}$$

$$y_t = y_{t-1} - d_y y_{t-1} \tag{4.6.2}$$

Here, I used the logistic growth model for the prey (x) while using the exponential decay model for the predators (y). r_x is the growth rate of the prey, and d_y is the death rate of the predators ($0 < dy < 1$).

To implement additional assumptions about the predator-prey interactions, we need to figure out which part of the equations should be modified. In this example it is obvious, because we already know that the interactions should change the death rate of the prey and the growth rate of the predators. These terms are not yet present in the equations above, so we can simply add a new unknown term to each equation:

$$x_t = x_{t-1} + r x_{t-1}(1 - \frac{x_{t-1}}{K}) - d_x(y_{t-1})x_{t-1} \tag{4.6.3}$$

$$y_t = y_{t-1} - d y_{t-1} + r_y(x_{t-1})y_{t-1} \tag{4.6.4}$$

Now the problems are much better defined. We just need to come up with a mathematical form for d_x and r_y.

The death rate of the prey should be 0 if there are no predators, while it should approach 1 (= 100% mortality rate!) if there are too many predators. There are a number of mathematical formulas that behave this way. A simple example would be the following hyperbolic function

$$d_x(y) = 1 - \frac{1}{by + 1}, \tag{4.6.5}$$

where b determines how quickly d_x increases as y increases. The growth rate of the predators should be 0 if there are no prey, while it can go up indefinitely as the prey population increases. Therefore, the simplest possible mathematical form could be

$$r_y(x) = cx, \tag{4.6.6}$$

where c determines how quickly r_y increases as x increases. Let's put these functions back into the equations. We obtain the following:

$$x_t = x_{t-1} + r x_{t-1}(1 - \frac{x_{t-1}}{K}) - (1 - \frac{1}{by_{t-1} + 1})x_{t-1} \tag{4.6.7}$$

$$y_t = y_{t-1} - d y_{t-1} + c x_{t-1} y_{t-1} \tag{4.6.8}$$

? Exercise 4.6.1

Test the equations above by assuming extreme values for x and y, and make sure the model behaves as we intended.

? Exercise 4.6.2

Implement a simulation code for the equations above, and observe the model behavior for various parameter settings.

Figure 4.6.3 shows a sample simulation result with $r = b = d = c = 1$, $K = 5$ and $x_0 = y_0 = 1$. The system shows an oscillatory behavior, but as its phase space visualization (Fig. 4.6.3 right) indicates, it is not a harmonic oscillation seen in linear systems, but it is a nonlinear oscillation with distorted orbits.

The model we have created above is actually a variation of the *Lotka-Volterra model*, which describes various forms of predator-prey interactions. The Lotka-Volterra model is probably one of the most famous mathematical models of nonlinear dynamical systems that involves multiple variables.

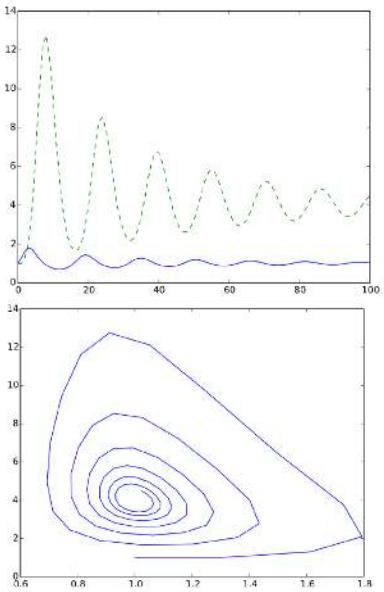


Figure 4.6.3: Simulation results of the predator-prey model. Left: State variables plotted over time. Blue (solid) = prey, green (dashed) = predators. Right: Phase space visualization of the same result.

? Exercise 4.6.3

Try several different parameter settings for r, b, d, and c, and see how the system's behavior changes. In some cases you may find an unrealistic, invalid behavior (e.g., indefinite growth of predators). If so, revise the model to fix the problem.

In this chapter , we reviewed some basics of mathematical modeling in difference equations. As I keep saying, the best way to learn modeling is through practice. Here are some more modeling exercises. I hope you enjoy them!

? Exercise 4.6.4

Develop a discrete-time mathematical model of two species competing for the same resource, and simulate its behavior.

? Exercise 4.6.5

Consider the dynamics of public opinions about political ideologies. For simplicity, let's assume that there are only three options: conservative, liberal, and neutral. Conservative and liberal are equally attractive (or annoying, maybe) to people, with no fundamental asymmetry between them. The popularities of conservative and liberal ideologies can be represented by two variables, p_c and p_t, respectively $(0 \leq pc \leq 1; 0 \leq pl \leq 1; 0 \leq pc + pl \leq 1)$. This implies that $1 - p_c - p_l = p_n$ represents the popularity of neutral.

Assume that at each election poll, people will change their ideological states among the three options according to the irrelative popularities in the previous poll. For example, the rate of switching from option X to option Y can be considered proportional to $(p_Y - p_X)$ if $p_Y > p_X$, or 0 otherwise. You should consider six different cases of such switching behaviors (conservative to liberal, conservative to neutral, liberal to conservative, liberal to neutral, neutral to conservative, and neutral to liberal) and represent them in dynamical equations. Complete a discrete-time mathematical model that describes this system, and simulate its behavior. See what the possible final outcomes are after a sufficiently long time period.
Complete a discrete-time mathematical model that describes this system, and simulate its behavior. See what the possible final outcomes are after a sufficiently long time period.

? Exercise 4.6.6

Revise the model of public opinion dynamics developed in the previous exercise so that the political parties of the two ideologies (conservative and liberal) run a political campaign to promote voters' switching to their ideologies from their competitions, at a rate ***inversely*** proportional to their current popularity (i.e., the less popular they are, the more intense their campaign will be). Simulate the behavior of this revised model and see how such political campaigning changes the dynamics of the system.

This page titled 4.6: Building Your Own Model Equations with Multiple Variables is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

5: Discrete-Time Models II - Analysis

5.1: Finding Equilibrium Points

5.2: 5.2 Phase Space Visualization of Continuous-State Discrete-Time Models

5.3: 5.3 Cobweb Plots for One-Dimensional Iterative Maps

5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model

5.5: Variable Rescaling of Discrete-Time Models

5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems

5.7: 5.7 Linear Stability Analysis of Discrete-Time Nonlinear Dynamical Systems

---

This page titled 5: Discrete-Time Models II - Analysis is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.1: Finding Equilibrium Points

When you analyze an autonomous, first-order discrete-time dynamical systems (a.k.a. iterative map)

$$x_t = F(x_{t-1}). \tag{5.1.1}$$

one of the first things you should do is to find its *equilibrium points* (also called fixed points or steady states), i.e., states where the system can stay unchanged over time. Equilibrium points are important for both theoretical and practical reasons. Theoretically, they are key points in the system's phase space, which serve as meaningful references when we understand the structure of the phase space. And practically, there are many situations where we want to sustain the system at a certain state that is desirable for us. In such cases, it is quite important to know whether the desired state is an equilibrium point, and if it is, whether it is stable or unstable. To find equilibrium points of a system, you can substitute all the x's in the equation with a constant x_{eq} (either scalar or vector) to obtain

$$x_{eq} = F(x_{eq}). \tag{5.1.2}$$

and then solve this equation with regard to x_{eq}. If you have more than one state variable, you should do the same for all of them.

Example 5.1.1:

For example, here is how you can find the equilibrium points of the logistic growth model:

$$x_t = x_{t-1} + r x_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{5.1.3}$$

Solution

Replacing all the x's with x_{eq} in Equation 5.1.3, we obtain the following:

Misplaced &

Misplaced &

Misplaced &

The result shows that the population will not change if there are no organisms $(x_{eq} = 0)$ or if the population size reaches the *carrying capacity* of the environment $(x_{eq} = K)$. Both make perfect sense.

Exercise 5.1.1

Obtain the equilibrium point(s) of the following difference equation:

$$x_t = 2x_{t-1} - x_{t-1}^2 \tag{5.1.4}$$

Exercise 5.1.2

Obtain the equilibrium point(s) of the following two-dimensional difference equation model:

$$x_t = x_{t-1} y_{t-1} \tag{5.1.5}$$

$$y_t = y_{t-1}(x_{t-1} - 1) \tag{5.1.6}$$

Exercise 5.1.3

Obtain the equilibrium point(s) of the following difference equation:

$$x_t = x_{t-1} - x_{t-2}^2 + 1 \tag{5.1.7}$$

Note that this is a second-order difference equation,so you will need to first convert it into a first-order form and then find the equilibrium point(s).

This page titled 5.1: Finding Equilibrium Points is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.2: 5.2 Phase Space Visualization of Continuous-State Discrete-Time Models

Once you find where the equilibrium points of the system are, the next natural step of analysis would be to draw the entire picture of its phase space (if the system is two or three dimensional).

For discrete-time systems with continuous-state variables (i.e., state variables that take real values), drawing a phase space can be done very easily using straightforward computer simulations, just like we did in Fig. 4.4.2. To reveal a large-scale structure of the phase space, however, you will probably need to draw many simulation results starting from different initial states. This can be achieved by modifying the initialization function so that it can receive specific initial values for state variables, and then you can put the simulation code into for loops that sweep the parameter values for a given range. For example:

Code 5.1: phasespace-drawing.py

```
from pylab import *

def initialize(x0, y0): ###
    global x, y, xresult, yresult
    x = x0 ###
    y = y0 ###
    xresult = [x]
    yresult = [y]

def observe():
    global x, y, xresult, yresult
    xresult.append(x)
    yresult.append(y)

def update():
    global x, y, xresult, yresult
    nextx = 0.5 * x + y
    nexty = -0.5 * x + y
    x, y = nextx, nexty
```

```
for x0 in arange(-2, 2, .5): ###
    for y0 in arange(-2, 2, .5): ###
        initialize(x0, y0) ###
        for t in xrange(30):
            update()
            observe()
        plot(xresult, yresult, 'b') ###
```

```
show()
```

Revised parts from the previous example are marked with ###. Here, the `arange` function was used to vary initial `x` and `y` values over $[-2, 2]$ at interval 0.5. For each initial state, a simulation is conducted for 30 steps, and then the result is plotted in blue (the 'b' option of `plot`). The output of this code is Fig. 5.2.1, which clearly shows that the phase space of this system is made of many concentric trajectories around the origin.

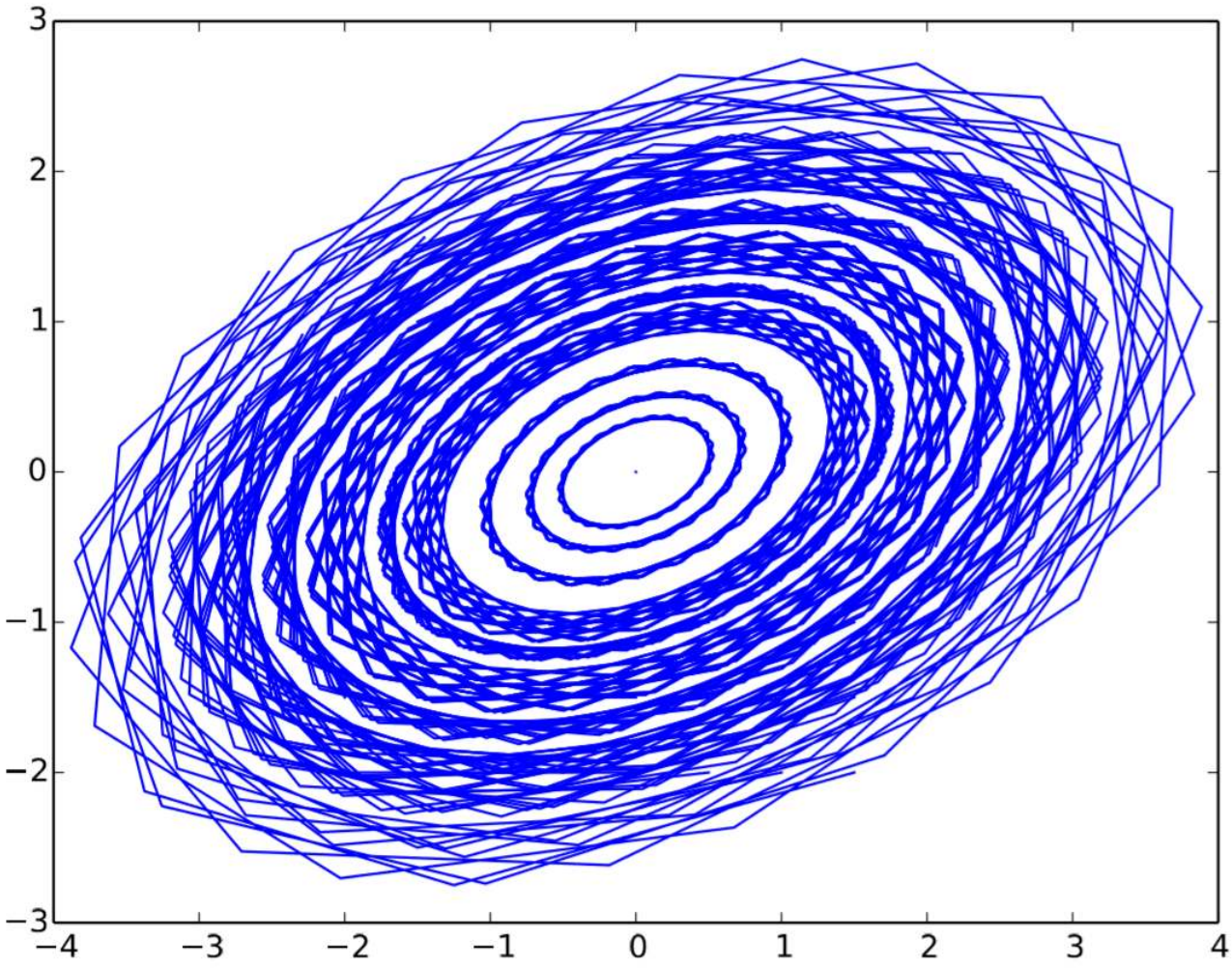


Figure **5.2.1**: Phase space drawn using Code 5.1.

? Exercise 5.2.1

Draw a phase space of the following two-dimensional difference equation model in Python:

$$x_t = x_{t-1} + 0.1(x_{t-1} - x_{t-1}y_{t-1}) \tag{5.2.1}$$

$$y_t = y_{t-1} + 0.1(y_{t-1} - x_{t-1}y_{t-1}) \tag{5.2.2}$$

$$(x > 0, x > 0) \tag{5.2.3}$$

Three-dimensional systems can also be visualized in a similar manner. For example, let's try visualizing the following three-dimensional difference equation model:

$$x_t = 0.5x + y \tag{5.2.4}$$

$$y_t = -0.5x + y \tag{5.2.5}$$

$$z_t = -x - y + z \tag{5.2.6}$$

Plotting in 3-D requires an additional `matplotlib` component called `Axes3D`. A sample code is given in Code 5.2. Note the new `import Axes3D` line at the beginning, as well as the two additional lines before the for loops. This code produces the result shown in Fig. 5.2.2.

Code 5.2: phasespace-drawing-3d.py

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

def initialize(x0, y0, z0):
    global x, y, z, xresult, yresult, zresult
    x = x0
    y = y0
    z = z0
    xresult = [x]
    yresult = [y]
    zresult = [z]

def observe():
    global x, y, z, xresult, yresult, zresult
    xresult.append(x)
    yresult.append(y)
    zresult.append(z)

def update():
    global x, y, z, xresult, yresult, zresult
    nextx = 0.5 * x + y
    nexty = -0.5 * x + y
    nextz = - x - y + z
    x, y, z = nextx, nexty, nextz
```

```
ax = gca(projection='3d')

for x0 in arange(-2, 2, 1):
    for y0 in arange(-2, 2, 1):
        for z0 in arange(-2, 2, 1):
            initialize(x0, y0, z0)
            for t in xrange(30):
                update()
                observe()
            ax.plot(xresult, yresult, zresult, 'b')

show()
```

Note that it is generally not a good idea to draw many trajectories in a 3-D phase space, because the visualization would become very crowded and difficult to see. Drawing a small number of characteristic trajectories is more useful.

In general, you should keep in mind that phase space visualization of discrete-time models may not always give images that are easily visible to human eye. This is because the state of a discrete-time system can jump around in the phase space, and thus the trajectories can cross over each other (this will not happen for continuous-time models). Here is an example. Replace the content of the update function in Code 5.1 with the following:

Code 5.3: phasespace-drawing-bad.py

```
    nextx = - 0.5 * x - 0.7 * y
    nexty = x - 0.5 * y
```

As a result, you get Fig. 5.2.3. While this may be aesthetically pleasing, it doesn't help our understanding of the system very much because there are just too many trajectories overlaid in the diagram. In this sense, the straightforward phase space visualization may not always be helpful for analyzing discrete-time dynamical systems. In the following sections, we will discuss a few possible workarounds for this problem.

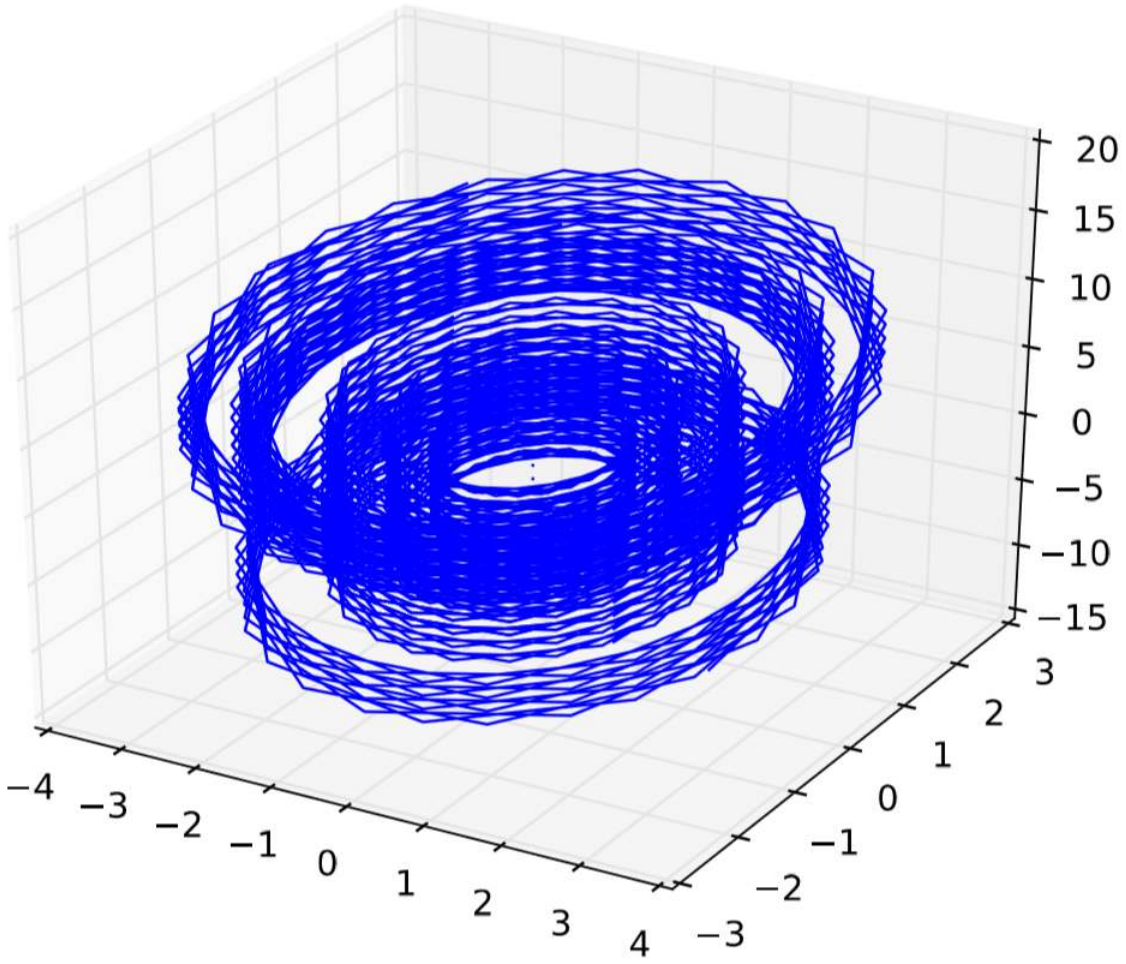


Figure **5.2.2**: Three-dimensional phase space drawn with Code 5.2. If you are drawing this from an interactive environment, such as Anaconda Spyder or Enthought Canopy, you can rotate the 3-D plot interactively by clicking and dragging.

Figure **5.2.3**: Phase space drawn with Code 5.3.

This page titled 5.2: 5.2 Phase Space Visualization of Continuous-State Discrete-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.3: 5.3 Cobweb Plots for One-Dimensional Iterative Maps

One possible way to solve the overcrowded phase space of a discrete-time system is to create two phase spaces, one for time $t-1$ and another for t, and then draw trajectories of the system's state in a meta-phase space that is obtained by placing those two phase spaces orthogonally to each other. In this way, you would potentially disentangle the tangled trajectories to make them visually understandable.

However, this seemingly brilliant idea has one fundamental problem. It works only for one-dimensional systems, because two- or higher dimensional systems require fouror more dimensions to visualize the meta-phase space, which can't be visualized in the three-dimensional physical world in which we are confined.

This meta-phase space idea is still effective and powerful for visualizing the dynamics of one-dimensional iterative maps. The resulting visualization is called a *cobweb plot*, which plays an important role as an intuitive analytical tool to understand the nonlinear dynamics of one-dimensional systems.

Here is how to manually draw a cobweb plot of a one-dimensional iterative map, $x_t = f(x_{t-1})$, with the range of x_t being $[x_{min}, x_{max}]$. Get a piece of paper and a pen, and do the following:

1. Draw a square on your paper. Label the bottom edge as the axis for x_{t-1}, and the left edge as the axis for x_t. Label the range of their values on the axes (Fig. 5.3.1).

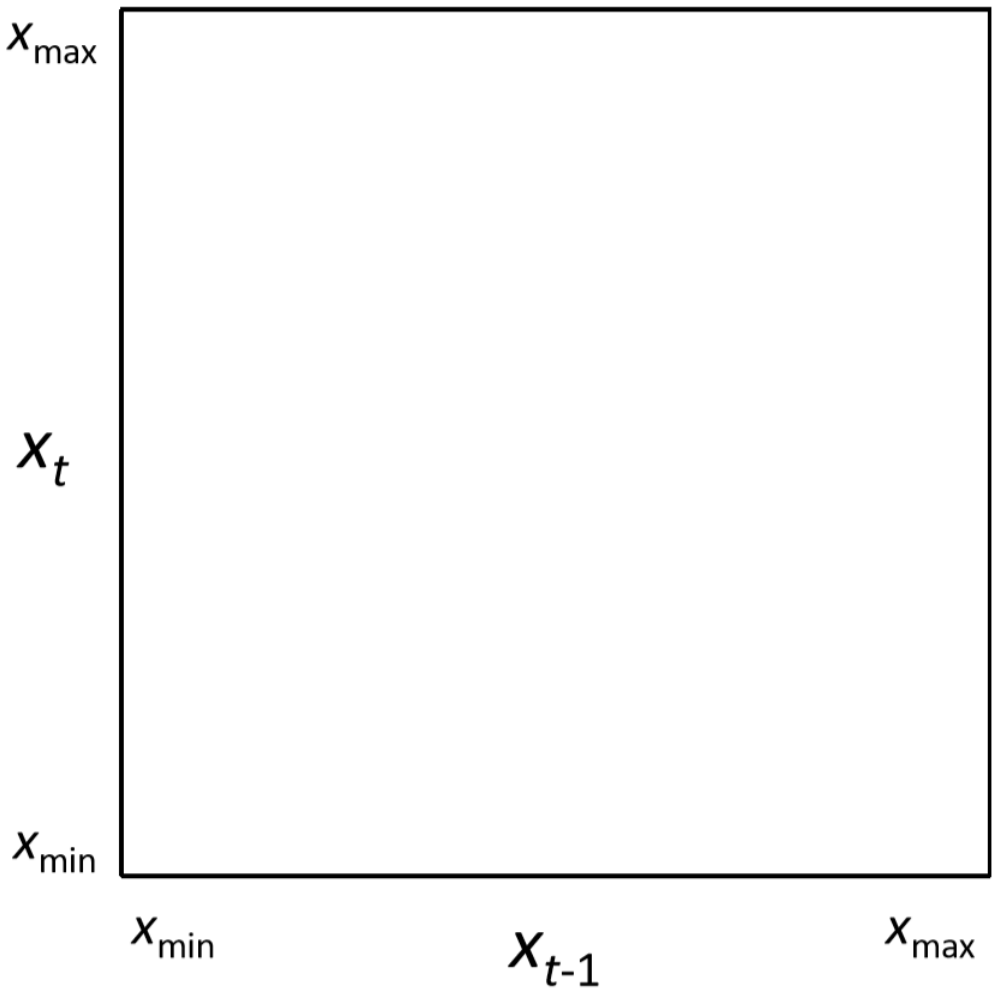


Figure **5.3.1**: Drawing a cobweb plot (1).

2. Draw a curve $x_t = f(x_{t-1})$ and a diagonal line $x_t = x_{t-1}$ within the square (Fig. 5.3.2). Note that the system's equilibrium points appear in this plot as the points where the curve and the line intersect.

3. Draw a trajectory from x_{t-1} to x_t. This can be done by using the curve $x_t = f(x_{t-1})$ (Fig. 5.3.3). Start from a current state value on the bottom axis (initially, this is the initial value x_0, as shown in Fig. 5.3.3), and move vertically until you reach the curve. Then switch the direction of the movement to horizontal and reach the left axis. You end up at the next value of the system's state (x_1 in Fig. 5.3.3). The two red arrows connecting the two axes represent the trajectory between the two consecutive time points.

4. Reflect the new state value back onto the horizontal axis. This can be done as a simple mirror reflection using the diagonal line (Fig. 5.3.4). This completes one step of the "manual simulation" on the cobweb plot.

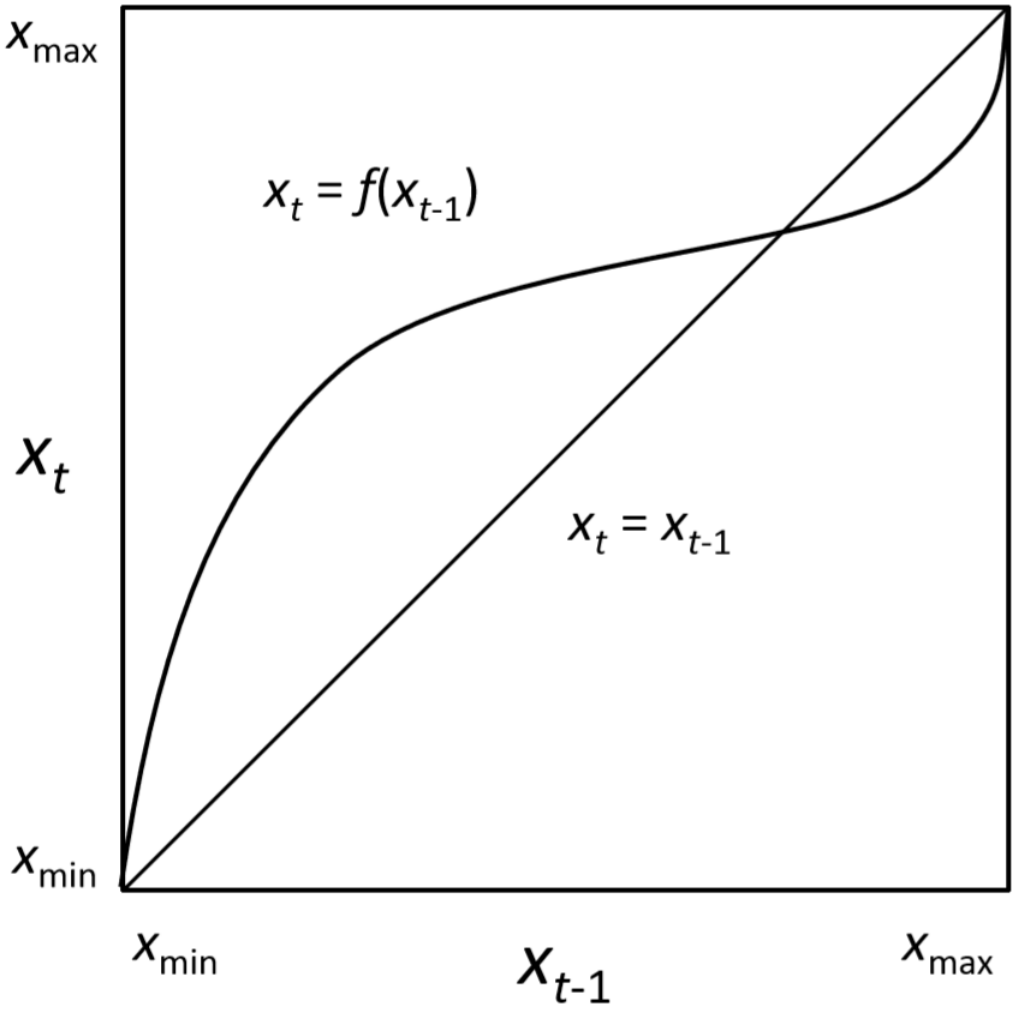


Figure **5.3.2**: Drawing a cobweb plot (2).

5. Repeat the steps above to see where the system eventually goes (Fig. 5.3.5).
6. Once you get used to this process, you will notice that you don't really have to touch either axis. All you need to do to draw a cobweb plot is to bounce back and forth between the curve and the line (Fig. 5.3.6)—*move vertically to the curve, horizontally to the line, and repeat.*

? Exercise 5.3.1

Draw a cobweb plot for each of the following models:

- $$x_t = x_{t-1} + 0.1, x_0 = 0.1$$
- $$x_t = 1.1x_{t-1}, x_0 = 0.1$$

? Exercise 5.3.2

Draw a cobweb plot of the following logistic growth model with $r = 1, K = 1$, $N_0 = 0.1$:

$$N_t = N_{t-1} + rN_{t-1}(1 - \frac{N_{t-1}}{K}) \tag{5.3.1}$$

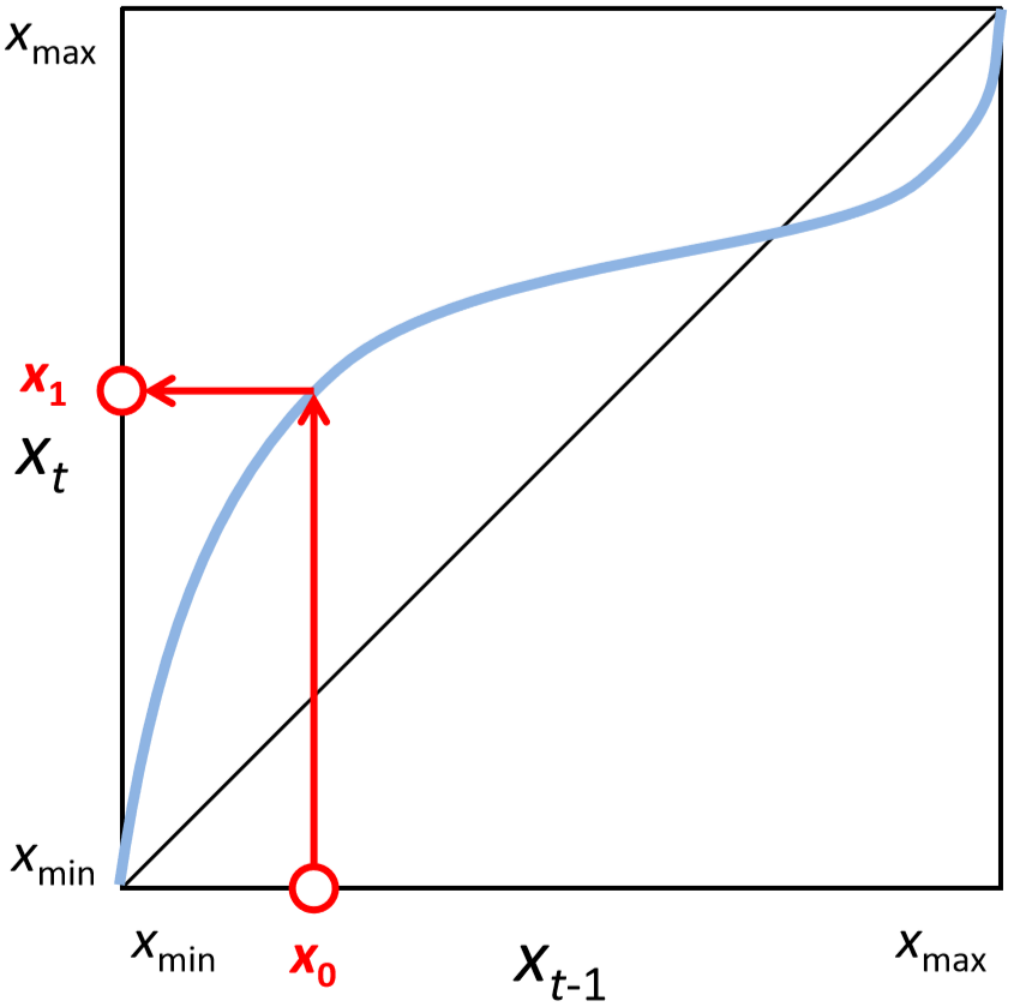


Figure 5.3.3: Drawing a cobweb plot (3).

Cobweb plots can also be drawn using Python. Code 5.4 is an example of how to draw a cobweb plot of the exponential growth model (Code 4.9). Its output is given in Fig. 5.4.1.

Code 5.4: cobweb-plot.py

```
from pylab import *

a = 1.1

def initialize():
    global x, result
    x = 1.
    result = [x]

def observe():
    global x, result
    result.append(x)

def f(x): ### iterative map is now defined as f(x)
```



Figure **5.3.4**: Drawing a cobweb plot (4).

```
    return a * x

def update():
    global x, result
    x = f(x)

initialize()
for t in xrange(30):
    update()
    observe()

### drawing diagonal line
xmin, xmax = 0, 20
plot([xmin, xmax], [xmin, xmax], 'k')

### drawing curve
rng = arange(xmin, xmax, (xmax - xmin) / 100.)
```

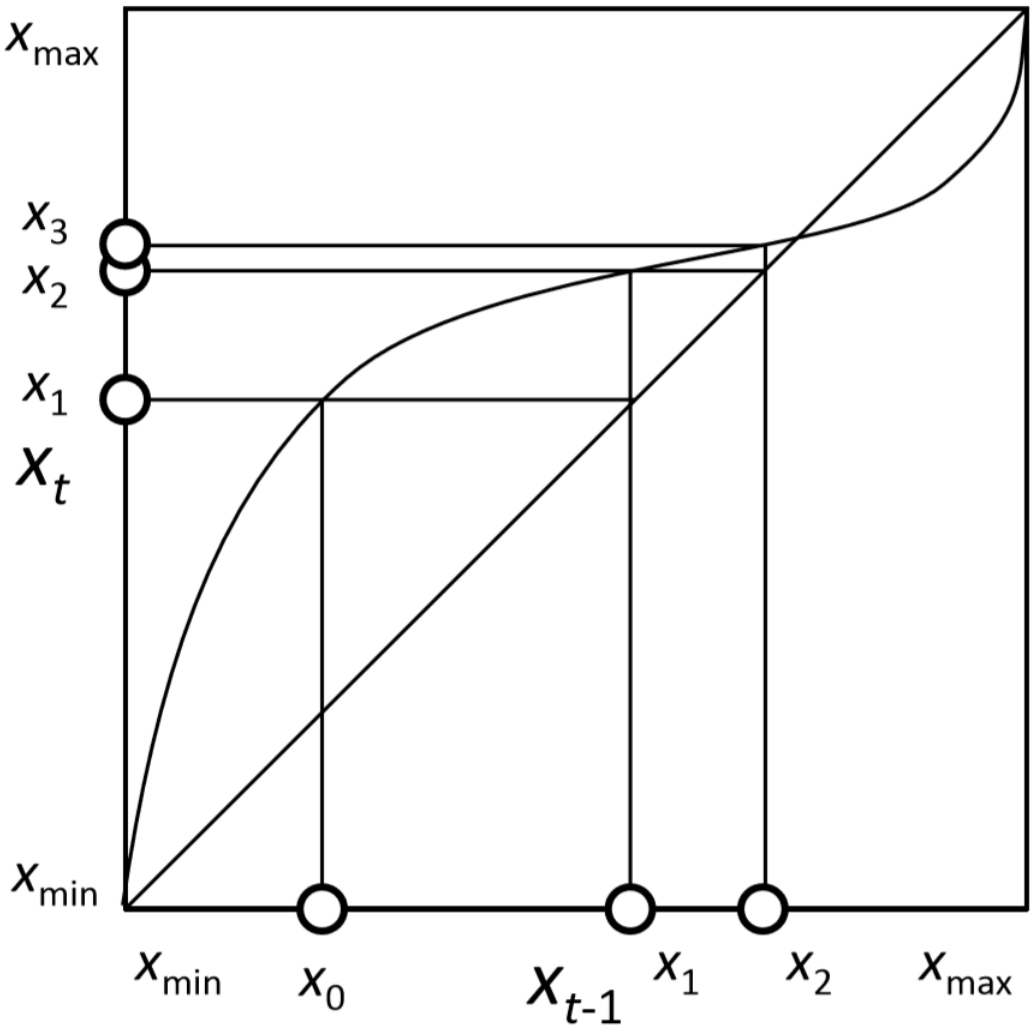


Figure **5.3.5**: Drawing a cobweb plot (5).

```
plot(rng, map(f, rng), 'k')

### drawing trajectory
horizontal = [result[0]]
vertical = [result[0]]
for x in result[1:]:
    horizontal.append(vertical[-1])
    vertical.append(x)
    horizontal.append(x)
    vertical.append(x)
plot(horizontal, vertical, 'b')

show()
```

? Exercise 5.3.3

Using Python, draw a cobweb plot of the logistic growth model with $r = 2.5$, $K = 1$, $N_0 = 0.1$.

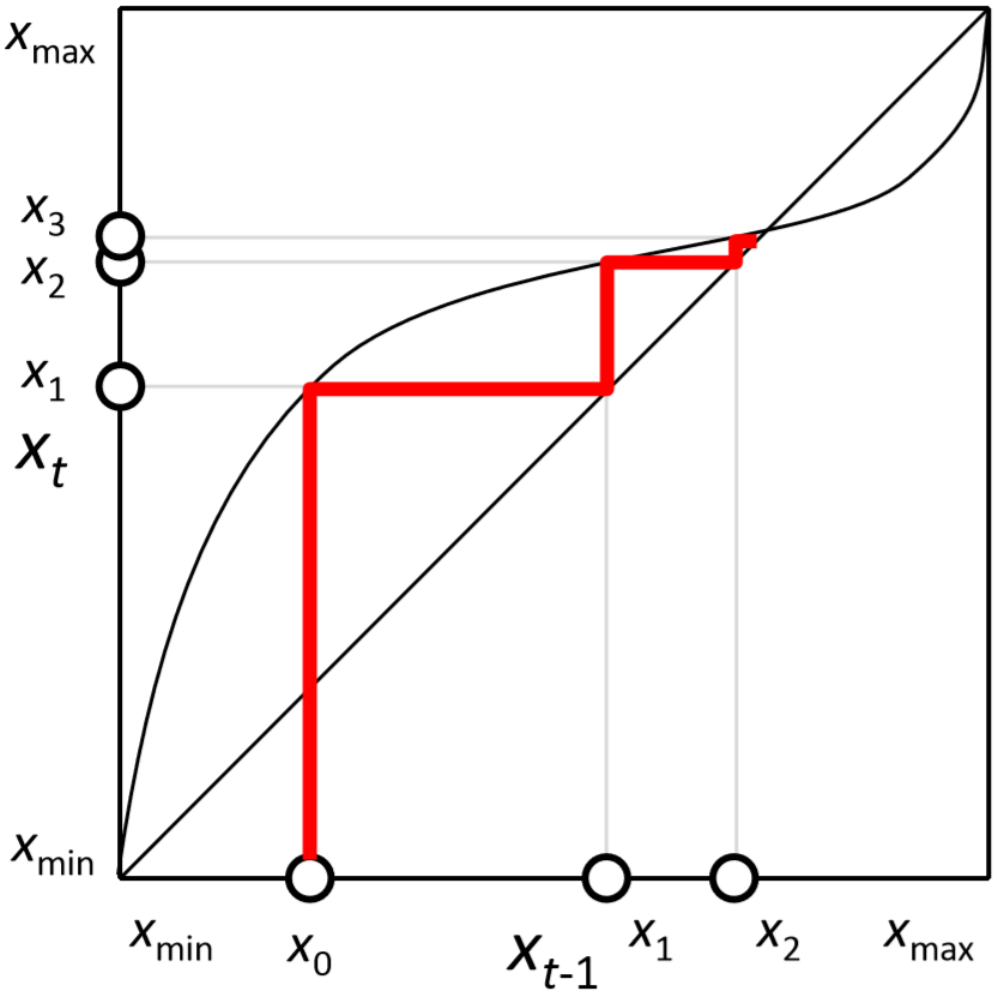


Figure 5.3.6: Drawing a cobweb plot (6).

This page titled 5.3: 5.3 Cobweb Plots for One-Dimensional Iterative Maps is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model

The cobweb plot approach discussed above works only for one-dimensional systems, because we can't embed such plots for any higher dimensional systems in a 3-D physical space. However, this dimensional restriction vanishes i*f the system's states are discrete and finite*. For such a system, you can always enumerate all possible state transitions and create the entire phase space of the system as a *state-transition graph*, which can be visualized reasonably well even within a 2-D visualization space.

Here is an example. Let's consider the following second-order (i.e., two-dimensional) difference equation:

$$x_t = x_{t-1}x_{t-2} \mod 6 \tag{5.4.1}$$

The "mod 6" at the end of the equation means that its right hand side is always a remainder of the division of $x_{t-1}x_{t-2}$ by 6. This means that the possible state of x is limited only to 0, 1, 2, 3, 4, or 5, i.e., the state of the system (x, y) (where y is the previous

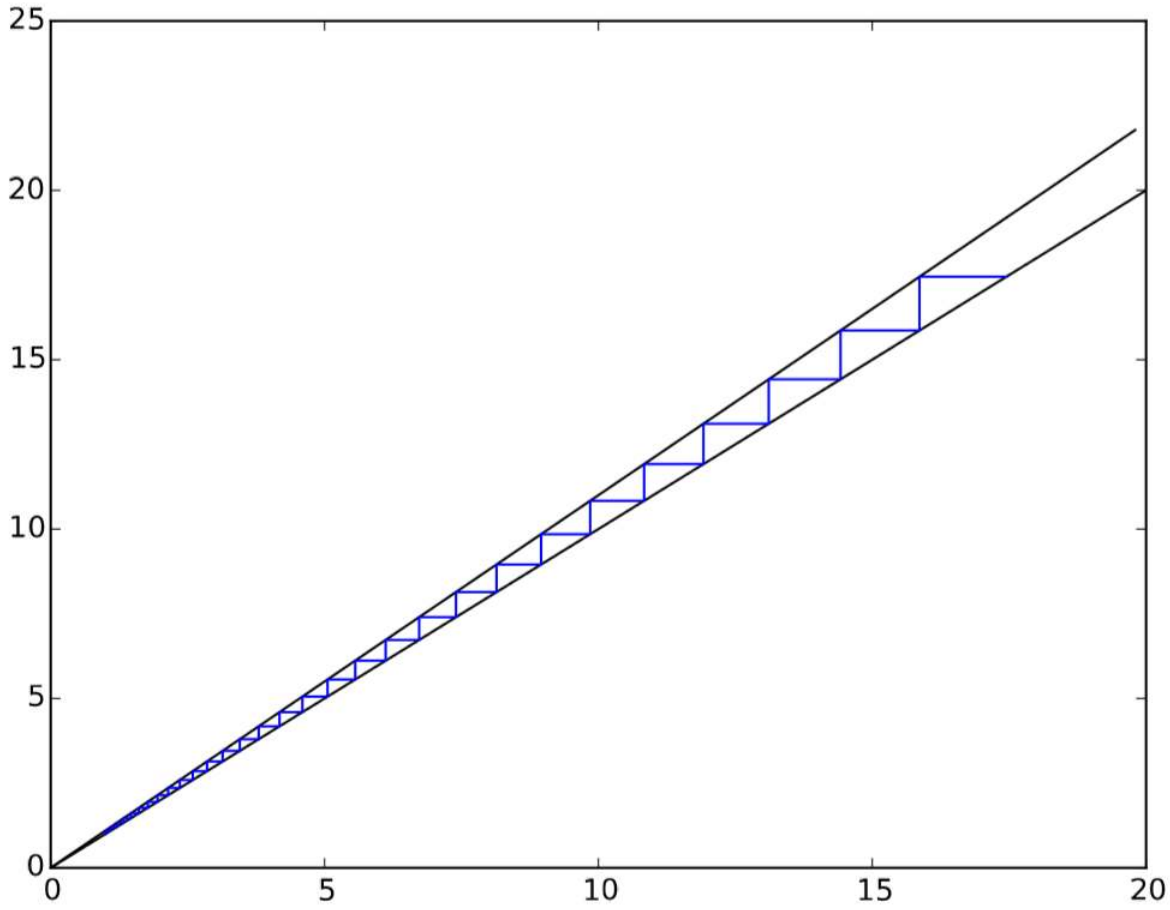


Figure 5.4.1: Visual output of Code 5.4. This is a cobweb plot of the exponential growth model simulated in Code 4.9.

value of x) is confined within a finite set of $6 \times 6 = 36$ possible combinations ranging from (0, 0) to (5, 5). It is very easy to enumerate all the 36 states and numerically study which state each of them will transition to. The result of this analysis looks like:

- $(0, 0) \to (0, 0)$
- $(1, 0) \to (0, 1)$
- $(2, 0) \to (0, 2)$
- ...

- $(3,2) \to (0,3)$
- $(4,2) \to (2,4)$
- $(5,2) \to (4,5)$
- ...
- $(3,5) \to (3,3)$
- $(4,5) \to (2,4)$
- $(5,5) \to (1,5)$

By enumerating all the state transitions, we obtain a list of connections between the discrete states, which forms a *network*, or a *graph* in mathematical terminology. We will learn more about modeling and analysis of networks later in this textbook, but I can give you a little bit of a preview here. We can use a Python module called *NetworkX [27]* to construct and visualize the network[1]. See Code 5.5.

Code 5.5: graph-based-phasespace.py

```
from pylab import *
import networkx as nx

g = nx.DiGraph()

for x in range(6):
    for y in range(6):
        g.add_edge((x, y), ((x * y) % 6, x))

ccs = [cc for cc in nx.connected_components(g.to_undirected())]
n = len(ccs)
w = ceil(sqrt(n))
h = ceil(n / w)
for i in xrange(n):
    subplot(h, w, i + 1)
    nx.draw(nx.subgraph(g, ccs[i]), with_labels = True)

show()
```

In this example, a network object, named g, is constructed using NetworkX's `DiGraph` (directed graph) object class, because the state transitions have a direction from one state to another. I also did some additional tricks to improve the result of the visualization. I split the network into multiple separate *connected components* using NetworkX's c `onnected_ components` function, and then arranged them in a grid of plots using pylab's subplot feature. The result is shown in Fig. 5.5.1.

Each of the six networks represent one connected component, which corresponds to one *basin of attraction*. The directions of transitions are indicated by thick line segments instead of conventional arrowheads, i.e., each transition goes from the thin end to the thick end of a line (although some thick line segments are hidden beneath the nodes in the crowded areas). You can follow the directions of those links to find out where the system is going in each basin of attraction. The attractor may be a single state or a

dynamic loop. Don't worry if you don't understand the details of this sample code. We will discuss how to use NetworkX in more detail later.

? Exercise 5.4.1

Draw a phase space of the following difference equation within the range $0 \leq x < 100$ by modifying Code 5.5:

$$x_t = x_{t-1}^{x_{t-1}} \ mod 100 \tag{5.4.2}$$

This page titled 5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.5: Variable Rescaling of Discrete-Time Models

Aside from finding equilibrium points and visualizing phase spaces, there are many other mathematical analyses you can do to study the dynamics of discrete-time models. But

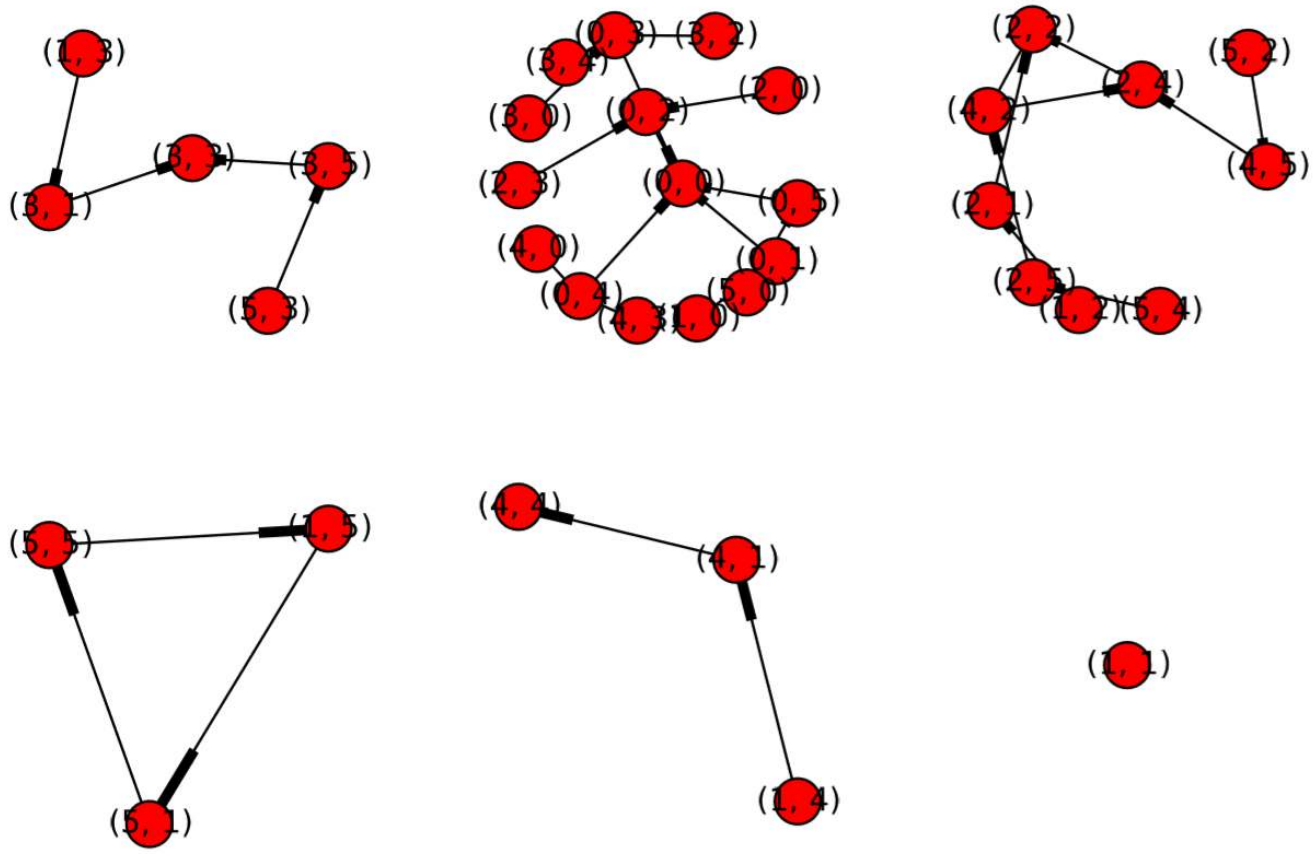


Figure **5.5.1**: Graph-based phase space of Eq. (5.4.1) drawn with Code 5.5.

before jumping right into such paper-and-pencil mathematical work, I would like to show you a very useful technique that can make your mathematical work much easier. It is called *variable rescaling*.

Variable rescaling is a technique to eliminate parameters from your model without losing generality. The basic idea is this: Variables that appear in your model represent quantities that are measured in some kind of units, but those units can be arbitrarily chosen without changing the dynamics of the system being modeled. This must be true for all scientific quantities that have physical dimensions—switching from inches to centimeters should not cause any change in how physics works! This means that you have the freedom to choose any convenient unit for each variable, some of which may simplify your model equations.

Let's see how variable rescaling works with an example. Here is the logistic growth model we discussed before:

$$x_t = x_{t-1} + r x_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{5.5.1}$$

There is only one variable in this model, x, so there is only one unit we can change, i.e., the unit of the population counts. The first step of variable rescaling is to replace each of the variables with a new notation made of a non-zero constant and a new state variable, like this:

$$x \to \alpha x' \tag{5.5.2}$$

With this replacement, Equation ??? becomes

$$\alpha x_t{}' = \alpha x'_{t-1} + r\alpha x'_{t-1}\left(1 - \frac{\alpha x'_{t-1}}{K}\right). \tag{5.5.3}$$

The second step is to simplify the equation and then find a "convenient" choice for the constant that will make your equation simpler. This is a rather open-ended process with many different directions to go, so you will need to do some explorations to find out what kind of unit choices make your model simplest. For the logistic growth model, Equation ??? can be further simplified, for example, like

$$x'_t = x'_{t-1} + rx'_{t-1}\left(1 - \frac{\alpha x'_{t-1}}{K}\right) \tag{5.5.4}$$

$$= x'_{t-1}\left(1 + r\left(1 - \frac{\alpha x'_{t-1}}{K}\right)\right) \tag{5.5.5}$$

$$= x'_{t-1}\left(1 + r - \frac{r\alpha x'_{t-1}}{K}\right) \tag{5.5.6}$$

$$= (1+r)x'_{t-1}\left(1 - \frac{r\alpha x'_{t-1}}{K(1+r)}\right). \tag{5.5.7}$$

Here, the most convenient choice of α would be $\alpha = \frac{K(1+r)}{r}$ with which Equation ??? becomes

$$x'_t = (1+r)x'_{t-1}(1 - x'_{t-1}). \tag{5.5.8}$$

Furthermore, you can always define new parameters to make equations even simpler. Here, you can define a new parameter $r' = 1 + r$, with which you obtain the following final equation:

$$x'_t = r'x'_{t-1}(1 - x'_{t-1}) \tag{5.5.9}$$

Note that this is not the only way of rescaling variables; there are other ways to simplify the model. Nonetheless, you might be surprised to see how simple the model can become. The dynamics of the rescaled model are still exactly the same as before, i.e., the original model and the rescaled model have the same mathematical properties. We can learn a few more things from this result.

- First, Equation ??? tells us what is the meaningful unit for you to use in measuring the population in this context.
- Second, even though the original model appeared to have two parameters (r and K), this model essentially has only one parameter, r', so exploring values of r' should give you all possible dynamics of the model (i.e., there is no need to explore in the $r - K$ parameter space).

In general, if your model has k variables, you may be able to eliminate up to k parameters from the model by variable rescaling (but not always). By the way, this simplified version of the logistic growth model obtained above,

$$x_t = rx_{t-1}(1 - x_{t-1}), \tag{5.5.10}$$

has a designated name; it is called the *logistic map* and is arguably the most extensively studied 1-D nonlinear iterative map. This will be discussed in more detail in Chapter 8. Here is a summary of variable rescaling:

How-TO: Rescaling

You should try variable rescaling to eliminate as many parameters as possible from your model before conducting a mathematical analysis. You may be able to eliminate as many parameters as the variables in your model. To rescale variables, do the following:

1. Replace all variables with a non-zero constant times a new variable.
2. Simplify the model equations.
3. Find "convenient" choices for the constants that will make your equations as simple as possible.
4. Define new parameters, as needed, to make the equations even simpler.

Exercise 5.5.1

Simplify the following difference equation by variable rescaling:

$$x_t = \frac{a}{x_{t-1} + b} \tag{5.5.11}$$

Exercise 5.5.2

Simplify the following two-dimensional predator-prey difference equation model by variable rescaling:

$$x_t = x_{t-1} + r x_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) - \left(1 - \frac{1}{b y_{t-1} + 1}\right) x_{t-1} \tag{5.5.12}$$

$$y_t = y_{t-1} - d y_{t-1} + c x_{t-1} y_{t-1} \tag{5.5.13}$$

This page titled 5.5: Variable Rescaling of Discrete-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems

One of the main objectives of rule-based modeling is to make predictions of the future. So, it is a natural question to ask where the system will eventually go in the (infinite) long run. This is called the *asymptotic behavior* of the system when time is taken to infinity, which turns out to be fully predictable if the system is linear.

Within the scope of discrete time models, *linear dynamical systems* are systems whose dynamics can be described as:

$$x_t = Ax_{t-1}, \tag{5.6.1}$$

where x is the state vector of the system and A is the coefficient matrix. Technically, you could also add a constant vector to the right hand side, such as

$$x_t = Ax_{t-1} + a, \tag{5.6.2}$$

but this can always be converted into a constant-free form by adding one more dimension, i.e.,

$$y_t = \begin{pmatrix} x_t \\ 1 \end{pmatrix} = \begin{pmatrix} A & a \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ 1 \end{pmatrix} = By_{t-1} \tag{5.6.3}$$

Therefore, we can be assured that the constant-free form of Equati)on ??? covers all possible behaviors of linear difference equations.

Obviously, Equation 5.6.1 has the following closed-form solution:

$$x_t = A^t x_0 \tag{5.6.4}$$

This is simply because A is multiplied to the state vector x from the left at every time step. Now the key question is this: How will Eq. (5.1.3) behave when $t \to \infty$? In studying this, the exponential function of the matrix, A_t, is a nuisance. We need to turn it into a more tractable form in order to understand what will happen to this system as t gets bigger. And this is where *eigenvalues* and *eigenvectors* of the matrix A come to play a very important role. Just to recap, eigenvalues λ_i and eigenvectors v_i of A are the scalars and vectors that satisfy

$$Av_i = \lambda_i v_1. \tag{5.6.5}$$

In other words, throwing at a matrix one of its eigenvectors will "destroy the matrix" and turn it into a mere scalar number, which is the eigenvalue that corresponds to the eigenvector used. If we repeatedly apply this "matrix neutralization" technique, we get

$$A^t v_i = A^{t-1}\lambda_i v_i = A^{t-2}\lambda_i^2 v_i = \ldots = \lambda_i^t v_i \tag{5.6.6}$$

This looks promising. Now,we just need to apply the above simplification to Equation 5.6.4. To do so, we need to represent the initial state x_0 by using A's eigenvectors as the basis vectors, i.e.,

$$x_0 = b_1 v_1 + b_2 v_2 + \ldots + b_n v_n, \tag{5.6.7}$$

where n is the dimension of the state space (i.e., A is an $n \times n$ matrix). Most real-world $n \times n$ matrices are diagonalizable and thus have n linearly independent eigenvectors, so here we assume that we can use them as the basis vectors to represent any initial state x_0[2]. If you replace x_0 with this new notation in Equation 5.6.4, we get the following:

$$x_t = A^t(b_1 v_1 + b_2 v_2 + \ldots + b_n v_n) \tag{5.6.8}$$

$$= b_1 A^t v_1 + b_2 A^t v_2 + \ldots + b_n A^t v_n \tag{5.6.9}$$

$$= b_1 \lambda_1^t v_1 + b_2 \lambda_2^t v_2 + \ldots + b_n \lambda_n^t v_n \tag{5.6.10}$$

This result clearly shows that the asymptotic behavior of xt is given by a summation of multiple exponential terms of λ_i. There are competitions among those exponential terms, and which term will eventually dominate the others is determined by the absolute value of λ_i. For example, if λ_1. has the largest absolute value ($|\lambda_1| > |\lambda_2|, |\lambda_3|, \ldots |\lambda_n|$), then

$$x_t = \lambda_1^t (b_1 v_1 + b_2 (\frac{\lambda_2}{\lambda_1})^t v_2 + \ldots + b_n (\frac{\lambda_n}{\lambda_1})^t v_n \tag{5.6.11}$$

$$\lim_{t \to \infty} \approx \lambda_1^t b_1 v_1. \tag{5.6.12}$$

This eigenvalue with the largest absolute value is called a ***dominant eigenvalue***, and its corresponding eigenvector is called a dominant eigenvector, which will dictate which direction (a.k.a. *mode* in physics) the system's state will be going asymptotically. Here is an important fact about linear dynamical systems:

> If the coefficient matrix of a linear dynamical system has just one dominant eigenvalue, then the state of the system will asymptotically point to the direction given by its corresponding eigenvector *regardless of the initial state.*

This can be considered a very simple, trivial, linear version of *self-organization*. Let's look at an example to better understand this concept. Consider the asymptotic behavior of the Fibonacci sequence:

$$x_t = x_{t-1} + x_{t-2} \tag{5.6.13}$$

As we studied already, this equation can be turned into the following two-dimensional first-order model:

$$x_t = x_{t-1} + y_{t-1} \tag{5.6.14}$$

$$y_t = x_{t-1} \tag{5.6.15}$$

This can be rewritten by letting $(x_t, y_t) \Rightarrow x_t$ and using a vector-matrix notation, as:

$$x_t = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} x_{t-1} \tag{5.6.16}$$

So, we just need to calculate the eigenvalues and eigenvectors of the above coefficient matrix to understand the asymptotic behavior of this system. Eigenvalues of a matrix A can be obtained by solving the following equation for λ:

$$\det \begin{pmatrix} 1-\lambda & 1 \\ 1 & -\lambda \end{pmatrix} = -(1-\lambda)\lambda - 1 = \lambda^2 - \lambda - 1 = 0, \tag{5.6.17}$$

which gives

$$\lambda = \frac{1 \pm \sqrt{5}}{2} \tag{5.6.18}$$

as its solutions. Note that one of them ($(1+\sqrt{5})/2 = 1.618\ldots$) is the *golden ratio*! It is interesting that the golden ratio appears from such a simple dynamical system. Of course, you can also use Python to calculate the eigenvalues and eigenvectors (or, to be more precise, their approximated values). Do the following:

Code 5.6:

```
from pylab import *
eig([[1, 1], [1, 0]])
```

The `eig` function is there to calculate eigenvalues and eigenvectors of a square matrix. You immediately get the following results:

Code 5.7:

```
(array([ 1.61803399, -0.61803399]), array([[ 0.85065081, -0.52573111],
       [ 0.52573111,  0.85065081]]))
```

The first `array` shows the list of eigenvalues (it surely includes the golden ratio), while the second one shows the eigenvector matrix (i.e., a square matrix whose column vectors are eigenvectors of the original matrix). The eigenvectors are listed in the same order as eigenvalues. Now we need to interpret this result. The eigenvalues are 1.61803399 and -0.61803399. Which one is dominant? The answer is the first one, because its absolute value is greater than the second one's. This means that, asymptotically, the system's behavior looks like this:

$$x_t \approx 1.61803399 x_{t-1} \tag{5.6.19}$$

Namely, the dominant eigenvalue tells us the asymptotic ratio of magnitudes of the state vectors between two consecutive time points (in this case, it approaches the golden ratio). If the absolute value of the dominant eigenvalue is greater than 1, then the system will diverge to infinity, i.e., *the system is unstable*. If less than 1, the system will eventually shrink to zero, i.e., *the system is stable*. If it is precisely 1, then the dominant eigenvector component of the system's state will be conserved with neither divergence nor convergence, and thus the system may converge to a non-zero equilibrium point. The same interpretation can be applied to non-dominant eigenvalues as well.

An eigenvalue tells us whether a particular component of a system's state (given by its corresponding eigenvector) grows or shrinks over time. For discrete-time models:

- $|\lambda| > 1$ means that the component is growing.
- $|\lambda| < 1$ means that the component is shrinking.
- $|\lambda| = 1$ means that the component is conserved

For discrete-time models, *the absolute value of the dominant eigenvalue* λ_d determines the stability of the whole system as follows:

- $|\lambda_d| > 1$: The system is *unstable*, diverging to infinity.
- $|\lambda_d| < 1$: The system is *stable*, converging to the origin.
- $|\lambda_d| = 1$: The system is *stable*, but the dominant eigenvector component is conserved, and therefore the system may converge to a non-zero equilibrium point

We can now look at the dominant eigenvector that corresponds to the dominant eigenvalue, which is (0.85065081, 0.52573111). This eigenvector tells you the asymptotic direction of the system's state. That is, after a long period of time, the system's state (x_t, y_t) will be proportional to (0.85065081, 0.52573111), *regardless of its initial state*. Let's confirm this analytical result with computer simulations.

? Exercise 5.6.1

Visualize the phase space of Equation 5.6.16.

The results are shown in Fig. 5.6.1, for 3, 6, and 9 steps of simulation. As you can see in the figure, the system's trajectories asymptotically diverge toward the direction given by the dominant eigenvector (0.85065081, 0.52573111), as predicted in the analysis above.

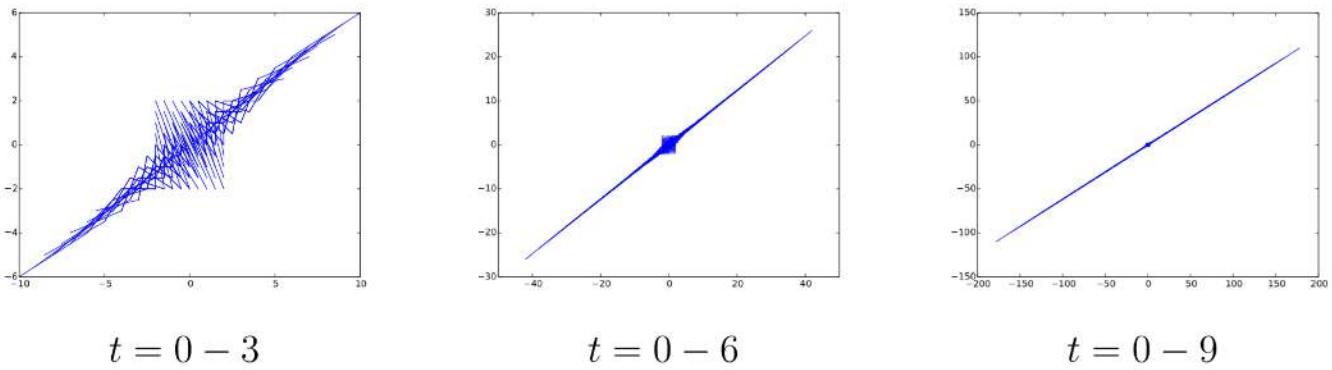


Figure 5.6.1: Phase space visualizations of Equation 5.6.16 for three different simulation lengths.

Figure 5.6.2 illustrates the relationships among the eigenvalues, eigenvectors, and the phase space of a discrete-time dynamical system. The two eigenvectors show the directions of two *invariant lines* in the phase space (shown in red). Any state on each of those lines will be mapped onto the same line. There is also an eigenvalue associated with each line (λ_1 and λ_2 in the figure). If its absolute value is greater than 1, the corresponding eigenvector component of the system's state is growing exponentially ((λ_1, (v_1), whereas if it is less than 1, the component is shrinking exponentially ((λ_2, v_2). In addition, for discrete-time models, if the eigenvalue is negative, the corresponding eigenvector component alternates its sign with regard to the origin every time the system's state is updated (which is the case for (λ_2, v_2 in the figure).

Here is a summary perspective for you to understand the dynamics of linear systems:

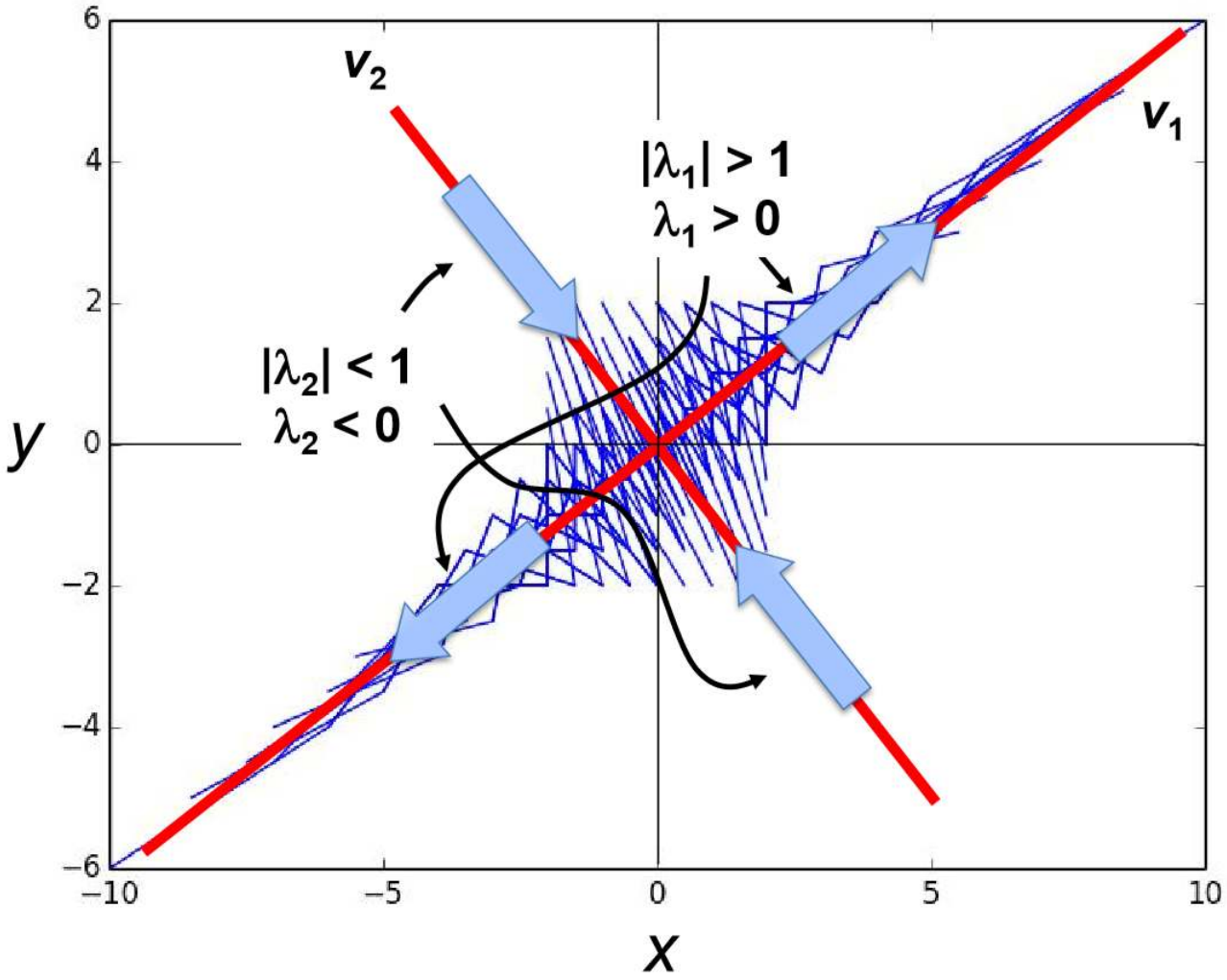


Figure **5.6.2**: Relationships among eigenvalues, eigenvectors, and the phase space of a discrete-time dynamical system.

> Dynamics of a linear system are decomposable into multiple independent one-dimensional exponential dynamics, each of which takes place along the direction given by an eigenvector. A general trajectory from an arbitrary initial condition can be obtained by a simple linear superposition of those independent dynamics.

One more thing. Sometimes you may find some eigenvalues of a coefficient matrix to be complex conjugate, not real. This may happen only if the matrix is asymmetric (i.e., symmetric matrices always have only real eigenvalues). If this happens, the eigenvectors also take complex values, which means that there are no invariant lines in the phase space. So, what is happening there? The answer is *rotation*. As you remember, linear systems can show oscillatory behaviors, which are rotations in their phase space. In such cases, their coefficient matrices have complex conjugate eigenvalues. The meaning of the absolute values of those

complex eigenvalues is still the same as before—greater than 1 means instability, and less than 1 means stability. Here is a summary:

> If a linear system's coefficient matrix has complex conjugate eigenvalues, the system's state is rotating around the origin in its phase space. The absolute value of those complex conjugate eigenvalues still determines the stability of the system, as follows:
>
> - $|\lambda| > 1$ means rotation with an expanding amplitude.
> - $|\lambda| < 1$ means rotation with a shrinking amplitude.
> - $|\lambda| = 1$ means rotation with a sustained amplitude.

Here is an example of such rotating systems, with coefficients taken from Code 4.13.

Code 5.8:

```
from pylab import *
eig([[0.5, 1], [-0.5, 1]])
```

The result is this:

Code 5.9:

```
(array([ 0.75+0.66143783j, 0.75-0.66143783j]), array([[ 0.81649658+0.j,
     0.81649658-0.j], [ 0.20412415+0.54006172j, 0.20412415-0.54006172j]]))
```

Now we see the complex unit `j` (yes, Python uses `j` instead of `i` to represent the imaginary unit `i`) in the result, which means this system is showing oscillation. Moreover, you can calculate the absolute value of those eigenvalues:

Code 5.10:

```
map(abs, eig([[0.5, 1], [-0.5, 1]])[0])
```

Then the result is as follows:

Code 5.11:

```
[0.99999999999999989, 0.99999999999999989]
```

This means that $|\lambda|$ is essentially 1, indicating that the system shows sustained oscillation, as seen in Fig. 4.4.2. For higher-dimensional systems, various kinds of eigenvalues can appear in a mixed way; some of them may show exponential growth, some may show exponential decay, and some others may show rotation. This means that all of those behaviors are going on simultaneously and independently in the system. A list of all the eigenvalues is called the *eigenvalue spectrum* of the system (or just *spectrum* for short). The eigenvalue spectrum carries a lot of valuable information about the system's behavior, but often, the most important information is whether the system is stable or not, which can be obtained from the dominant eigenvalue.

? Exercise 5.6.1

Study the asymptotic behavior of the following three-dimensional difference equation model by calculating its eigenvalues and eigenvectors:

$$x_t = x_{t-1} - y_{t-1} \tag{5.6.20}$$

$$y_t = -x_{t-1} - 3y_{t-1} + z_{t-1} \tag{5.6.21}$$

$$z_t = y_{t-1} + z_{t-1} \tag{5.6.22}$$

? Exercise 5.6.2

Consider the dynamics of opinion diffusion among five people sitting in a ring-shaped structure. Each individual is connected to her two nearest neighbors (i.e., left and right). Initially they have random opinions (represented as random real numbers), but at every time step, each individual changes her opinion to the local average in her social neighborhood (i.e, her own opinion plus those of her two neighbors, divided by 3). Write down these dynamics as a linear difference equation with five variables, then study its asymptotic behavior by calculating its eigenvalues and eigenvectors.

? Exercise 5.6.3

What if a linear system has more than one dominant, real-valued eigenvalue? What does it imply for the relationship between the initial condition and the asymptotic behavior of the system?

[2]This assumption doesn't apply to *defective* (non-diagonalizable) matrices that don't have n linearly independent eigenvectors. However, such cases are rather rare in real-world applications, because any arbitrarily small perturbations added to a defective matrix would make it diagonalizable. Problems withs uch sensitive, ill-behaving properties are sometimes called *pathological* in mathematics and physics. For more details about matrix diagonalizability and other related issues, look at linear algebra textbooks, e.g. [28].

This page titled 5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

5.7: 5.7 Linear Stability Analysis of Discrete-Time Nonlinear Dynamical Systems

All of the discussions above about eigenvalues and eigenvectors are for linear dynamical systems. Can we apply the same methodology to study the asymptotic behavior of nonlinear systems? Unfortunately, the answer is a depressing *no*. Asymptotic behaviors of nonlinear systems can be very complex, and there is no general methodology to systematically analyze and predict them. We will revisit this issue later.

Having said that, we can still use eigenvalues and eigenvectors to conduct a *linear stability analysis* of nonlinear systems, which is an analytical method to determine the stability of the system at or near its equilibrium point by approximating its dynamics around that point as a linear dynamical system (*linearization*). While linear stability analysis doesn't tell much about a system's asymptotic behavior at large, it is still very useful for many practical applications, because people are often interested in how to sustain a system's state at or near a desired equilibrium, or perhaps how to disrupt the system's status quo to induce a fundamental change. The basic idea of linear stability analysis is to rewrite the dynamics of the system in terms of a *small perturbation* added to the equilibrium point of your interest. Here I put an emphasis onto the word "*small*" for a reason. When we say small perturbation in this context, we mean not just small but *really, really small* (infinitesimally small in mathematical terms), so small that we can safely ignore its square or any higher-order terms. This operation is what linearization is all about. Here is how linear stability analysis works. Let's consider the dynamics of a nonlinear difference equation

$$x_t = F(x_{t-1}) \tag{5.7.1}$$

around its equilibrium point x_{eq}. By definition, x_{eq} satisfies

$$x_{eq} = F(x_{eq}). \tag{5.7.2}$$

To analyze the stability of the system around this equilibrium point, we switch our perspective from a global coordinate system to a local one, by zooming in and capturing a small perturbation added to the equilibrium point, $\Delta x_t = x_t - x_{eq}$. Specifically, we apply the following replacement

$$\Delta x_t \Rightarrow x_{eq} - x_t \tag{5.7.3}$$

to Equation 5.7.1, to obtain

$$x_{eq} + \Delta x_t = F(x_{eq} + \Delta x_{t-1}). \tag{5.7.4}$$

The right hand side of the equation above is still a nonlinear function. If x_t is scalar and thus $F(x)$ is a scalar function, the right hand side can be easily approximated using the Taylor expansion as follows:

$$F(x_{eq} + \Delta x_{t-1}) \tag{5.7.5}$$

$$= F(x_{eq}) + F'(x_{eq})\Delta x_{t-1} + \frac{F''(x_{eq})}{2!} Deltax_{t-1}^2 + \frac{F'''(x_{eq})}{3!} Deltax_{t-1}^3 + \vdots +$$

$$\approx F(x_{eq}) + F'(x_{eq})\Delta x_{t-1} \tag{5.7.6}$$

This means that, for a scalar function F, $F_(x_{eq} + \Delta x)$ can be linearly approximated by the value of the function at x_{eq} plus a derivative of the function times the displacement from x_{eq}. Together with this result and Equation 5.7.2, Equation 5.7.4 becomes the following very simple linear difference equation:

$$\Delta x_t \approx F' x_{eq} \Delta x_{t-1} \tag{5.7.7}$$

This means that, if $|F_0(x_{eq})| > 1$, Δ_x grows exponentially, and thus the equilibrium point x_{eq} is unstable. Or if $|F_0(x_{eq})| < 1$, Δ_x shrinks exponentially, and thus x_{eq} is stable. Interestingly, this conclusion has some connection to the cobweb plot we discussed before. $|F_0(x_{eq})|$ is the slope of function F at an equilibrium point (where the function curve crosses the diagonal straight line in the cobweb plot). If the slope is too steep, either positively or negatively, trajectories will diverge away from the equilibrium point. If the slope is less steep than 1, trajectories will converge to the point. You may have noticed such characteristics when you drew the cobweb plots. Linear stability analysis offers a mathematical explanation of that.

Now, what if F is a multidimensional nonlinear function? Such an F can be spelled out as a set of multiple scalar functions, as follows:

$$x_{1,t} = F_1(x_1, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.7.8}$$

$$x_{2,t} = F_2(x_1, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.7.9}$$

$$\vdots$$

$$x_{n,t} = F_n(x_1, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.7.10}$$

Using variable replacement similar to Equation 5.7.3, these equations are rewritten as follows:

$$x_{1,eq} + \Delta x_{1,t} = F_1(x_{1,eq} + \Delta x_{1,t-1}, x_{2,eq} + \Delta x_{2,t-1} \ldots, x_{n,eq} + \Delta x_{n,t-1} \tag{5.7.11}$$

$$x_{2,eq} + \Delta x_{2,t} = F_2(x_{1,eq} + \Delta x_{1,t-1}, x_{2,eq} + \Delta x_{2,t-1} \ldots, x_{n,eq} + \Delta x_{n,t-1} \tag{5.7.12}$$

$$\vdots$$

$$x_{n,eq} + \Delta x_{2,t} = F_2(x_{1,eq} + \Delta x_{1,t-1}, x_{2,eq} + \Delta x_{2,t-1} \ldots, x_{n,eq} + \Delta x_{n,t-1} \tag{5.7.13}$$

Since there are many Δx_i's in this formula, the Taylor expansion might not apply simply. However, the assumption that they are extremely small helps simplify the analysis here. By zooming into an infinitesimally small area near the equilibrium point, each F_i looks like a completely flat "plane" in a multidimensional space (Fig. 5.7.1) where all nonlinear interactions among Δx_i's are negligible. This means that the value of F_i can be approximated by a simple linear sum of independent contributions coming from the n dimensions, each of which can be calculated in a manner similar to Eq. 5.7.6, as

$$\begin{aligned} &F_i(x_{1,eq} + \Delta x_{1,t-1}, x_{2,eq} + \Delta x_{2,t-1} \ldots, x_{n,eq} + \Delta x_{n,t-1} \\ &\approx F_i(x_{eq} + \frac{\partial F_i}{\partial x_1}|_{eq} \Delta x_{1,t-1} + \frac{\partial F_i}{\partial x_2}|_{eq} \Delta x_{2,t-1} + \ldots + \frac{\partial F_i}{\partial x_2}|_{eq} \Delta x_{n,t-1}. \end{aligned} \tag{5.7.14}$$

This linear approximation allows us to rewrite Eqs. 5.7.11, 5.7.12 5.7.13 into the following, very concise linear equation:

$$x_{eq} + \Delta x_t \approx F(x_{eq}) + \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_1} & \cdot & \cdot & \cdot & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdot & \cdot & \cdot & \frac{\partial F_2}{\partial x_n} \\ & \cdot & \cdot & \cdot & & \\ & \cdot & \cdot & \cdot & & \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdot & \cdot & \cdot & \frac{\partial F_n}{\partial x_n} \end{pmatrix} |_{x=x_{eq}} \Delta x_{t-1} \tag{5.7.15}$$

The coefficient matrix filled with partial derivatives is called a *Jacobian matrix* of the original multidimensional function F. It is a linear approximation of the nonlinear function

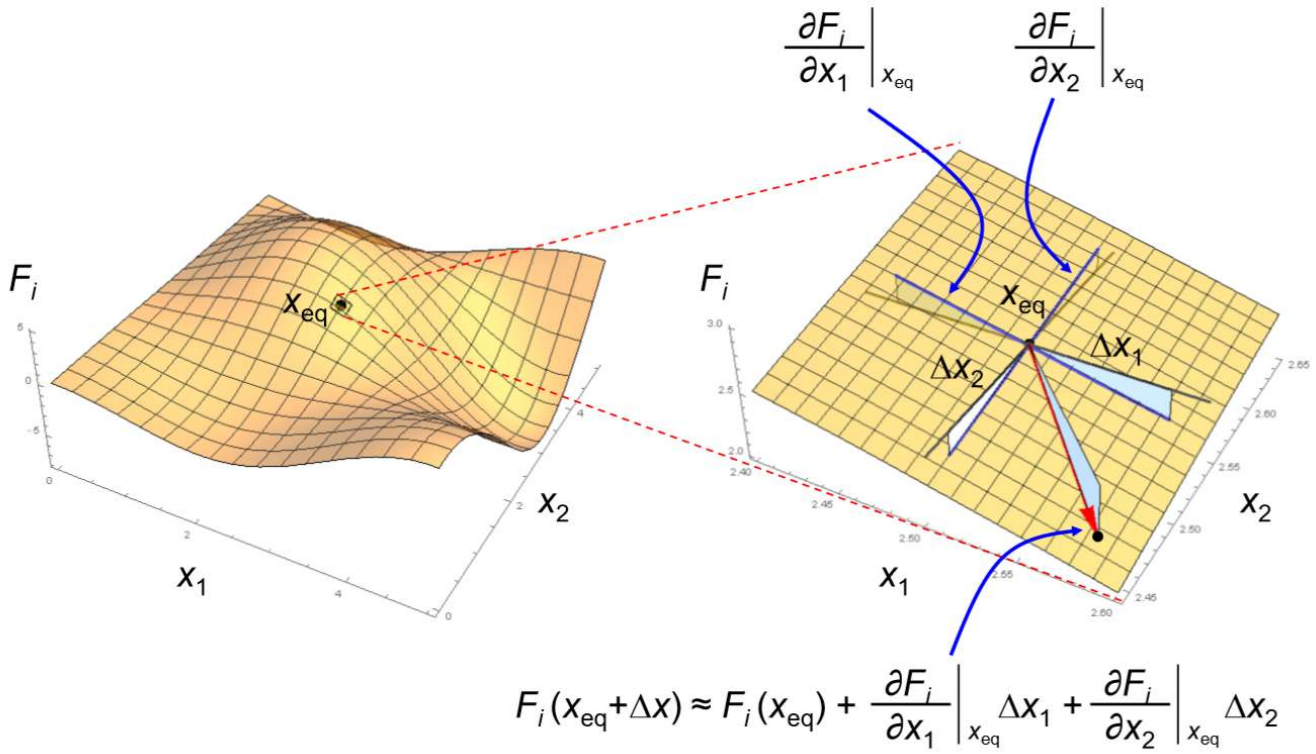


Figure **5.7.1**: Linearization of a multidimensional nonlinear function around an equilibrium point.

around $\boldsymbol{x = x_{eq}}$, just like a regular derivative of a scalar function. Note that the orders of rows and columns of a Jacobian matrix must match. Its $\boldsymbol{i}$-th row must be a list of spatial derivatives of $\boldsymbol{F_i}$, i.e., a function that determines the behavior of $\boldsymbol{x_i}$, while $\boldsymbol{x_i}$ must be used to differentiate functions for the $\boldsymbol{i}$-th column.

By combining the result above with Equation 5.7.2, we obtain

$$\boldsymbol{\Delta x_t \approx J \Delta x_{t-1},} \tag{5.7.16}$$

where $\boldsymbol{J}$ is the Jacobian matrix of $\boldsymbol{F}$ at $\boldsymbol{x = x_{eq}}$. Look at how simple it can get! The dynamics are approximated in a very simple linear form, which describes the behavior of the small perturbations around $\boldsymbol{x_{eq}}$ as the new origin.

Now we can calculate the eigenvalues of J to see if this system is stable or not, around $\boldsymbol{x_{eq}}$. If the absolute value of the dominant eigenvalue $\boldsymbol{\lambda_d}$is less than 1,the equilibrium point is stable; even if a small perturbation is added to the system's state, it asymptotically goes back to the equilibrium point. If$\boldsymbol{|\lambda d| > 1}$, the equilibrium point is unstable; any small perturbation added to the system's state grows exponentially and, eventually, the system's state moves away from the equilibrium point. Sometimes, an unstable equilibrium point may come with other eigenvalues that show stability. Such equilibrium points are called *saddle points*, where nearby trajectories are attracted to the equilibrium point in some directions but are repelled in other directions. If$\boldsymbol{|\lambda d| = 1}$, it indicates that the system may be neutral (also called *Lyapunov stable*), which means that the system's state neither diverges away from nor converges to the equilibrium point. But actually, proving that the point is truly neutral requires more advanced nonlinear analysis, which is beyond the scope of this textbook. Finally, if the eigenvalues are complex conjugates, oscillatory dynamics are going on around the equilibrium points. Such equilibrium points are called a stable or unstable *spiral focus* or a *neutral center*, depending on their stabilities. Figure 5.7.2 shows a schematic summary of these classifications of equilibrium points for two-dimensional cases.

Linear stability analysis of discrete-time nonlinear systems

1. Find an equilibrium point of the system you are interested in.
2. Calculate the Jacobian matrix of the system at the equilibrium point.

3. Calculate the eigenvalues of the Jacobian matrix.
4. If the absolute value of the dominant eigenvalue is:

• Greater than 1⇒The equilibrium point is unstable.

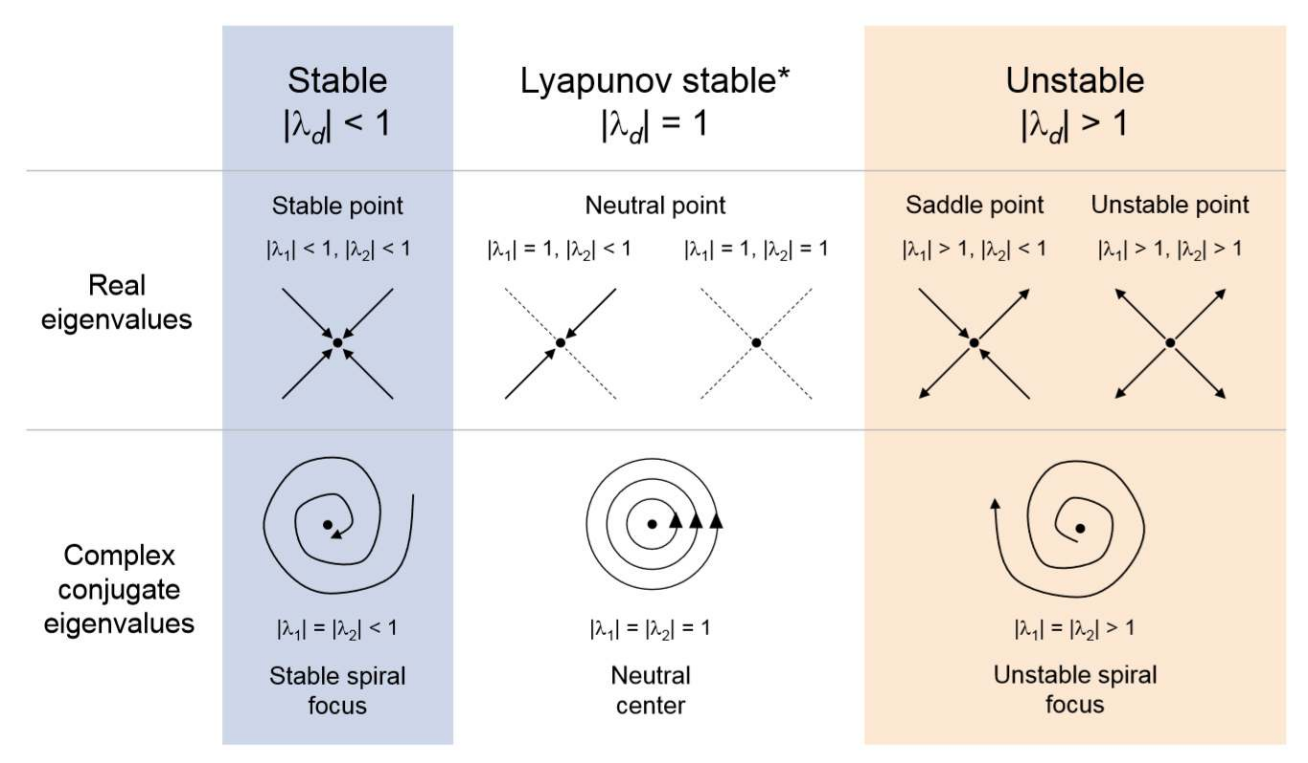


Figure 5.7.2: Schematic illustrations of several different types of equilibrium points and their nearby trajectories in 2-D discrete-time dynamical systems. The absolute value of the dominant eigenvalue $|\lambda_d|$ determines the overall stability of the equilibrium point (top), although it would generally require a nonlinear analysis to show that the equilibrium point is truly neutral/Lyapunov stable (*).

– If other eigenvalues have absolute values less than 1, the equilibrium point is a saddle point.

• Less than 1⇒The equilibrium point is stable.

• Equal to 1⇒The equilibrium point may be neutral (Lyapunov stable).

5. In addition, if there are complex conjugate eigenvalues involved, oscillatory dynamics are going on around the equilibrium point. If those complex conjugate eigenvalues are the dominant ones, the equilibrium point is called a stable or unstable *spiral focus* (or a *neutral center* if the point is neutral).

? Exercise 5.7.1

Consider the following iterative map $(a > 0, b > 0)$

$$x_t = x_{t-1} + a\sin(bx_{t-1}) \tag{5.7.17}$$

Conduct linear stability analysis to determine whether this model is stable or not at its equilibrium point $x_{eq} = 0$.

? Exercise 5.7.2

Consider the following two-dimensional difference equation model:

$$x_t = x_{t-1} + 2x_{t-1}(1 - x_{t-1}) - x_{t-1}y_{t-1} \tag{5.7.18}$$

$$y_t = y_{t-1} + 2y_{t-1}(1 - y_{t-1}) - x_{t-1}y_{t-1} \tag{5.7.19}$$

1. Find all of its equilibrium points.

2. Calculate the Jacobian matrix at the equilibrium point where $x > 0$ and $y > 0$.

3. Calculate the eigenvalues of the matrix obtained above.

4. Based on the result, classify the equilibrium point into one of the following: stable point, unstable point, saddle point, stable spiral focus, unstable spiral focus, or neutral center.

? Exercise 5.7.3

Consider the following two-dimensional difference equation model:

$$x_t = x_{t-1}y_{t-1} \tag{5.7.20}$$

$$y_t = y_{t-1}(x_{t-1} - 1) \tag{5.7.21}$$

1. Find all equilibrium points(which you may have done already in Exercise 5.7.2).

2. Calculate the Jacobian matrix at each of the equilibrium points.

3. Calculate the eigenvalues of each of the matrices obtained above.

4. Based on the results, discuss the stability of each equilibrium point.

This page titled 5.7: 5.7 Linear Stability Analysis of Discrete-Time Nonlinear Dynamical Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

6: Continuous-Time Models I - Modeling

6.1: Continuous-Time Models with Differential Equations

6.2: Classifications of Model Equation

6.3: Connecting Continuous - Time Models with DiscreteTime Models

6.4: Simulating Continuous-Time Models

6.5: Building Your Own Model Equation

This page titled 6: Continuous-Time Models I - Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

6.1: Continuous-Time Models with Differential Equations

Continuous-time models are written in *differential equations*. They are probably more mainstream in science and engineering, and studied more extensively, than discrete-time models, because various natural phenomena (e.g., motion of objects, flow of electric current) take place smoothly over continuous time. A general mathematical formulation of a first-order continuous-time model is given by this:

$$\frac{dx}{dt} = F(x, t) \tag{6.1.1}$$

Just like in discrete-time models, x is the state of a system (which may be a scalar or vector variable). The left hand side is the time derivative of x, which is formally defined as

$$\frac{dx}{dt} = \lim_{\delta t \to 0} \frac{x(t + \delta t) - x(t)}{\delta t}. \tag{6.1.2}$$

Integrating a continuous-time model over t gives a trajectory of the system's state over time. While integration could be done algebraically in some cases, computational simulation (= numerical integration) is always possible in general and often used as the primary means of studying these models. One fundamental assumption made in continuous-time models is that the trajectories of the system's state are smooth everywhere in the phase space, i.e., the limit in the definition above always converges to a well-defined value. Therefore, continuous-time models don't show instantaneous abrupt changes, which could happen in discrete-time models.

This page titled 6.1: Continuous-Time Models with Differential Equations is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

6.2: Classifications of Model Equation

Distinctions between linear and nonlinear systems as well as autonomous and non-autonomous systems, which we discussed in Section 4.2, still apply to continuous-time models. But the distinction between first-order and higher-order systems are slightly different, as follows.

First-order system

A differential equation that involves first-order derivatives of state variables ($\frac{dx}{dt}$) only.

Higher-order system

A differential equation that involves higher-order derivatives of state variables ($\frac{d^2x}{dt^2}$, $\frac{d^3x}{dt^3}$, etc.).

Luckily, the following is still the case for continuous-time models as well:

continuous-time models

Non-autonomous, higher-order differential equations can always be converted into autonomous, first-order forms by introducing additional state variables.

Here is an example:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta \tag{6.2.1}$$

This equation describes the swinging motion of a simple pendulum, which you might have seen in an intro to physics course. θ is the angular position of the pendulum, g is the gravitational acceleration, and L is the length of the string that ties the weight to the pivot. This equation is obviously nonlinear and second-order. While we can't remove the nonlinearity from the model, we can convert the equation to a first-order form, by introducing the following additional variable:

$$\omega = \frac{d\theta}{dt} \tag{6.2.2}$$

Using this, the left hand side of Equation 6.2.1 can be written as $\frac{d\omega}{dt}$, and therefore, the equation can be turned into the following first-order form:

$$\frac{d\theta}{dt} = \omega \tag{6.2.3}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta \tag{6.2.4}$$

This conversion technique works for third-order or any higher-order equations as well, as long as the highest order remains finite. Here is another example.

Example 6.2.1: driven pendulum

Consider the non-autonomous equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta + k\sin(2\pi ft + \phi) \tag{6.2.5}$$

This is a differential equation of the behavior of a *driven pendulum*. The second term on the right hand side represents a periodically varying force applied to the pendulum by, e.g., an externally controlled electromagnet embedded in the floor. As we discussed before, this equation can be converted to the following first-order form:

Multiple \label

Now we need to eliminate t inside the $\sin$ function. Just like we did for the discrete-time cases, we can introduce a "clock" variable, say τ, as follows:

$$\frac{d\tau}{dt} = 1,\ \tau(0) = 0 \tag{6.2.6}$$

This definition guarantees $\tau(t) = t$. Using this, the full model can be rewritten as follows:

Multiple \label

This is now made of just autonomous, first-order differential equations.

This conversion technique always works, assuring us that autonomous, first-order equations can cover all the dynamics of any non-autonomous, higher-order equations.

? Exercise 6.2.1

Convert the following differential equation into first-order form.

$$\frac{d^2}{dt^2} - x\frac{dx}{dt} + x^2 = 0 \tag{6.2.7}$$

? Exercise 6.2.2

Convert the following differential equation into an autonomous, first order form.

$$\frac{d^2x}{dt^2} - a\cos bt = 0 \tag{6.2.8}$$

For your information, the following facts are also applicable to differential equations, as well as to difference equations:

Linear dynamical systems can show only exponential growth/decay, periodic oscillation, stationary states (no change), or their hybrids (e.g., exponentially growing oscillation)[a].

[a]sometimes they can also show behaviors that are represented by polynomials (or products of polynomials and exponentials) of time. This occurs when their coefficient matrices are *non-diagonalizable*.

Linear equations are always analytically solvable, while nonlinear equations don't have analytical solutions in general.

This page titled 6.2: Classifications of Model Equation is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

6.3: Connecting Continuous - Time Models with DiscreteTime Models

Continuous-time models and discrete-time models are different mathematical models with different mathematical properties. But it is still possible to develop a "similar" continuous-time model from a discrete-time model, and vice versa. Here we discuss how you can jump across the border back and forth between the two time treatments.

Assume you already have an autonomous first-order discrete-time model

$$x_t = F(x_{t-1}), \tag{6.3.1}$$

and you want to develop a continuous-time model analogous to it. You set up the following "container" differential equation

$$\frac{dx}{dt} = G(x), \tag{6.3.2}$$

and try to find out how F and G are related to each other.

Here, let me introduce a very simple yet useful analogy between continuous- and discrete-time models:

$$\frac{dx}{dt} \approx \frac{\Delta x}{\Delta t} \tag{6.3.3}$$

This may look almost tautological. But the left hand side is a ratio between two infinitesimally small quantities, while the right hand side is a ratio between two quantities that are small yet have definite non-zero sizes. Δx is the difference between $x(t+\Delta t)$ and $x(t)$, and Δt is the finite time interval between two consecutive discrete time points. Using this analogy, you can rewrite Equation 6.3.2 as

$$\frac{\Delta x}{\Delta t} = \frac{x(t+\Delta t) - x(t)}{\Delta t} \approx G(x(t)), \tag{6.3.4}$$

$$x(t+\Delta t) \approx x(t) + G(x(t))\Delta t. \tag{6.3.5}$$

By comparing this with Eq.(6.16), we notice the following analogous relationship between F and G:

$$F(x) \leftrightarrow \frac{F(x) - x}{\Delta t} \tag{6.3.6}$$

For linear systems in particular, $F(x)$ and $G(x)$ are just the product of a coefficient matrix and a state vector. If $F(x) = Ax$ and $G(x) = Bx$, then the analogous relationships become

$$Ax \leftrightarrow x + Bx\Delta t, \tag{6.3.7}$$

$$A \leftrightarrow I + B\Delta t \tag{6.3.8}$$

or

$$B \leftrightarrow \frac{A - I}{\Delta t}. \tag{6.3.9}$$

I should emphasize that these analogous relationships between discrete-time and continuous-time models do not mean they are mathematically equivalent. They simply mean that the models are constructed according to similar assumptions and thus they may have similar properties. In fact, analogous models often share many identical mathematical properties, yet there are certain fundamental differences between them. For example, one- or two-dimensional discrete-time iterative maps can show chaotic behaviors, but their continuous-time counterparts never show chaos. We will discuss this issue in more detail later.

Nonetheless, knowing these analogous relationships between discrete-time and continuous-time models is helpful in several ways. First, they can provide convenient pathways when you develop your own mathematical models. Some natural phenomena may be conceived more easily as discrete-time, stepwise processes, while others may be better conceived as continuous-time, smooth processes. You can start building your model in either way, and when needed, convert the model from discrete-time to continuous-time or vice versa. Second, Equation 6.3.5 offers a simple method to numerically simulate continuous-time models. While this method is rather crude and prone to accumulating numerical errors, the meaning of the formula is quite straight forward, and the implementation of simulation is very easy, so we will use this method in the following section. Third, the relationship between the

coefficient matrices given in Equation 6.3.9 is very helpful for understanding mathematical differences of stability criteria between discrete-time and continuous-time models. This will be detailed in the next chapter.

? Exercise 6.3.1

Consider the dynamics of a system made of three parts, A, B, and C. Each takes a real-valued state whose range is [−1,1]. The system behaves according to the following state transitions:

- A adopts B's current state as its next state.
- B adopts C's current state as its next state.
- C adopts the average of the current states of A and B as its next state.

First, create a discrete-time model of this system, and then convert it into a continuous-time model using Equation 6.3.9.

This page titled 6.3: Connecting Continuous - Time Models with DiscreteTime Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

6.4: Simulating Continuous-Time Models

Simulation of a continuous-time model is equivalent to the numerical integration of differential equations, which, by itself, is a major research area in applied mathematics and computational science with more than a century of history. There are a large number of methodologies developed for how to accomplish accurate, fast, efficient numerical integrations. It would easily take a few books and semesters to cover them, and this textbook is not intended to do that. Instead, here we focus on the simplest possible method for simulating a continuous time model, by using the following formula as an approximation of a differential equation

$$\frac{dx}{dt} = G(x); \tag{6.4.1}$$

$$x(t + \Delta t) = x(t) + G(x(t))\Delta t \tag{6.4.2}$$

This method is called the *Euler forward method*. Its basic idea is very straightforward; you just keep accumulating small increases/decreases of x that is expected from the local derivatives specified in the original differential equation. The sequence produced by this discretized formula will approach the true solution of the original differential equation at the $\lim_{\Delta t \to 0}$, although in practice, this method is less accurate for finite-sized Δt than other more sophisticated methods (see Exercise 6.6.3). Having said that, its intuitiveness and easiness of implementation have a merit, especially for those who are new to computer simulation. So let's stick to this method for now.

In nearly all aspects, simulation of continuous-time models using the Euler forward method is identical to the simulation of discrete-time models we discussed in Chapter 4. Probably the only technical difference is that we have Δt as a step size for time, which may not be 1, so we also need to keep track of the progress of time in addition to the progress of the state variables. Let's work on the following example to see how to implement the Euler forward method.

Here we consider simulating the following continuous-time logistic growth model for $0 \le t < 50$ in Python, with $x(0) = 0.1$, $r = 0.2$, $K = 1$ and $\Delta t = 0.01$:

$$\frac{dx}{dt} = rx(1 - \frac{x}{K}) \tag{6.4.3}$$

The first (and only) thing we need to do is to discretize time in the model above. Using Equation 6.4.2, the equation becomes

$$x(t + \Delta t) = x(t) + rx(t)(1 - \frac{x(t)}{K})\Delta t, \tag{6.4.4}$$

which is nothing more than a typical difference equation. So, we can easily revise Code 4.10 to create a simulator of Equation 6.4.3:

Code 6.1: logisticgrowth-continuous.py

```
from pylab import *

r = 0.2
K = 1.0
Dt = 0.01
```

```
def initialize():
    global x, result, t, timesteps
    x = 0.1
    result = [x]
    t = 0.
    timesteps = [t]

def observe():
    global x, result, t, timesteps
    result.append(x)
    timesteps.append(t)

def update():
    global x, result, t, timesteps
    x = x + r * x * (1 - x / K) * Dt
    t = t + Dt

initialize()
while t < 50.:
    update()
    observe()

plot(timesteps, result)
show()
```

Figure 6.4.1. If you choose even smaller values for Δt, the curve will get closer to the true solution of Equation 6.4.3.

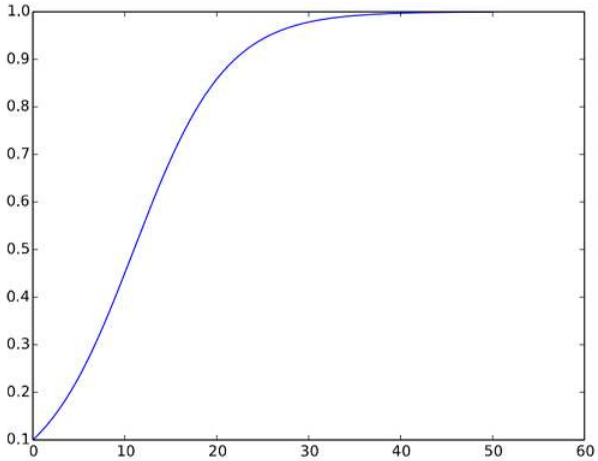


Figure 6.4.1: Visual output of Code 6.1.

? Exercise 6.4.1

Vary Δt to have larger values in the previous example and see how the simulation result is affected by such changes.

As the exercise above illustrates, numerical integration of differential equations involves some technical issues, such as the stability and convergence of solutions and the possibility of "artifacts" arising from discretization of time. You should always be attentive to these issues and be careful when implementing simulation codes for continuous-time models. I hope the following exercises will help you further investigate those subtleties of the numerical integration of differential equations.

? Exercise 6.4.2: Lotka-Volterra (predator prey) model

Simulate the following continuous-time Lotka-Volterra (predator prey) model for $0 \leq t < 50$ in Python, with $x(0) = y(0) = 0.1$, $a = b = c = d = 1$ and $\Delta t = 0.01$. Visualize the simulation results over time and also in a phase space.

$$\frac{dx}{dt} = ax - bxy \tag{6.4.5}$$

? Exercise 6.4.3: Other integration methods

There are many other more sophisticated methods for the numerical integration of differential equations, such as the backward Euler method, Heun's method, the Runge-Kutta methods,etc. Investigate some of those methods to see how they work and why their results are better than that of the Euler forward method.

This page titled 6.4: Simulating Continuous-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

6.5: Building Your Own Model Equation

Principles and best practices of building your own equations for a continuous-time model are very much the same as those we discussed for discrete-time models in Sections 4.5 and 4.6. The only difference is that, in differential equations, you need to describe time derivatives, i.e., instantaneous rates of change of the system's state variables, instead of their actual values in the next time step.

Here are some modeling exercises. The first one is exactly the same as the one in Section 4.6, except that you are now writing the model in differential equations, so that you can see the differences between the two kinds of models. The other two are on new topics, which are relevant to chemistry and social sciences. Work on these exercises to get some experience writing continuous-time models!

? Exercise 6.5.1

Develop a continuous-time mathematical model of two species competing for the same resource, and simulate its behavior.

? Exercise 6.5.2

Imagine two chemical species, S and E, interacting in a test tube. Assume that E catalyzes the production of itself using S as a substrate in the following chemical reaction:

$$S + E \rightarrow 2E \tag{6.5.1}$$

Develop a continuous-time mathematical model that describes the temporal changes of the concentration of S and E and simulate its behavior.

? Exercise 6.5.3

When a new pop song is released, it sounds attractive to people and its popularity increases. After people get used to the song, however, it begins to sound boring to them, and its popularity goes down. Develop a continuous-time mathematical model that captures such rise and fall in a typical pop song'slife, and simulate its behavior.

This page titled 6.5: Building Your Own Model Equation is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

7: Continuous-Time Models II - Analysis

7.1: Finding Equilibrium Points

7.2: Phase Space Visualization

7.3: Variable Rescaling of Continuous-Time Models

7.4: Asymptotic Behavior of Continuous-Time Linear Dynamical Systems

7.5: Linear Stability Analysis of Nonlinear Dynamical Systems

---

This page titled 7: Continuous-Time Models II - Analysis is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

7.1: Finding Equilibrium Points

Finding equilibrium points of a continuous-time model $\frac{dx}{dt} = G(x)$ can be done in the same way as for a discrete-time model, i.e., by replacing all x's with x_{eq}'s (again, note that these could be vectors). This actually makes the left hand side zero, because x_{eq} is no longer a dynamical variable but just a static constant. Therefore, things come down to just solving the following equation

$$0 = G(x_{eq})$$

with regard to x_{eq}. For example, consider the following logistic growth model:

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right) \tag{7.1.1}$$

Replacing all the x's with x_{eq}'s, we obtain

$$0 = rx_{eq}\left(1 - \frac{x}{K}\right) \tag{7.1.2}$$

$$x_{eq} = 0, K \tag{7.1.3}$$

It turns out that the result is the same as that of its discrete-time counterpart(see Eq.(5.1.6)).

? Exercise 7.1.1

Find the equilibrium points of the following model:

$$\frac{dx}{dt} = x^2 - rx + 1 \tag{7.1.4}$$

? Exercise 7.1.2: Simple Pendulum

Find the equilibrium points of the following model of a simple pendulum:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta$$

? Exercise 7.1.3: Susceptible-Infected-Recovered model

The following model is called a *Susceptible-Infected-Recovered (SIR) model*, a mathematical model of epidemiological dynamics. S is the number of susceptible individuals, I is the number of infected ones, and R is the number of recovered ones. Find the equilibrium points of this model.

$$\frac{dS}{dt} = -aSI \tag{7.1.5}$$

$$\frac{dI}{dt} = aSI - bI \tag{7.1.6}$$

$$\frac{dR}{dt} = bI \tag{7.1.7}$$

This page titled 7.1: Finding Equilibrium Points is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

7.2: Phase Space Visualization

A phase space of a continuous-time model, once time is discretized, can be visualized in the exact same way as it was in Chapter 5, using Codes 5.1 or 5.2. This is perfectly fine. In the meantime, Python's `matplotlib` has a specialized function called `streamplot`, which is precisely designed for drawing phase spaces of continuous-time models. It works only for two-dimensional phase space visualizations, but its output is quite neat and sophisticated. Here is how you can use this function:

Code 7.1: phasespace-drawing-streamplot.py

```
from pylab import *

xvalues, yvalues = meshgrid(arange(0, 3, 0.1), arange(0, 3, 0.1))

xdot = xvalues - xvalues * yvalues
ydot = - yvalues + xvalues * yvalues
```

```
streamplot(xvalues, yvalues, xdot, ydot)

show()
```

The `streamplot` function takes four arguments. The first two (`xvalues` and `yvalues`) are discretized `x` -and `y` -values in a phase space, each of which is given as a two-dimensional array. The meshgrid function generates such `array` 's for this purpose. The 0.1 in arange determines the resolution of the space. The last two arguments (`xdot` and `ydot`) describe the values of dx/dt and dy/dt on each point. Each of `xdot` and `ydot` should also be given as a two-dimensional array, but since `xvalues` and `yvalues` are already in the array structure, you can conduct arithmetic operations directly on them, as shown in the code above. In this case, the model being visualized is the following simple predator-prey equations:

$$\frac{dx}{dt} = x - xy \tag{7.2.1}$$

$$\frac{dy}{dt} = -y + xy \tag{7.2.2}$$

The result is shown in Fig. 7.2.1. As you can see, the `streamplot` function automatically adjusts the density of the sample curves to be drawn so that the phase space structure is easily visible to the eye. It also adds arrow heads to the curves so we can understand which way the system's state is flowing.

One nice feature of a continuous-time model's phase space is that, since the model is described in continuous differential equations, their trajectories in the phase space are all smooth with no abrupt jump or intersections between each other. This makes their phase space structure generally more visible and understandable than those of discrete-time models.

? Exercise 7.2.1

Draw a phase space of the following differential equation (motion of a simple pendulum) in Python:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta \tag{7.2.3}$$

Moreover, such smoothness of continuous-time models allows us to analytically visualize and examine the structure of their phase space. A typical starting point to do so is to find the nullclines in a phase space. A nullcline is a set of points where at least one of the time derivatives of the state variables becomes zero. These nullclines serve

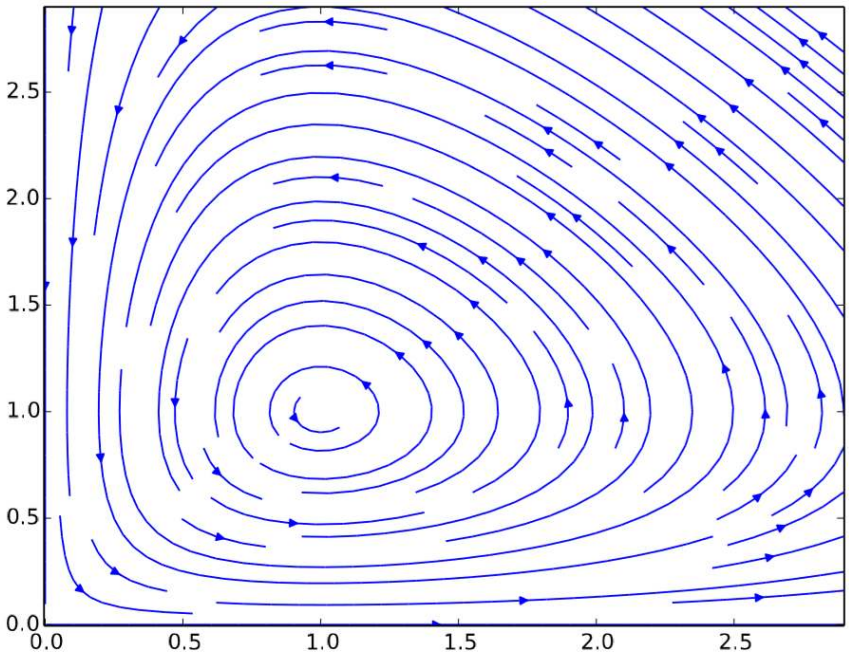


Figure 7.2.1: Phase space drawn with Code 7.1.

as “walls” that separate the phase space into multiple contiguous regions. Inside each region, the signs of the time derivatives never change (if they did, they would be caught in a nullcline), so just sampling one point in each region gives you a rough picture of how the phase space looks. Let’s learn how this analytical process works with the following **Lotka-Volterra model**:

$$\frac{dx}{dt} = ax - bxy \tag{7.2.4}$$

$$\frac{dy}{dt} = -cy + dxy \tag{7.2.5}$$

$$x \geq 0, y \geq 0, a > 0, b > 0, c > 0, d > 0 \tag{7.2.6}$$

First, find the nullclines. This is a two-dimensional system with two , so there must be two sets of nullclines; one set is derived from $\frac{dx}{dt} = 0$, and another set is derived from $\frac{dy}{dt} = 0$. They can be obtained by solving each of the following equations:

$$0 = ax - bxy \tag{7.2.7}$$

$$0 = -cy + dxy \tag{7.2.8}$$

The first equation gives

$$x = 0, or\ y = \frac{a}{b}. \tag{7.2.9}$$

These are two straight lines, which constitute one set of nullclines for \frac{dx}{dt} = 0\) (i.e., you could call each line a single nullcline). In the meantime, the second one gives

$$y = 0, or\ x = \frac{c}{d}. \tag{7.2.10}$$

Again, these two lines constitute another set of nullclines for $\frac{dy}{dt} = 0$. These results can be visualized manually as shown in Fig. 7.2.2. Equilibrium points exist where the two sets of nullclines intersect.

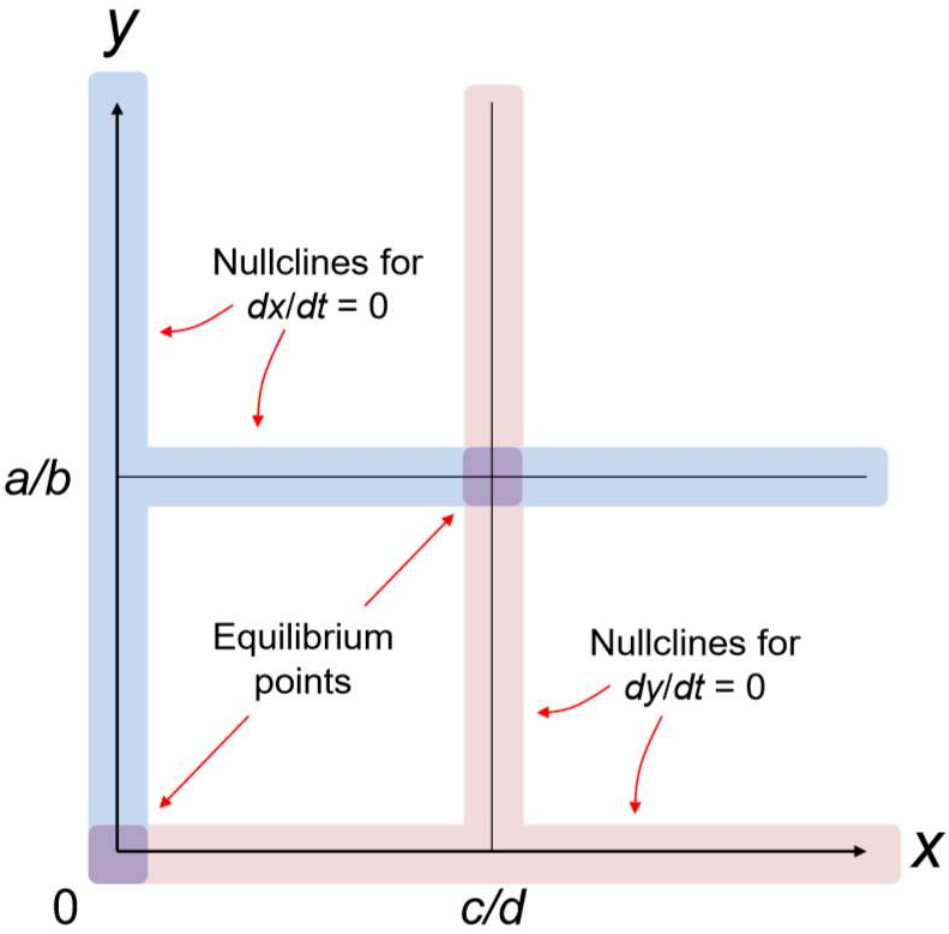


Figure **7.2.2**: Drawing a phase space (1): Adding nullclines.

Everywhere on the first set of nullclines, $\frac{dx}{dt}$ is zero, i.e., there is no "horizontal" movement in the system's state. This means that all local trajectories on and near those nullclines must be flowing vertically. Similarly, everywhere on the second set of nullclines, $\frac{dy}{dt}$ is zero, therefore there is no "vertical" movement and all the local trajectories flow horizontally. These facts can be indicated in the phase space by adding tiny line segments onto each nullcline (Fig. 7.2.3).

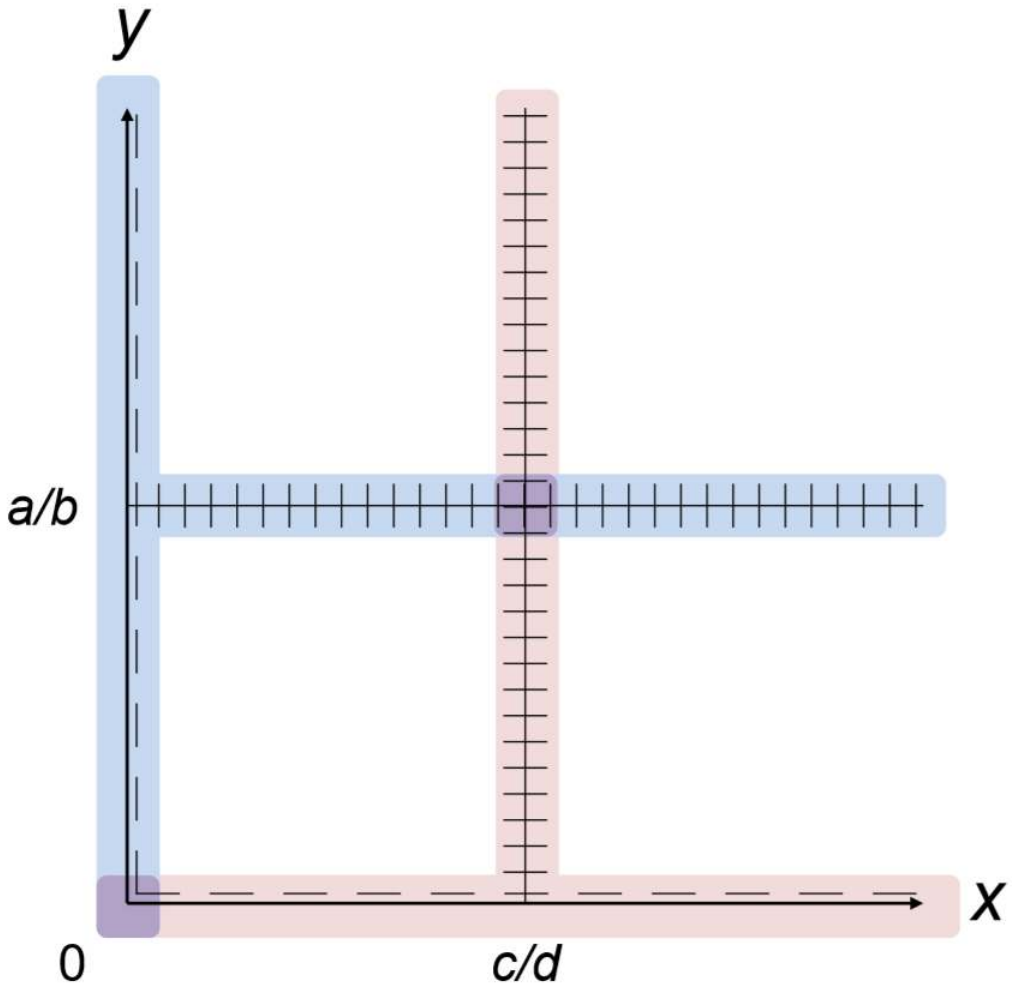


Figure **7.2.3**: Drawing a phase space (2): Adding directions to the nullclines.

Now the phase space is divided into four regions. It is guaranteed that the trajectories in each of those regions flow only in one of the following four directional categories:

$$\cdot \frac{dx}{dt} > 0, \frac{dy}{dt} > 0(''toNortheast'')$$

$$\cdot \frac{dx}{dt} < 0, \frac{dy}{dt} > (to''Northwest'')$$

$$\cdot \frac{dx}{dt} > 0, \frac{dy}{dt} < 0(to''Southeast'')$$

$$\cdot \frac{dx}{dt} <, \frac{dy}{dt} < 0(to''Southwest'')$$

Inside any region, a trajectory never switches between these four categories, because if it did, such a switching point would have to have already appeared as part of the nullclines. Therefore, sampling just one point from each region is sufficient to know which direction the trajectories are flowing in. For example, you can pick $(2c/d, 2a/b)$ as a sample point in the upper right region. You plug this coordinate into the model equations to obtain

$$\frac{dx}{dt}|_{(x,y)=(\frac{2c}{d},\frac{2a}{b})} = a\frac{2c}{d} - b\frac{2c}{d}\frac{2a}{b} = -\frac{2ac}{d} < 0, \tag{7.2.11}$$

$$\frac{dy}{dt}|_{(x,y)=(\frac{2c}{d},\frac{2a}{b})} = -c\frac{2a}{b} + d\frac{2c}{d}\frac{2a}{b} = \frac{2ac}{b} > -. \tag{7.2.12}$$

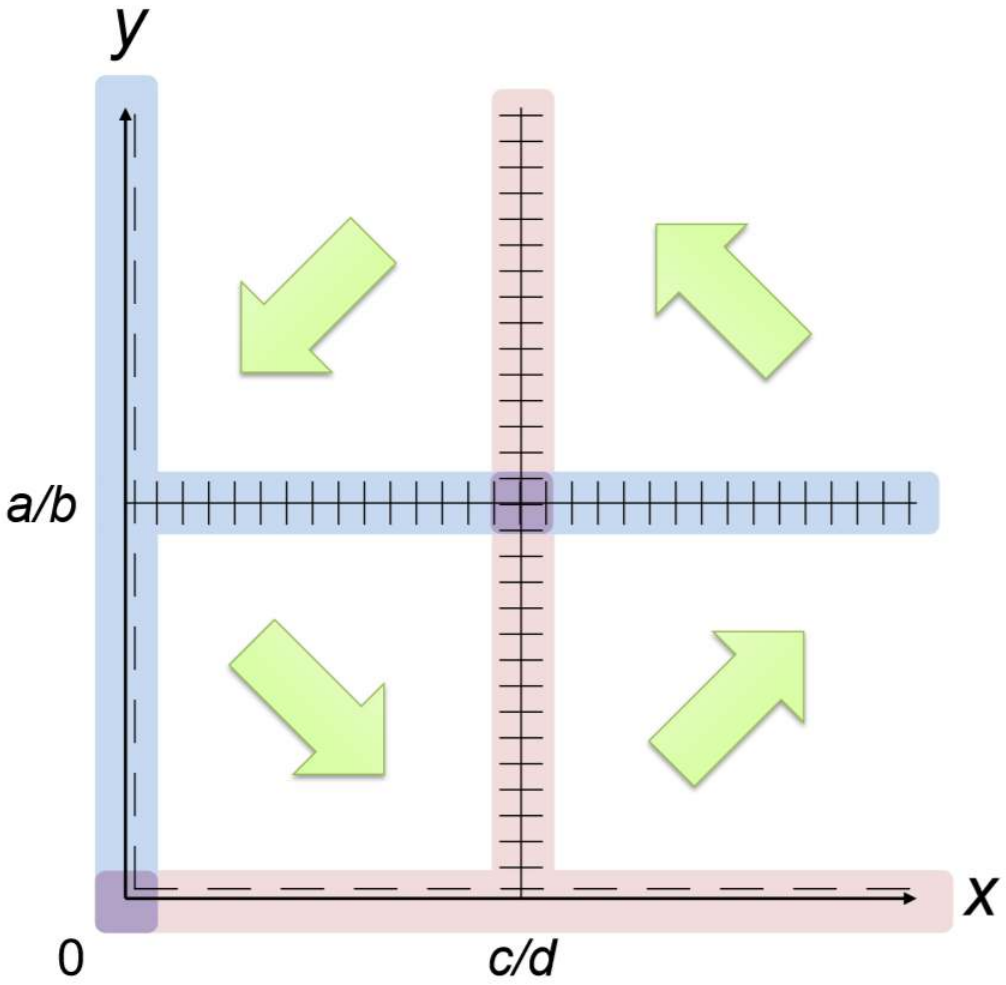


Figure 7.2.4: Drawing a phase space (3): Adding directions of trajectories in each region.

This kind of manual reconstruction of phase space structure can't tell you the exact shape of a particular trajectory, which are typically obtained through numerical simulation. For example, in the phase space manually drawn above, all we know is that the system's behavior is probably rotating around the equilibrium point at $(x, y) = (c/d, a/b)$, but we can't tell if the trajectories are closed orbits, spiral into the equilibrium point, or spiral away from the equilibrium point, until we numerically simulate the system's behavior.

Having said that, there is still merit in this analytical work. First, analytical calculations of nullclines and directions of trajectories provide information about the underlying structure of the phase space, which is sometimes unclear in a numerically visualized phase space. Second, analytical work may allow you to construct a phase space without specifying detailed parameter values (as we did in the example above), whose result is more general with broader applicability to real-world systems than a phase space visualization with specific parameter values.

? Exercise 7.2.1

Draw an outline of the phase space of the following SIR model (variable R is omitted here) by studying nullclines and estimating the directions of trajectories within each region separated by those nullclines.

$$\frac{dS}{dt} = -aSI \tag{7.2.13}$$

$$\frac{dI}{dt} = aSI - bI \tag{7.2.14}$$

$$S \geq 0, I \geq 0, a > 0, b > 0 \tag{7.2.15}$$

Exercise 7.2.2

Draw an outline of the phase space of the following equation by studying nullclines and estimating the directions of trajectories within each region separated by those nullclines.

$$\frac{d^2x}{dt^2} - x\frac{dx}{dt} + x^2 = 0 \tag{7.2.16}$$

This page titled 7.2: Phase Space Visualization is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

7.3: Variable Rescaling of Continuous-Time Models

Variable rescaling of continuous-time models has one distinct difference from variable rescaling that of discrete-time models. That is, you get one more variable you can rescale: time. This may allow you to eliminate one more parameters from your model compared to discrete-time cases.

Here is an example: the logistic growth model. Remember that its discrete-time version

$$x_t = x_{t-1} + r x_{t-1}(1 - \frac{x_{t-1}}{K}) \tag{7.3.1}$$

was simplified to the following form:

$$x'_t = r' x'_{t-1}(1 - x'_{t-1}) \tag{7.3.2}$$

There was still one parameter (r') remaining in the model even after rescaling. In contrast, consider a continuous-time version of the same logistic growth model:

$$\frac{dx}{dt} = rx(1 - \frac{x}{K}) \tag{7.3.3}$$

Here we can apply the following two rescaling rules to both state variable x and time t:

$$x \to ax' \tag{7.3.4}$$

$$t \to \beta t' \tag{7.3.5}$$

With these replacements, Equation 7.3.3 is simplified as

$$\frac{d(ax')}{d(\beta t')} = r a x'(1 - \frac{ax'}{K}) \tag{7.3.6}$$

$$\frac{\beta}{\alpha} \cdot \frac{d(ax')}{d(\beta t')} = \frac{\beta}{\alpha} \cdot r\alpha x'(1 - \frac{\alpha x'}{K}) \tag{7.3.7}$$

$$\frac{dx'}{dt'} = r\beta x'(1 - \frac{\alpha x'}{K}) \tag{7.3.8}$$

$$\frac{dx'}{dt'} = x'(1 - x') \tag{7.3.9}$$

with $\alpha = K$ and $\beta = 1/r$. Note that the final result doesn't contain any parameter left! This means that, unlike its discrete-time counterpart, a continuous-time logistic growth model doesn't change its essential behavior when the model parameters (r, K) are varied. They only change the scaling of trajectories along the t or x axis.

? Exercise 7.3.1

Simplify the following differential equation by variable rescaling:

$$\frac{dx}{dt} = ax^2 + bx + c \tag{7.3.10}$$

? Exercise 7.3.2

Simplify the following differential equation by variable rescaling:

$$\frac{dx}{dt} = \frac{a}{x + b} \tag{7.3.11}$$

$$a > 0, b > 0 \tag{7.3.12}$$

? Exercise 7.3.3

Simplify the following two-dimensional differential equation model by variable rescaling:

$$\frac{dx}{dt} = ax(1-x) - bxy \tag{7.3.13}$$

$$\frac{dy}{dt} = cy(1-y) - dxy \tag{7.3.14}$$

This page titled 7.3: Variable Rescaling of Continuous-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

7.4: Asymptotic Behavior of Continuous-Time Linear Dynamical Systems

A general formula for continuous-time linear dynamical systems is given by

$$\frac{dx}{dt} = Ax, \tag{7.4.1}$$

where x is the state vector of the system and A is the coefficient matrix. As discussed before, you could add a constant vector a to the right hand side, but it can always be converted into a constant-free form by increasing the dimensions of the system, as follows:

$$y = \begin{pmatrix} x \\ 1 \end{pmatrix} \tag{7.4.2}$$

$$\frac{dy}{dt} = \begin{pmatrix} A & a \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ 1 \end{pmatrix} = By \tag{7.4.3}$$

Note that the last-row, last-column element of the expanded coefficient matrix is now 0, not 1, because of Eq. 6.3.9. This result guarantees that the constant-free form given in Equation 7.4.1 is general enough to represent various dynamics of linear dynamical systems. Now, what is the asymptotic behavior of Equation 7.4.1? This may not look so intuitive, but it turns out that there is a closed-form solution available for this case as well. Here is the solution, which is generally applicable for any square matrix A:

$$x(t) = c^{At}x(0) \tag{7.4.4}$$

Here, e^X is a matrix exponential for a square matrix X, which is defined as

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}, \tag{7.4.5}$$

ith $X^(0) = I$. This is a Taylor series-based definition of a usual exponential, but now it is generalized to accept a square matrix instead of a scalar number (which is a 1 x 1 square matrix, by the way). It is known that this infinite series always converges to a well-defined square matrix for any X. Note that $e^(X)$ is the same size as X.

? Exercise 7.4.1

Confirm that the solution Equation ??? satisfies Equation 7.4.1.

The matrix exponential e^X has some interesting properties. First, its eigenvalues are the exponentials of X'seigenvalues. Second, its eigenvectors are thesame as X's eigenvectors. That is:

$$Xv = \lambda v \;\Rightarrow\; x^X v = e^\lambda v \tag{7.4.6}$$

? Exercise 7.4.2

Confirm Equation 7.4.6 using Equation 7.4.5.

We can use these properties to study the asymptotic behavior of Equation 7.4.4. As in Chapter 5, we assume that A is diagonalizable and thus has as many linearly independent eigenvectors as the dimensions of the state space. Then the initial state of the system can be represented as

$$x(0) = b_1v_1 + b_2v_2 + \ldots b_nv_n, \tag{7.4.7}$$

where n is the dimension of the state space and vi are the eigenvectors of A (and of e^A). Applying this to Equation 7.4.4 results in

$$x(t) = e^{At}(b_1v_1 + b_2v_2 + \ldots b_nv_n) \tag{7.4.8}$$

$$= (b_1e^{At}v_1 + b_2e^{At}v_2 + \ldots b_ne^{At}v_n)7.48) \tag{7.4.9}$$

$$= (b_1e^{\lambda_1 t}v_1 + b_2e^{(\lambda_2 t}v_2 + \ldots b_ne^{\lambda_n t}v_n). \tag{7.4.10}$$

This result shows that the asymptotic behavior of $x(t)$ is given by a summation of multiple exponential terms of e^{λ_i} (note the difference—this was λ_i for discrete-time models). Therefore, which term eventually dominates others is determined by the absolute value of e^{λ_i}. Because $|e^{\lambda_i}| = e^{Re(\lambda_i)}$, this means that the eigenvalue that has the largest real part is the dominant eigenvalue for continuous-time models. For example, if λ_1 has the largest real part $(Re(\lambda_1) > Re(\lambda_2), Re(\lambda_3), \dots Re(\lambda_n))$, then

$$x(t) = e^{\lambda_1 t}(b_1 v_1 + b_2 e^{(\lambda_2 - \lambda_1)t} v_2 + \dots b_n e^(\lambda_n - \lambda_1)t v_n), \tag{7.4.11}$$

$$\lim_{t \to \infty} x(t) \approx e^{\lambda_1 t} b_1 v_1. \tag{7.4.12}$$

Similar to discrete-time models, the dominant eigenvalues and eigenvectors tell us the asymptotic behavior of continuous-time models, but with a little different stability criterion. Namely, if the *real part* of the dominant eigenvalue is greater than 0, then the system diverges to infinity, i.e., *the system is unstable*. If it is less than 0, the system eventually shrinks to zero, i.e., the system is stable. If it is precisely 0, then the dominant eigenvector component of the system's state is conserved with neither divergence nor convergence, and thus the system may converge to a non-zero equilibrium point. The same interpretation can be applied to non-dominant eigenvalues as well.

An eigenvalue tells us whether a particular component of a system's state (given by its corresponding eigenvector) grows or shrinks over time. For continuous-time models:

- Re(λ) > 0 means that the component is growing.
- Re(λ) < 0 means that the component is shrinking.
- Re(λ) = 0 means that the component is conserved.

For continuous-time models, the real part of the dominant eigenvalue λd determines the stability of the whole system as follows:

• Re$(\lambda_d) > 0$: The system is ***unstable***, diverging to infinity.

• Re$(\lambda_d) < 0$: The system is ***stable***, converging to the origin.

• Re$(\lambda_d) = 0$: The system is ***stable***, but the dominant eigenvector component is conserved, and therefore the system may converge to a non-zero equilibrium point.

Here is an example of a general two-dimensional linear dynamical system in continuous time (a.k.a. the "love affairs" model proposed by Strogatz [29]):

$$\frac{dx}{dt} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} x = Ax \tag{7.4.13}$$

The eigenvalues of the coefficient matrix can be obtained by solving the following equation for λ:

$$\det \begin{pmatrix} a - \lambda & b \\ c & d - \lambda \end{pmatrix} = 0 \tag{7.4.14}$$

Or:

$$(a - \lambda)(d - \lambda) - bc = \lambda^2 - (a + d)\lambda + ad - bc \tag{7.4.15}$$

$$= \lambda^2 - Tr(A)\lambda + det(A) = 0 \tag{7.4.16}$$

Here, $Tr(A)$ is the trace of matrix A, i.e., the sum of its diagonal components. The solutions of the equation above are

$$\lambda = \frac{Tr(A) \pm \sqrt{Tr(A)^2 - 4det(A)}}{2}. \tag{7.4.17}$$

Between those two eigenvalues ,which one isdominant? Since the radical on the numerator gives either a non-negative real value or an imaginary value, the one with a "plus" sign always has the greater real part. Now we can find the conditions for which this

system is stable. The real part of this dominant eigenvalue is given as follows:

$$Re(\lambda_d) = \begin{cases} \frac{Tr(A)}{2} \text{ if } Tr(A)^2 < 4det(A) \\ \frac{Tr(A)+\sqrt{Tr(A)^2-4det(A)}}{2} \text{ if } Tr(A)^2 \geq 4det(A) \end{cases} \tag{7.4.18}$$

If $Tr(A)^2 < 4\ det(A)$, the stability condition is simply

$$Tr(A) < 0. \tag{7.4.19}$$

If $Tr(A)^2 \geq 4\ det(A)$, the stability condition is derived as follows:

$$Tr(A) + \sqrt{Tr(A)^2 - 4\ det(A)} < 0 \tag{7.4.20}$$

$$\sqrt{Tr(A)^2 - 4\ det(A)} < -Tr(A) \tag{7.4.21}$$

Since the radical on the left hand side must be non-negative, $Tr(A)$ must be negative, at least. Also, by squaring both sides, we obtain

$$Tr(A)^2 - 4det(A) < Tr(A)^2 \tag{7.4.22}$$

$$-4det(A) < 0, \tag{7.4.23}$$

$$det(A) > 0. \tag{7.4.24}$$

By combining all the results above, we can summarize how the two-dimensional linear dynamical system's stability depends on $Tr(A)$ and $det(A)$ in a simple diagram as shown in Fig. 7.5. Note that this diagram is applicable only to two-dimensional systems, and it is not generalizable for systems that involve three or more variables.

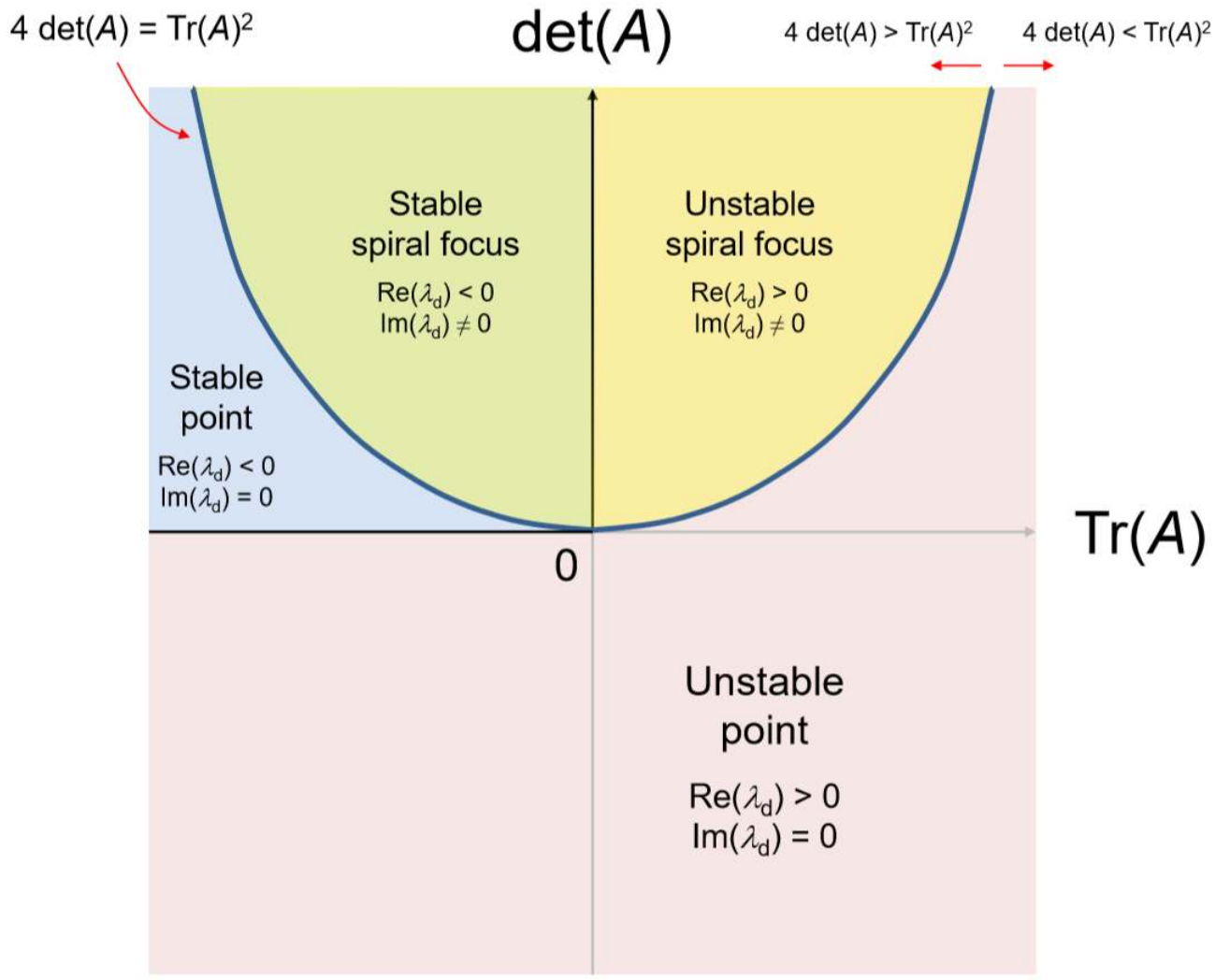


Figure 7.4.1: Overview of how the stability of a two-dimensional continuous-time linear dynamical system depends on $Tr(A)$ and $det(A)$.

? Exercise 7.4.1

Show that the unstable points with $det(A) < 0$ are saddle points.

? Exercise 7.4.2

Determine the stability of the following linear systems:

$$\cdot \frac{dx}{dt} = \begin{pmatrix} -1 & 2 \\ 2 & -2 \end{pmatrix} x$$

$$\cdot \frac{dx}{dt} = \begin{pmatrix} 0.5 & -1.5 \\ 1 & -1 \end{pmatrix} x$$

? Exercise 7.4.3

Confirm the analytical result shown in Fig. 7.4.1 by conducting numerical simulations in Python and by drawing phase spaces of the system for several samples of A.

This page titled 7.4: Asymptotic Behavior of Continuous-Time Linear Dynamical Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

7.5: Linear Stability Analysis of Nonlinear Dynamical Systems

Finally, we can apply *linear stability* analysis to continuous-time nonlinear dynamical systems. Consider the dynamics of a nonlinear differential equation

$$\frac{dx}{dt} = F(x) \tag{7.5.1}$$

around its equilibrium point x_{eq}. By definition, x_{eq} satisfies

$$0 = F(x_{eq}). \tag{7.5.2}$$

To analyze the stability of the system around this equilibrium point, we do the same coordinate switch as we did for discrete-time models. Specifically, we apply the following replacement

$$x(t) \Rightarrow x_{eq} + \Delta x(t) \tag{7.5.3}$$

to Equation 7.5.1, to obtain

$$\frac{d(x_{eq} + \Delta x)}{dt} = \frac{d\Delta x}{dt} = F(x_{eq} + \Delta x) \tag{7.5.4}$$

Now that we know the nonlinear function F on the right hand side can be approximated using the ***Jacobian matrix***, the equation above is approximated as

$$\frac{d\Delta x}{dt} \approx F(x_{eq}) + J\Delta x, \tag{7.5.5}$$

where J is the Jacobian matrix of F at $x = x_{eq}$ (if you forgot what the Jacobian matrix was, see Eq. (5.7.18)). By combining the result above with Equation 7.5.2, we obtain

$$\frac{d\Delta x}{dt} \approx J\Delta x \tag{7.5.6}$$

Note that the final result is very similar to that of discrete-time models. The rest of the process is something we are already familiar with: Calculate the eigenvalues of J and interpret the results to determine the stability of equilibrium point x_{eq}. The only differences from discrete-time models are that you need to look at the real parts of the eigenvalues, and then compare them with 0, not 1. Figure 7.6 shows a schematic summary of classifications of equilibrium points for two-dimensional continuous-time dynamical systems.

Linear stability analysis of continuous-time nonlinear systems

1. Find an equilibrium point of the system you are interested in.
2. Calculate the Jacobian matrix of the system at the equilibrium point.
3. Calculate the eigenvalues of the Jacobian matrix.
4. If the real part of the dominant eigenvalue is:
    - Greater than 0 $\Rightarrow$The equilibrium point is unstable.
        - If other eigenvalues have real parts less than 0, the equilibrium point is a saddle point.
    - Less than0 $\Rightarrow$The equilibrium point is stable.
    - Equal to0 $\Rightarrow$The equilibrium point may be neutral (Lyapunov stable).
5. In addition, if there are complex conjugate eigenvalues involved, oscillatory dynamics are going on around the equilibrium point. If those complex conjugate eigenvalues are the dominant ones, the equilibrium point is called a stable or unstable spiral focus (or a neutral center if the point is neutral).

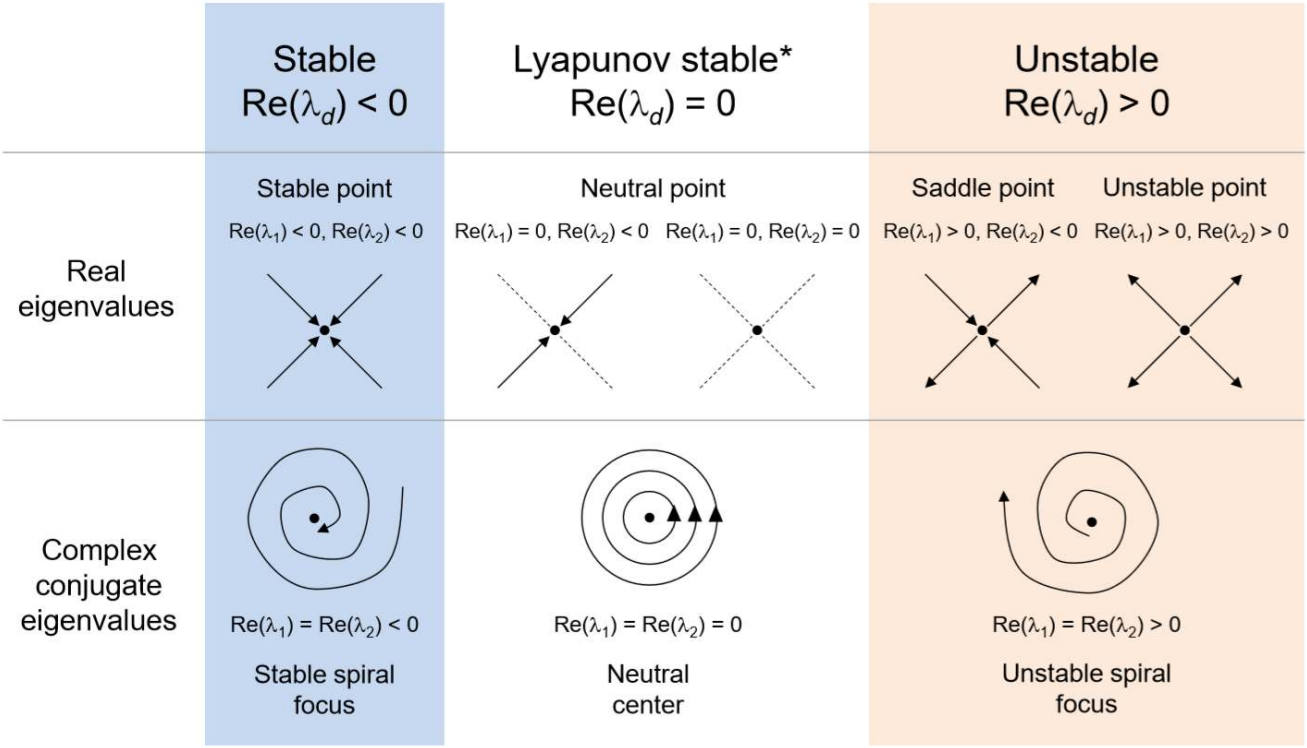


Figure **7.5.1**: Schematic illustrations of several different types of equilibrium points and their nearby trajectories in 2-D continuous-time dynamical systems, shown in the same format as in Fig. 5.7.2. The real part of the dominant eigenvalue $Re(\lambda_d)$ determines the overall stability of the equilibrium point (top), although it would generally require nonlinear analysis to show that the equilibrium point is truly neutral/Lyapunov stable (*).

? Exercise 7.5.1

Consider the logistic growth model ($r > 0, K > 0$):

$$\frac{dx}{dt} = rx(1 - \frac{x}{K}) \tag{7.5.7}$$

Conduct a linear stability analysis to determine whether this model is stable or not at each of its equilibrium points $x_{eq} = 0, K$.

? Exercise 7.5.2

Consider the following differential equations that describe the interaction between two species called *commensalism* (species x benefits from the presence of species y but doesn't influence y):

$$\frac{dx}{dt} = -x + rxy - x^2 \tag{7.5.8}$$

$$\frac{dy}{dt} = y(1 - y) \tag{7.5.9}$$

$$x \geq 0, y \geq 0, r > 1 \tag{7.5.10}$$

1. Find all the equilibrium points.
2. Calculate the Jacobian matrix at the equilibrium point where $x > 0$ and $y > 0$.
3. Calculate the eigenvalues of the matrix obtained above.
4. Based on the result, classify the equilibrium point into one of the following: Stable point, unstable point, saddle point, stable spiral focus, unstable spiral focus, or neutral center.

Exercise 7.5.3

Consider the differential equations of the $SIR model$:

\[\frac{dS}{dI}= -aSI\label{(7.74)}]

$$\frac{dI}{dt} = aSI - bI \tag{7.5.11}$$

$$\frac{dR}{dt} = bI \tag{7.5.12}$$

As you see in the equations above, R doesn't influence the behaviors of S and I, so you can safely ignore the third equation to make the model two-dimensional. Do the following:

1. Find all the equilibrium points (which you may have done already in Exercise 7.1.3).

2. Calculate the Jacobian matrix at each of the equilibrium points.

3. Calculate the eigenvalues of each of the matrices obtained above. 4. Based on the results, discuss the stability of each equilibrium point.

This page titled 7.5: Linear Stability Analysis of Nonlinear Dynamical Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

8: Bifurcations

8.1: 8.1 What are Bifurcations?

8.2: Bifurcations in 1-D Continuous-Time Models

8.3: Hopf Bifurcations in 2-D Continuous-Time Models

8.4: Bifurcations in Discrete-Time Models

This page titled 8: Bifurcations is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

8.1: 8.1 What are Bifurcations?

One of the important questions you can answer by mathematically analyzing a dynamical system is how the system's long-term behavior depends on its parameters. Most of the time, you can assume that a slight change in parameter values causes only a slight quantitative change in the system's behavior too, with the essential structure of the system's phase space unchanged. However, sometimes you may witness that a slight change in parameter values causes a drastic, qualitative change in the system's behavior, with the structure of its phase space topologically altered. This is called a *bifurcation*, and the parameter values at which a bifurcation occurs are called the *critical thresholds*.

Bifurcation

Bifurcation is a qualitative, topological change of a system's phase space that occurs when some parameters are slightly varied across their critical thresholds.

Bifurcations play important roles in many real-world systems as a switching mechanism. Examples include excitation of neurons, pattern formation in morphogenesis (this will be discussed later), catastrophic transition of ecosystem states, and binary information storage in computer memory, to name a few.

There are two categories of bifurcations. One is called a *local bifurcation*,which can be characterized by a change in the stability of equilibrium points. It is called local because it can be detected and analyzed only by using localized information around the equilibrium point. The other category is called a *global bifurcation*, which occurs when non-local features of the phase space, such as limit cycles (to be discussed later), collide with equilibrium points in a phase space. This type of bifurcation can't be characterized just by using localized information around the equilibrium point. In this textbook, we focus only onthe local bifurcations, as they can be easily analyzed using the concepts of linear stability that we discussed in the previous chapters.

Local bifurcations occur when the stability of an equilibrium point changes between stable and unstable. Mathematically, this condition can be written down as follows:

Local bifurcations occur when the eigenvalues λ_i of the Jacobian matrix at an equilibrium point satisfy the following:

- **For discrete-time models:** $|\lambda_i| = 1$ for some i, while$|\lambda_i| < 1$ for the rest.
- **For continuous-time models:** $Re(\lambda_i) = 0$ for some i, while $Re(\lambda_i) < 0$ for the rest.

Theseconditionsdescribeacriticalsituationwhentheequilibriumpointisabouttochange its stability. We can formulate these conditions in equations and then solve them in terms of the parameters, in order to obtain their critical thresholds. Let's see how this analysis can be done through some examples below.

This page titled 8.1: 8.1 What are Bifurcations? is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

8.2: Bifurcations in 1-D Continuous-Time Models

For bifurcation analysis, continuous-time models are actually simpler than discrete-time models (we will discuss the reasons for this later). So let's begin with the simplest example, a continuous-time, first-order, autonomous dynamical system with just one variable:

$$\frac{dx}{dt} = F(x) \tag{8.2.1}$$

In this case, the Jacobian matrix is a 1×1 matrix whose eigenvalue is its content itself (because it is a scalar), which is given by $\frac{dF}{dx}$. Since this is a continuous-time model, the critical condition at which a bifurcation occurs in this system is given by

$$Re\,\frac{dF}{dx}\Big|_{x=x_{eq}} = \frac{dF}{dx}\Big|_{x=x_{eq}} = 0. \tag{8.2.2}$$

Let's work on the following example:

$$\frac{dx}{dt} = r - x^2 \tag{8.2.3}$$

The first thing we need to do is to find the equilibrium points, which is easy in this case. Letting $\frac{dx}{dt} = 0$ immediately gives

$$x_{eq} = \pm\sqrt{r}, \tag{8.2.4}$$

which means that equilibrium points exist only for non-negative r. The critical condition when a bifurcation occurs is given as follows:

$$\frac{dF}{dx} = -2x \tag{8.2.5}$$

$$\frac{dF}{dx}|_{x=x_{eq}} = \pm 2\sqrt{r} = 0 \tag{8.2.6}$$

$$r = 0 \tag{8.2.7}$$

Therefore, now we know a bifurcation occurs when $r = 0$. Moreover, by plugging each solution of Equation ??? into $\frac{dF}{dx} = -2x$, we know that one equilibrium point is stable while the other is unstable. These results are summarized in Table 8.1.

Table* 8.2.1: Summary of bifurcation analysis of* $dx/dt = r - x^2$.

Equilibrium point	$r < 0$	$0 < r$
$x_{eq} = \sqrt{r}$	doesn't exist	stable
$x_{eq} = -\sqrt{r}$	doesn't exist	unstable

There is a more visual way to illustrate the results. It is called a *bifurcation diagram*. This works only for systems with one variable and one parameter, but it is still conceptually helpful in understanding the nature of bifurcations. A bifurcation diagram can be drawn by using the parameter being varied as the horizontal axis, while using the location(s) of the equilibrium point(s) of the system as the vertical axis. Then you draw how each equilibrium point depends on the parameter, using different colors and/or line styles to indicate the stability of the point. Here is an example of how to draw a bifurcation diagram in Python:

Code 8.1: bifurcation-diagram.py

```
from pylab import *

def xeq1(r):
    return sqrt(r)

def xeq2(r):
    return -sqrt(r)

domain = linspace(0, 10)
plot(domain, xeq1(domain), 'b-', linewidth = 3)
```

```
plot(domain, xeq2(domain), 'r--', linewidth = 3)
plot([0], [0], 'go')
axis([-10, 10, -5, 5])
xlabel('r')
ylabel('x_eq')

show()
```

The result is shown in Fig. 8.2.1, where the blue solid curve indicates a stable equilibrium point $x_{eq} = \sqrt{r}$, and the red dashed curve indicates an unstable equilibrium point $x_{eq} = -\sqrt{r}$, with the green circle in the middle showing a neutral equilibrium point. This type of bifurcation is called a $saddle-node\ bifurcation$, in which a pair of equilibrium points appear (or collide and annihilate, depending on which way you vary r).

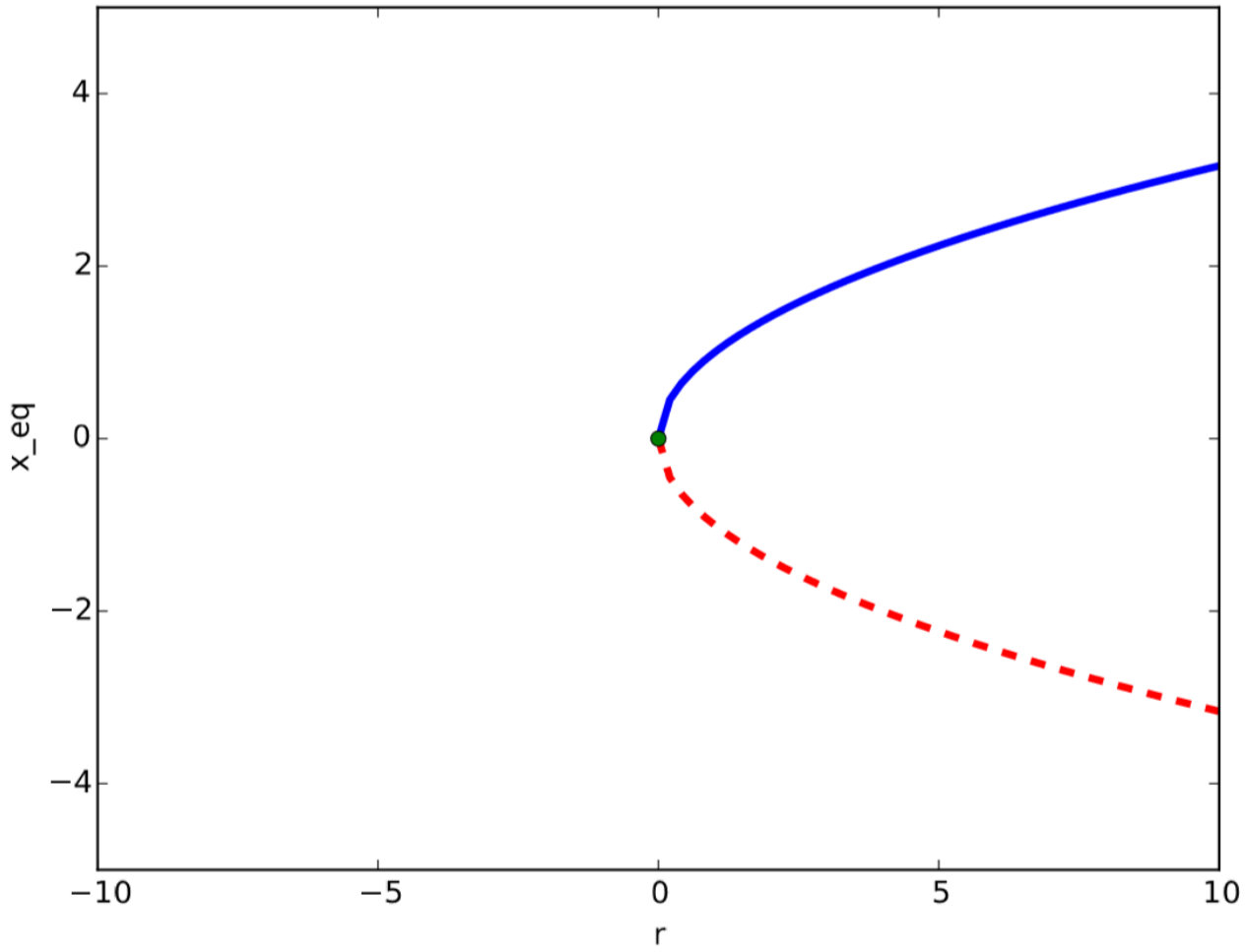


Figure 8.2.1: Visual output of Code 8.1, showing a bifurcation diagram of a saddle-node bifurcation, obtained from Eq 8.2.3.

Each vertical slice of the bifurcation diagram for a particular parameter value depicts a phase space of the dynamical system we are studying. For example, for $r = 5$ in the diagram above, there are two equilibrium points, one stable (blue/solid) and the other unstable (red/dashed). You can visualize flows of the system's state by adding a downward arrow above the stable equilibrium point, an upward arrow from the unstable one to the stable one, and then another downward arrow below the unstable one. In this way, it is clear that the system's state is converging to the stable equilibrium point while it is repelling from the unstable one. If you do the same for several different values of r, you obtain Fig. 8.2.2, which shows how to interpret this diagram.

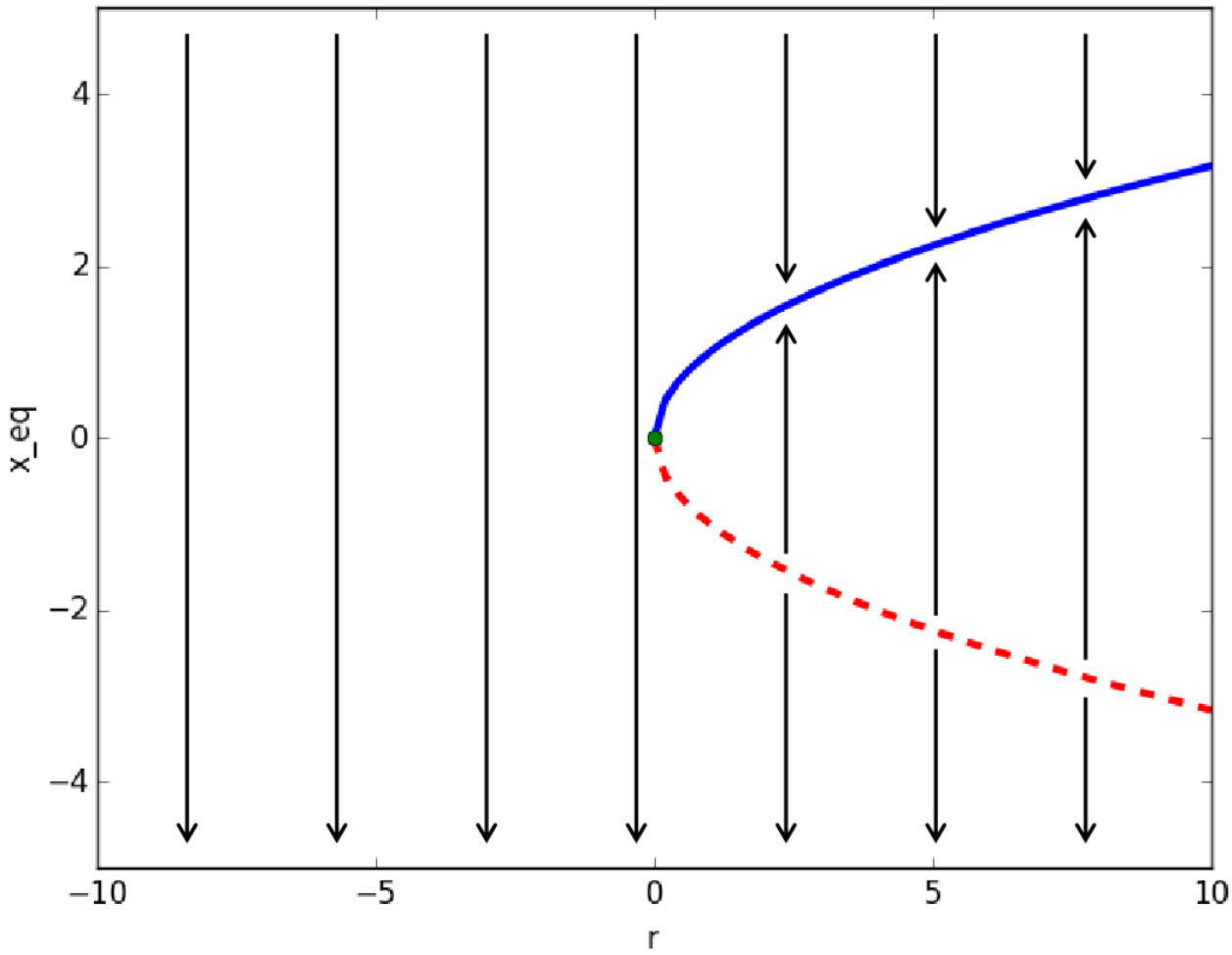


Figure 8.2.2: How to interpret a bifurcation diagram. Each vertical slice of the diagram depicts a phase space of the system for a particular parameter value.

There are other types of bifurcations. A *transcritical bifurcation* is a bifurcation where one equilibrium point "passes through" another, exchanging their stabilities. For example:

$$\frac{dx}{dt} = rx - x^2 \tag{8.2.8}$$

This dynamical system always has the following two equilibrium points

$$x_{eq} = 0, r, \tag{8.2.9}$$

with the exception that they collide when $r = 0$, which is when they swap their stabilities. Its bifurcation diagram is shown in Fig. 8.2.3.

Another one is a *pitchfork bifurcation*, where an equilibrium point splits into three. Two of these (the outermost two) have the same stability as the original equilibrium point, while the one between them has a stability opposite to the original one's stability. There are two types of pitchfork bifurcations. A *supercritical pitchfork bifurcation* makes a stable

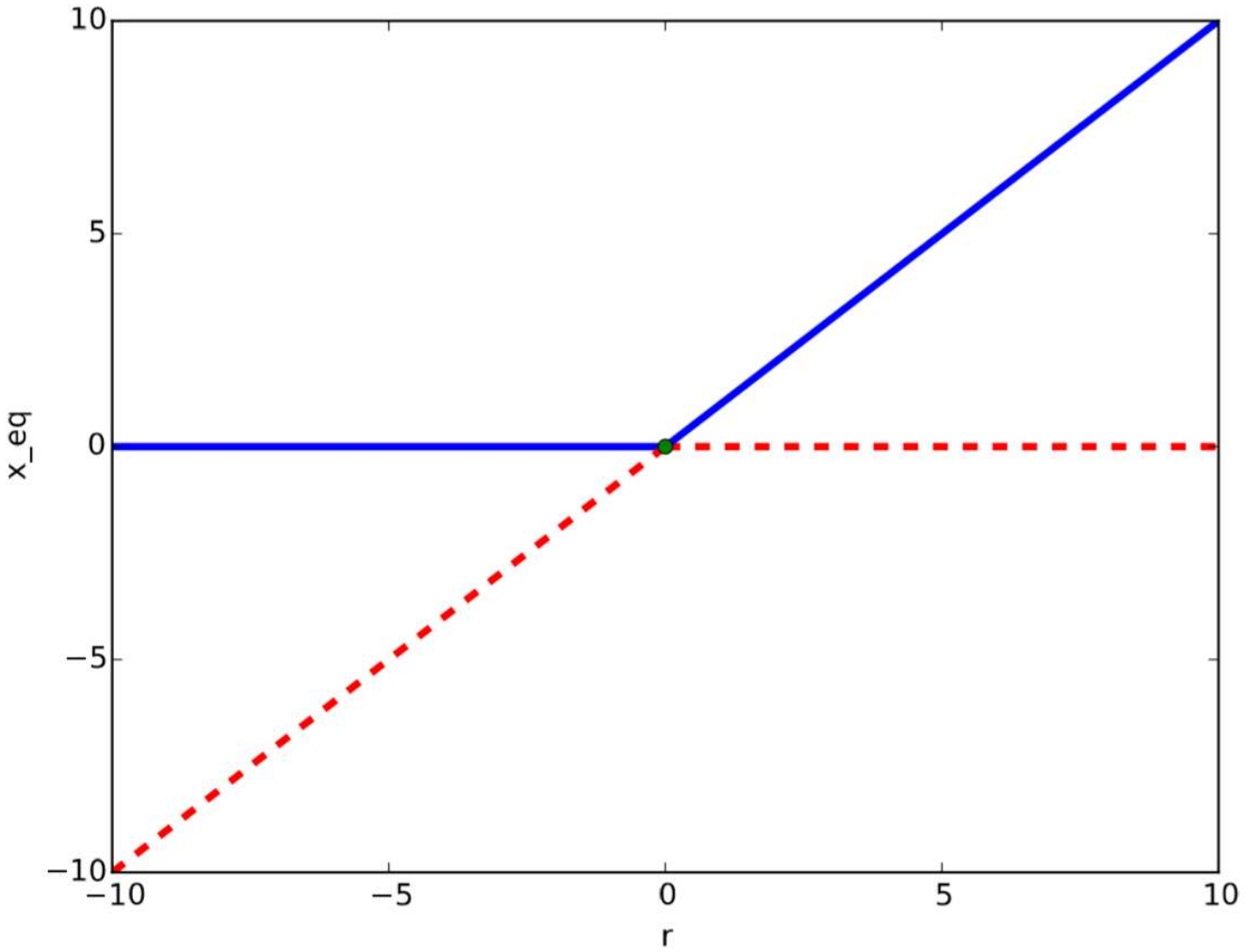


Figure **8.2.3**: Bifurcation diagram of a transcritical bifurcation, obtained from Eq 8.2.8.

equilibrium point split into three, two stable and one unstable. For example:

$$\frac{dx}{dt} = rx - x^3 \tag{8.2.10}$$

This dynamical system has the following three equilibrium points

$$x_{eq} = 0 \pm \sqrt{r}, \tag{8.2.11}$$

but the last two exist only for $r \geq 0$. You can show that $x_{eq} = 0$ is stable for $r < 0$ and unstable for $r > 0$, while $x_{eq} = \pm \sqrt{r}$ are always stable if they exist. Its bifurcation diagram is shown in Fig. 8.2.4.

In the meantime, a *subcritical pitchfork bifurcation* makes an unstable equilibrium point split into three, two unstable and one stable. For example:

$$\frac{dx}{dt} = rx + x^3 \tag{8.2.12}$$

This dynamical system has the following three equilibrium points

$$x_{eq} = 0 \pm \sqrt{-r}, \tag{8.2.13}$$

but the last two exist only for $r \leq 0$. Its bifurcation diagram is shown in Fig. 8.2.5.

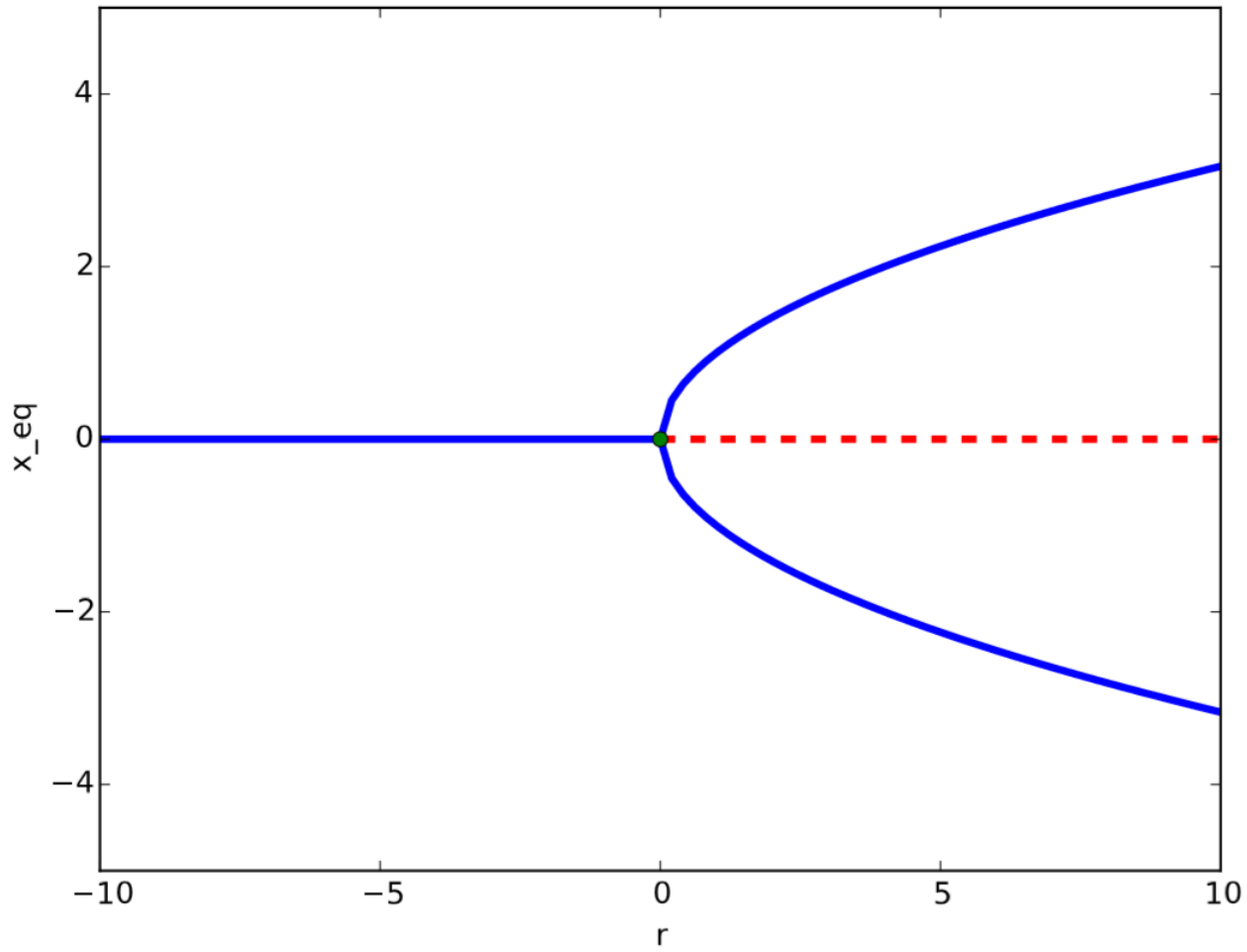


Figure 8.2.4: Bifurcation diagram of a supercritical pitchfork bifurcation, obtained from Eq 8.2.10.

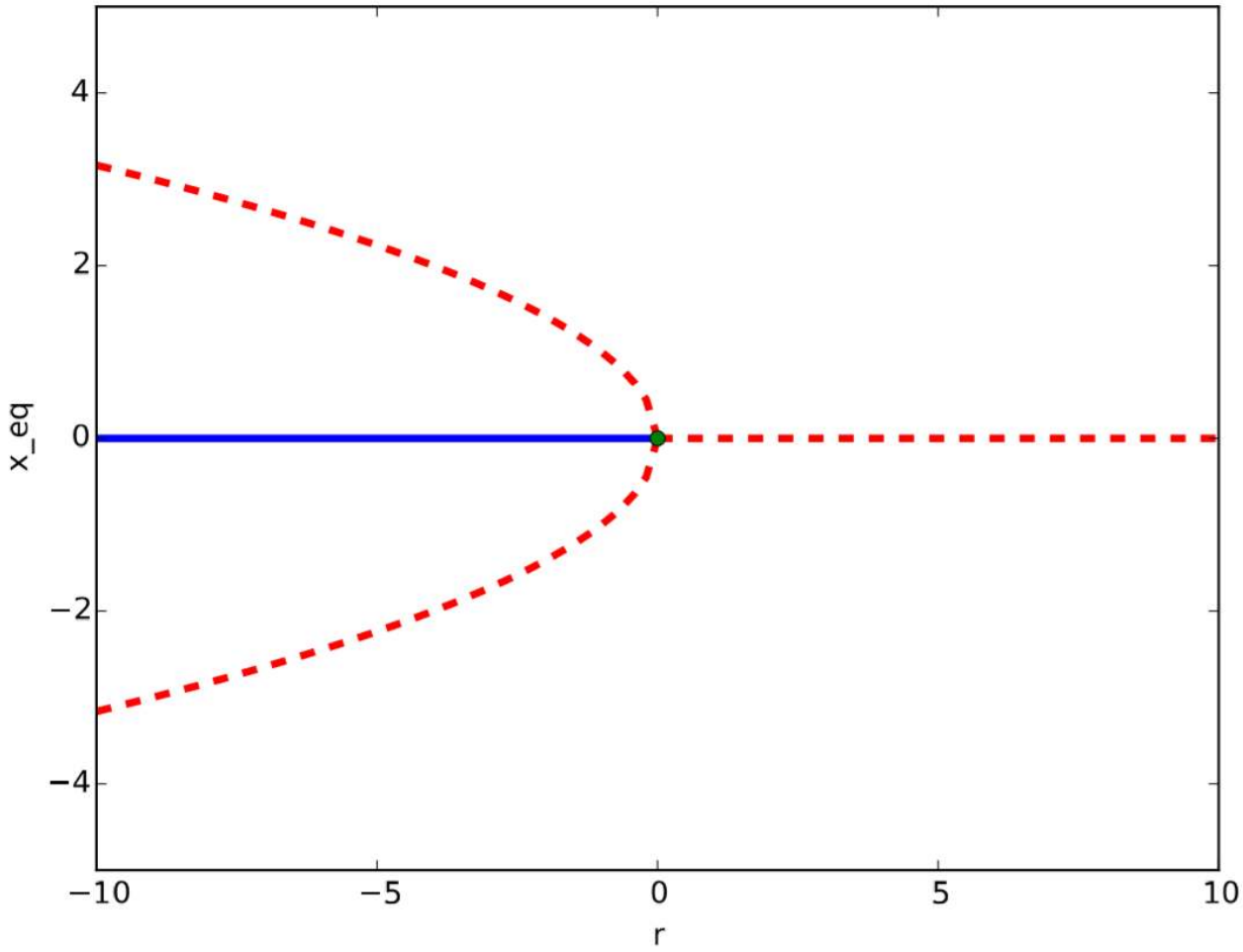


Figure 8.2.5: Bifurcation diagram of a subcritical pitchfork bifurcation, obtained from Eq 8.2.10.

These bifurcations can arise in combined forms too. For example:

$$\frac{dx}{dt} = r + x - x^3 \tag{8.2.14}$$

This dynamical system has three equilibrium points, which are rather complicated to calculate in a straightforward way. However, if we solve $\frac{dx}{dt} = 0$ in terms of r, we can easily obtain

$$r = -x + x^3, \tag{8.2.15}$$

which is sufficient for drawing the bifurcation diagram. We can also know the stability of each equilibrium point by calculating

$$Re\frac{dF}{dx}|_{x=x_{eq}} = 1 - 3x^3, \tag{8.2.16}$$

i.e., when $x^2 > \frac{1}{3}$, the equilibrium points are stable, otherwise they are unstable. The bifurcation diagram of this system is shown in Fig. 8.2.6.

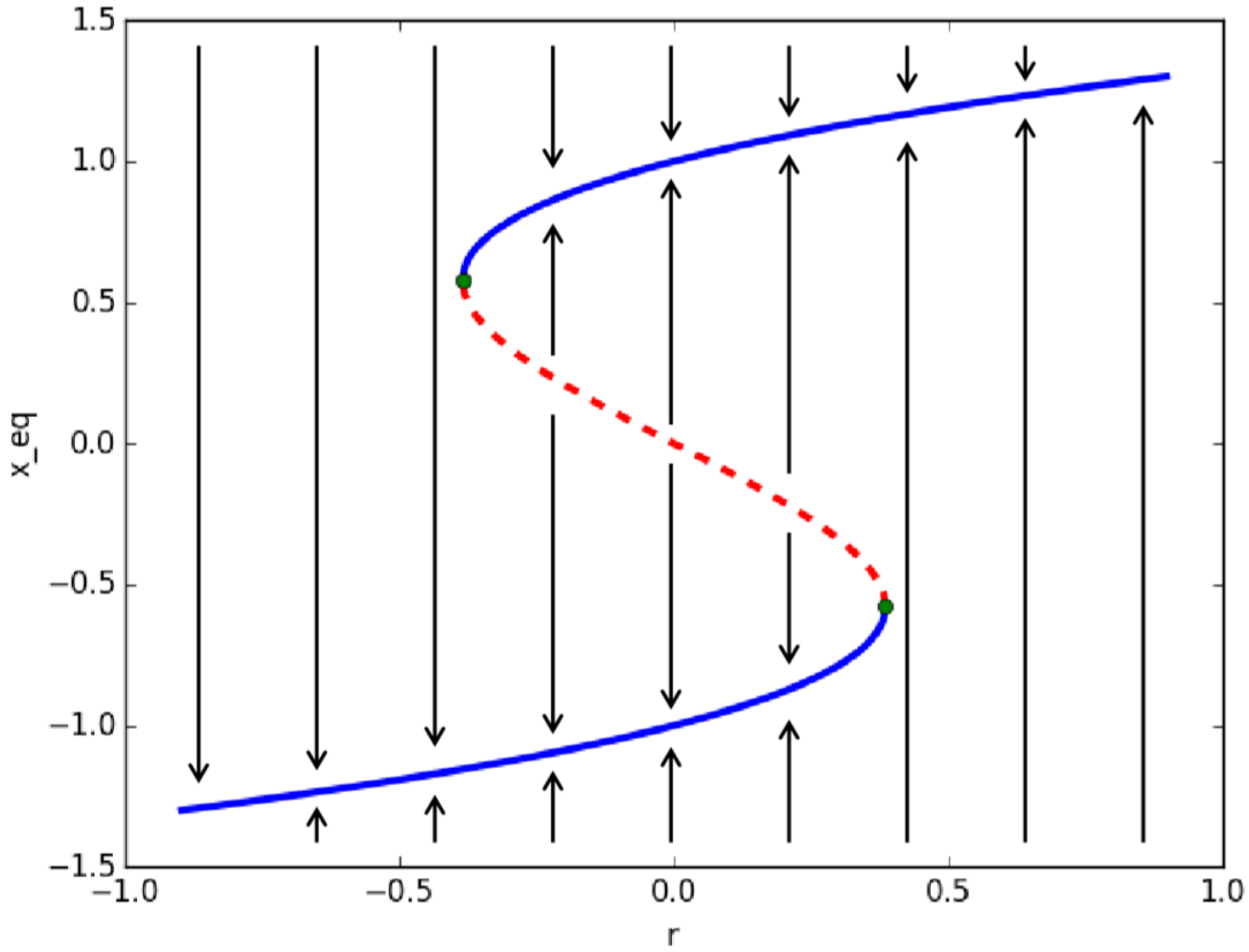


Figure 8.2.5: Bifurcation diagram showing hysteresis, obtained from Eq 8.2.14. Arrows are added to help with interpretation.

This diagram is a combination of two saddle-node bifurcations, showing that this system has *hysteresis* as its dynamical property. Hysteresis is the dependence of a system's output (asymptotic state in this case) not only on the input (parameter r in this case) but also on its history. To understand what this means, imagine that you are slowly changing r from -1 upward. Initially, the system's state stays at the stable equilibrium at the bottom of the diagram, which continues until you reach a critical threshold at $r \approx 0.4$. As soon as you cross this threshold, the system's state suddenly jumps up to another stable equilibrium point at the top of the diagram. Such a sudden jump in the system's state is often called a catastrophe. You get upset, and try to bring the system's state back to where it was, by reducing r. However, counter to your expectation, the system's state remains high even after you reduce r below 0.4. This is hysteresis; the system's asymptotic state depends not just on r, but also on where its state was in the immediate past. In other words, the system's state works as a memory of its history. In order to bring the system's state back down to the original value, you have to spend extra effort to reduce r all the way below another critical threshold, $r \approx -0.4$.

Such hysteresis could be useful; every bit (binary digit) of computer memory has this kind of bifurcation dynamics, which is why we can store information in it. But in other contexts, hysteresis could be devastating—if an ecosystem's state has this property (many studies indicate it does), it takes a huge amount of effort and resources to revert a deserted ecosystem back to a habitat with vegetation, for example.

? Exercise 8.2.1

Conduct a bifurcation analysis of the following dynamical system with parameter r:

$$\frac{dx}{dt} = rx(x+1) - x \tag{8.2.17}$$

Find the critical threshold of r at which a bifurcation occurs. Draw a bifurcation diagram and determine what kind of bifurcation it is.

? Exercise 8.2.2

Assume that two companies, A and B, are competing against each other for the market share in a local region. Let x and y be the market share of A and B, respectively. Assuming that there are no other third-party competitors, $x + y = 1$ (100%), and therefore this system can be understood as a one-variable system. The growth/decay of A's market share can thus be modeled as

$$\frac{dx}{dt} = ax(1-x)(x-y), \tag{8.2.18}$$

here x is the current market share of A, $1 - x$ is the size of the available potential customer base, and $x - y$ is the relative competitive edge of A, which can be rewritten as $x - (1 - x) = 2x - 1$. Obtain equilibrium points of this system and their stabilities. Then make an additional assumption that this regional market is connected to and influenced by a much larger global market, where company A's market share is somehow kept at p (whose change is very slow so we can consider it constant):

$$\frac{dx}{dt} = ax(1-x)(x-y) + r(p-x) \tag{8.2.19}$$

Here r is the strength of influence from the global to the local market. Determine a critical condition regarding r and p at which a bifurcation occurs in this system. Draw its bifurcation diagram over varying r with $a = 1$ and $p = 0.5$, and determine what kind of bifurcation it is. Finally, using the results of the bifurcation analysis, discuss what kind of marketing strategy you would take if you were a director of a marketing department of a company that is currently overwhelmed by its competitor in the local market. How can you "flip" the market?

This page titled 8.2: Bifurcations in 1-D Continuous-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

8.3: Hopf Bifurcations in 2-D Continuous-Time Models

For dynamical systems with two or more variables, the dominant eigenvalues of the Jacobian matrix at an equilibrium point could be complex conjugates. If such an equilibrium point, showing an oscillatory behavior around it, switches its stability, the resulting bifurcation is called a *Hopf bifurcation*. A Hopf bifurcation typically causes the appearance (or disappearance) of a *limit cycle* around the equilibrium point. A limit cycle is a cyclic, closed trajectory in the phase space that is defined as an asymptotic limit of other oscillatory trajectories nearby. You can check whether the bifurcation is Hopf or not by looking at the imaginary components of the dominant eigenvalues whose real parts are at a critical value (zero); if there are non-zero imaginary components, it must be a Hopf bifurcation.

Here is an example, a dynamical model of a nonlinear oscillator, called the *van der Pol oscillator*:

$$\frac{d^2x}{xt^2} + r(x^2-1)\frac{dx}{dt} + x = 0 \tag{8.3.1}$$

This is a second-order differential equation, so we should introduce an additional variable $y = dx/dt$ to make it into a 2-D first-order system, as follows:

$$\frac{dx}{dt} = y \tag{8.3.2}$$

$$\frac{dy}{dt} = r(x^2-1)y - x \tag{8.3.3}$$

From these, we can easily show that the origin, $(x, y) = (0, 0)$, is the only equilibrium point of this system. The Jacobian matrix of this system at the origin is given as follows:

$$\begin{pmatrix} 0 & 1 \\ 2rxy-1 & -r(x^2-1) \end{pmatrix} |_{(x,y)=(0,0)=} \begin{pmatrix} 0 & 1 \\ -1 & r \end{pmatrix} \tag{8.3.4}$$

The eigenvalues of this matrix can be calculated as follows:

$$\begin{vmatrix} 0-\lambda & 1 \\ -1 & r-\lambda \end{vmatrix} = 0 \tag{8.3.5}$$

$$-\lambda(r-\lambda)+1 = \lambda^2 - r\lambda + 1 = 0 \tag{8.3.6}$$

$$\lambda = \frac{r \pm \sqrt{r^2-4}}{2} \tag{8.3.7}$$

The critical condition for a bifurcation to occur is

$$Re(\lambda) = 0, \tag{8.3.8}$$

whose left hand side can be further detailed as

$$Re(\lambda) = \begin{cases} \dfrac{r \pm \sqrt{r^2-4}}{2} & \text{if } r^2 \geq 4, \text{or} \\ \dfrac{r}{2} & \text{if } r^2 < 4. \end{cases} \tag{8.3.9}$$

The first case can't be zero, so the only critical condition for a bifurcation to occur is the second case, i.e.

$r = 0$, when $Re(\lambda) = 0$ and $Im(\lambda) = \pm i$.

This is a Hopf bifurcation because the eigenvalues have non-zero imaginary parts when the stability change occurs. We can confirm this analytical prediction by numerical simulations of the model with systematically varied r, as follows:

Code 8.2: van-del-pol-Hopf-bifurcation.py

```
from pylab import *
```

```
Dt = 0.01

def initialize():
    global x, xresult, y, yresult
    x = y = 0.1
    xresult = [x]
    yresult = [y]

def observe():
    global x, xresult, y, yresult
    xresult.append(x)
    yresult.append(y)
```

```
def update():
    global x, xresult, y, yresult
    nextx = x + y * Dt
    nexty = y + (-r * (x**2 - 1) * y - x) * Dt
    x, y = nextx, nexty

def plot_phase_space():
    initialize()
    for t in xrange(10000):
        update()
        observe()
    plot(xresult, yresult)
    axis('image')
    axis([-3, 3, -3, 3])
```

```
    title('r = ' + str(r))

rs = [-1, -0.1, 0, .1, 1]
for i in xrange(len(rs)):
    subplot(1, len(rs), i + 1)
    r = rs[i]
    plot_phase_space()

show()
```

Figure 8.3.1 shows the results where a clear transition from a stable spiral focus (for $r < 0$) to an unstable spiral focus surrounded by a limit cycle (for $r > 0$) is observed.

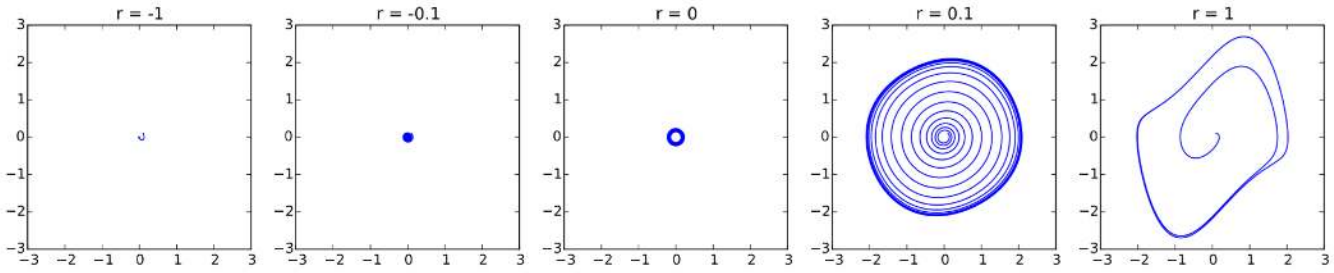


Figure 8.3.1: Visual output of Code 8.2.

? Exercise 8.3.1: FitzHugh-Nagumo model

The FitzHugh-Nagumo model [30, 31] is a simplified model of neuronal dynamics that can demonstrate both ***excited*** and ***resting*** behaviors of neurons. In a normal setting, this system's state converges and stays at a stable equilibrium point (resting), but when perturbed, the system's state moves through a large cyclic trajectory in the phase space before coming back to the resting state, which is observed as a big pulse when plotted over time (excitation). Moreover, under certain conditions, this system can show a nonlinear oscillatory behavior that continuously produces a sequence of pulses. The behavioral shift between convergence to the resting state and generation of a sequence of pulses occurs as a Hopf bifurcation, where the external current is used as a control parameter. Here are the model equations:

$$\frac{dx}{dt} = c(x - \frac{x^3}{3} + y + z) \tag{8.3.10}$$

$$\frac{dy}{dt} = -\frac{x - a + by}{c} \tag{8.3.11}$$

z is the key parameter that represents the external current applied to the neuron. Other parameters are typically constrained as follows:

$$1 - \frac{2}{3}b < a < 1 \tag{8.3.12}$$

$$0 < b < 1 \tag{8.3.13}$$

$$b < c^2 8.34) \tag{8.3.14}$$

With $a = 0.7$, $b = 0.8$, and $c = 3$, do the following:

- Numerically obtain the equilibrium point of this model for several values of z, ranging between -2 and 0. There is only one real equilibrium point in this system.
- Apply the result obtained above to the Jacobian matrix of the model, and numerically evaluate the stability of that equilibrium point for each value of z.
- Estimate the critical thresholds of z at which a Hopf bifurcation occurs. There are two such critical thresholds.
- Draw a series of its phase spaces with values of z varied from 0 to -2 to confirm your analytical prediction.

This page titled 8.3: Hopf Bifurcations in 2-D Continuous-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

8.4: Bifurcations in Discrete-Time Models

The bifurcations discussed above (saddle-node, transcritical, pitchfork, Hopf) are also possible in discrete-time dynamical systems with one variable:

$$x_t = F(x_{t-1}) \tag{8.4.1}$$

The Jacobian matrix of this system is, again, a 1×1 matrix whose eigenvalue is its content itself, which is given by dF/dx. Since this is a discrete-time model, the critical condition at which a bifurcation occurs is given by

$$\left|\frac{dF}{dt}\right|_{x=x_{eq}} = 1. \tag{8.4.2}$$

Let's work on the following example:

$$x_t = x_{t-1} + r - x_{t-1}^2 \tag{8.4.3}$$

This is a discrete-time analog of Eq. (8.1.3). Therefore, it has the same set of equilibrium points:

$$x_{eq} = \pm\sqrt{r} \tag{8.4.4}$$

Next, we calculate dF/dx as follows:

$$\frac{dF}{dx} = (r + x - x^2)' \tag{8.4.5}$$

$$= 1 - 2x \tag{8.4.6}$$

$$\left|\frac{dF}{dt}\right|_{x=\pm\sqrt{r}} = |1 \pm 2\sqrt{r}| \tag{8.4.7}$$

To find out which value of r makes this 1, we consider the following four scenarios:

- $1 + 2\sqrt{r} = 1 \Rightarrow r = 0$
- $1 - 2\sqrt{r} = 1 \Rightarrow r = 0$
- $1 + 2\sqrt{r} = -1 \Rightarrow (no\ real\ solution)$
- $1 - 2\sqrt{r} = -1 \Rightarrow r = 1$

As you see, $r = 0$ appears as a critical threshold again, at which a saddle-node bifurcation occurs. But now we see another critical threshold, $r = 1$, also showing up, which was not there in its continuous-time version. This is why I said before that continuous-time models are simpler than discrete-time \ models for bifurcation analysis; *discrete-time models can show a new form of bifurcation that wasn't possible in their continuous-time counterparts*. To learn more about this, we can study the stability of the two equilibrium points by checking whether $|dF/dx|$ is greater or less than 1 at each point. The result is summarized in Table 8.4.1.

Table 8.4.1: Summary of bifurcation analysis of $x_t = x_{t-1} + r - x_{t-1}^2$.

Equilibrium point	$r < 0$	$0 < r < 1$	$1 < r$
$x_{eq} = \sqrt{r}$	doesn't exist	stable	unstable
$x_{eq} = -\sqrt{r}$	doesn't exist	unstable	unstable

The result shown in the table is pretty much the same as before up to $r = 1$, but when $r > 1$, *both equilibrium points become unstable*. This must mean that the system is not allowed to converge toward either of them. Then what is going to happen to the system in this parameter range? We can let the system show its actual behavior by numerical simulations. Here is an example:

Code 8.3: period-doubling-bifurcation.py

```
from pylab import *

def initialize():
    global x, result
    x = 0.1
    result = [x]

def observe():
    global x, result
    result.append(x)

def update():
    global x, result
    x = x + r - x**2

def plot_phase_space():
    initialize()
    for t in xrange(30):
        update()
        observe()
    plot(result)
    ylim(0, 2)
    title('r = ' + str(r))

rs = [0.1, 0.5, 1.0, 1.1, 1.5, 1.6]
for i in xrange(len(rs)):
    subplot(2, 3, i + 1)
    r = rs[i]
    plot_phase_space()

show()
```

The results are shown in Fig. 8.4.1, visualizing actual trajectories of the system's state over time for several values of r. The cobweb plot version is also shown in Fig. 8.4.2 (the code is not shown here; try implementing the code yourself!).

There are some new behaviors that were not seen in continuous-time versions of the same model. For example, at $r = 0.5$, there was a signature of "overshooting" right before the system state converged to a stable equilibrium point. But more importantly, at $r = 1$, the system began to fail to converge to a single stable point, and for $r > 1$, it started oscillating between two distinct states. This is called a $period - doubling\ bifurcation$. It is a bifurcation that typically occurs in discrete-time systems (but it is also possible in continuous-time systems of higher dimensions), where the system loses stability of a period T trajectory and begins to move in

another trajectory with period $2T$. The bifurcation observed for $r = 1$ was a transition from an equilibrium point (which is a periodic trajectory with period $T = 1$, i.e., the system takes the same state value in each time step)

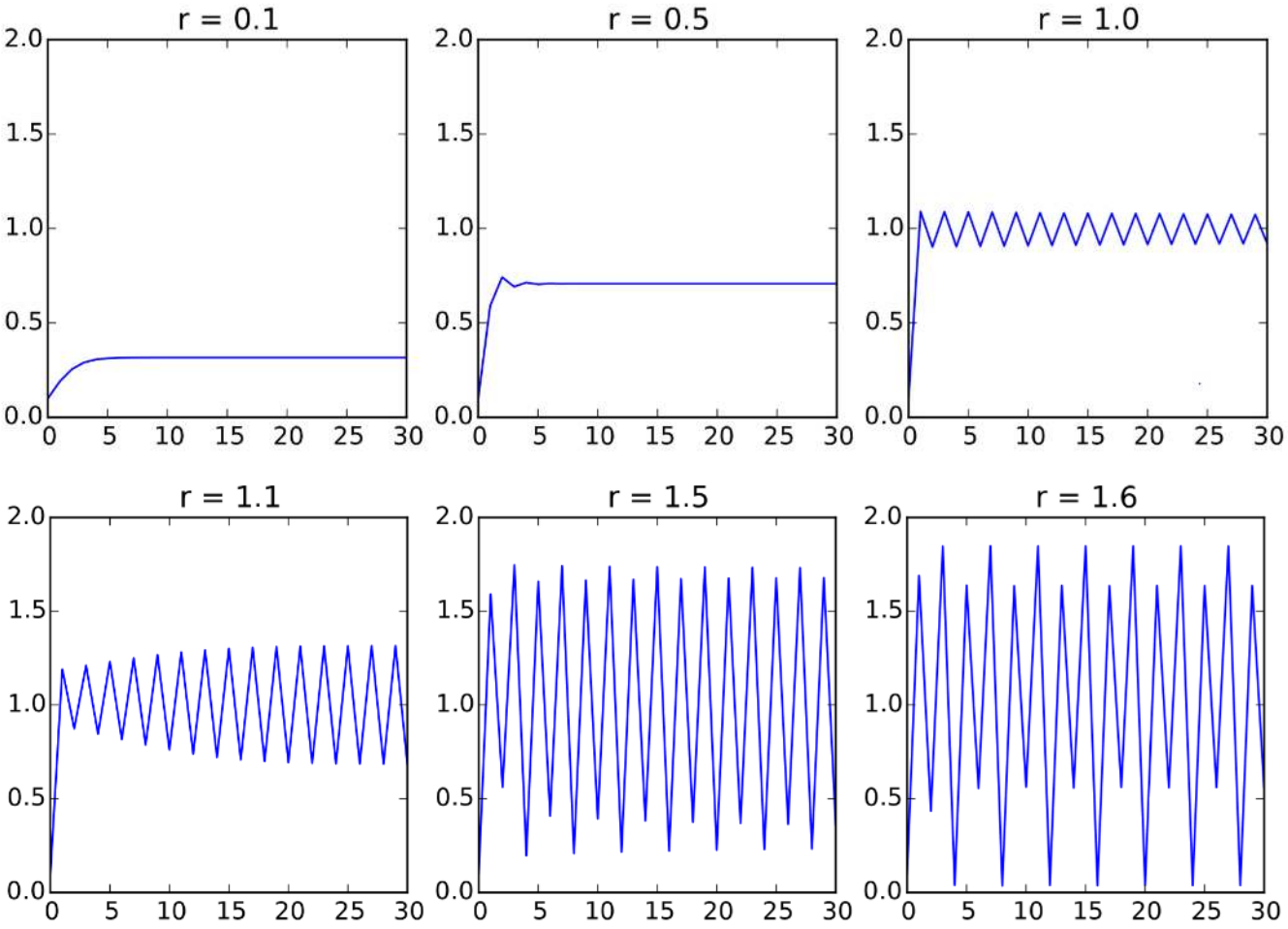
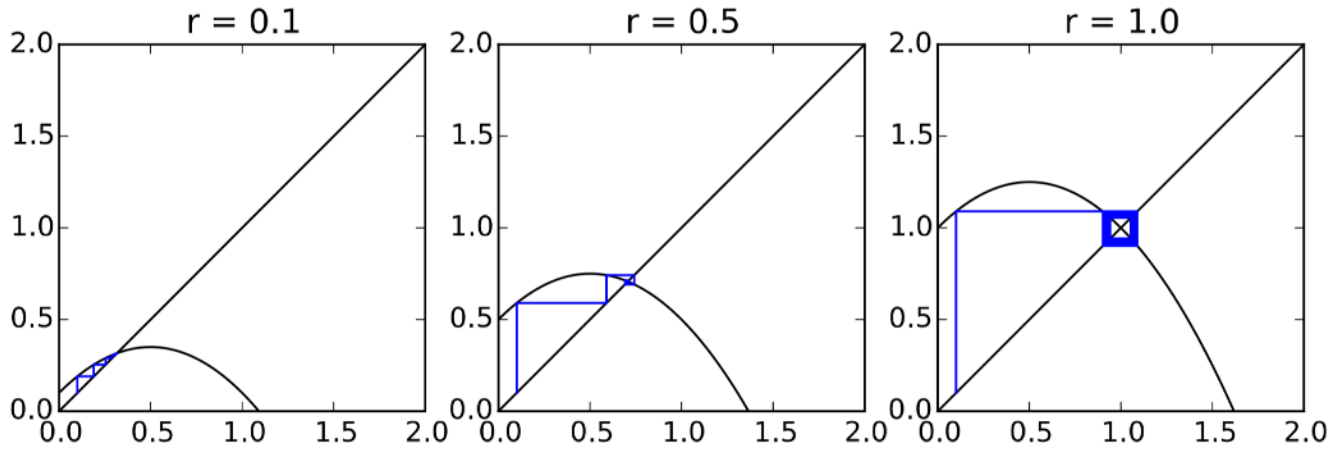


Figure 8.4.1: *Visual output of Code 8.3.*



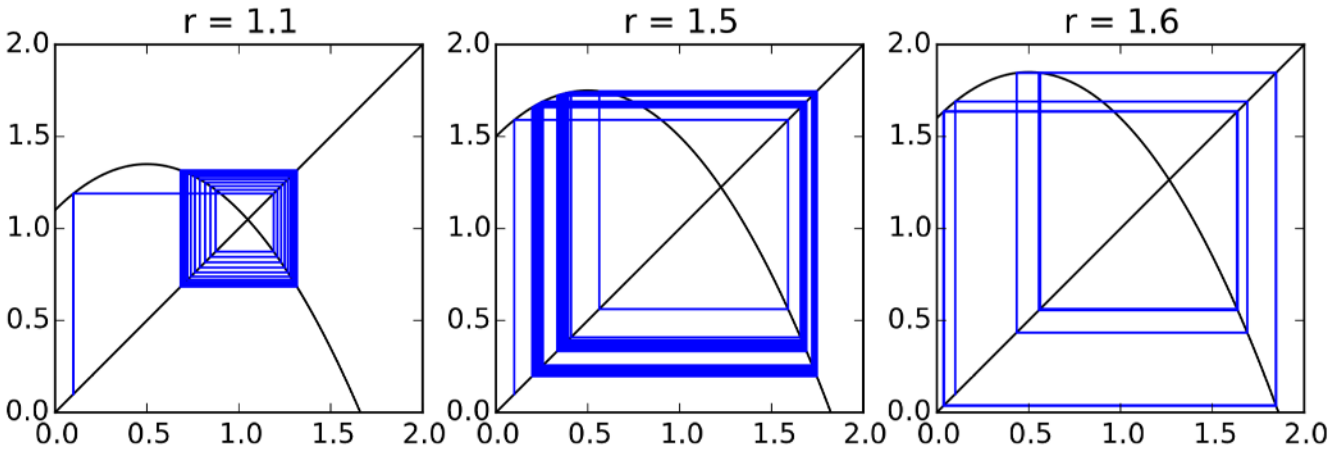


Figure 8.4.2: *Cobweb plot version of* Fig. 8.4.1.

to a trajectory with period $2T = 2$. Interestingly, the period-doubling bifurcations happen in a cascade if you keep increasing r. In this particular case, another period-doubling bifurcation was observed at $r = 1.5$, where the period-2 trajectory lost its stability and the system moved to a period-4 trajectory. This continues as you continue to increase r. The first period-doubling bifurcation from period-1 to period-2 trajectories can still be characterized as the loss of stability of an equilibrium point, but the dominant eigenvalue destabilizing the equilibrium point must be ***negative*** in order to induce the flipping behavior. This can be mathematically written as follows:

A first period-doubling bifurcation from period-1 to period-2 trajectories occurs in a discrete-time model when the eigenvalues λ_i of the Jacobian matrix at an equilibrium point satisfy the following:

$\lambda_i = -1$ for some i, while$|\lambda_i| < 1$ for the rest.

In the example above, $dF/dx|_{x_{eq}} = \sqrt{r} = -1$ when $r = 1$, which triggers the first period doubling bifurcation.

? Exercise 8.4.1

Consider the following discrete-time dynamical system:

$$x_t = (1 - a)x_{t-1} + ax_{t-1}^3 \tag{8.4.8}$$

This equation has $x_{eq} = 0$ as an equilibrium point. Obtain the value of a at which this equilibrium point undergoes a first period-doubling bifurcation.

Once the system begins to show period-doubling bifurcations, its asymptotic states are no longer captured by the locations of analytically obtained equilibrium points, as drawn in bifurcation diagrams (e.g., Figs. 8.2.1, 8.2.3, etc.). However, there is still a way to visualize bifurcation diagrams numerically by simulating the behavior of the system explicitly and then collecting the actual states the system visits for a certain period of time. Then we can plot their distributions in a diagram. The data points should be collected after a sufficiently long initial transient time has passed in each simulation, so that the system's trajectory is already showing its "final" behavior. Here is a sample code showing how to draw such a bifurcation diagram numerically:

Code 8.4: bifurcation-diagram-numerical.py

```
from pylab import *
```

```
def initialize():
    global x, result
    x = 0.1
    result = []

def observe():
    global x, result
    result.append(x)
```

```
def update():
    global x, result
    x = x + r - x**2

def plot_asymptotic_states():
    initialize()
    for t in xrange(100): # first 100 steps are discarded
        update()
    for t in xrange(100): # second 100 steps are collected
        update()
        observe()
    plot([r] * 100, result, 'b.', alpha = 0.3)

for r in arange(0, 2, 0.01):
    plot_asymptotic_states()

xlabel('r')
ylabel('x')
show()
```

In this code, r is gradually varied from 0 to 2 at intervals of 0.01. For each value of r, the model (Equation 8.4.3) is simulated for 200 steps, and only the second half of the state values are recorded in `result`. Once the simulation is finished, the states stored in `result` are plotted at a particular value of `r` in the plot (note that the expression `[r] * 100` in Python produces a list of one hundred `r`'s). The alpha option is used to make the markers transparent so that the densities of markers are also visible.

The result is shown in Fig. 8.4.3. This bifurcation diagram still shows how the system's state depends on the parameter, but what is plotted here are no longer analytically obtained equilibrium points but numerically sampled sets of asymptotic states, i.e., where the system is likely to be after a sufficiently long period of time. If the system is converging to a stable equilibrium point, you see one curve in this diagram too. Unstable equilibrium points never show up because they are never realized in numerical simulations. Once the system undergoes period-doubling bifurcations, the states in a periodic trajectory are all sampled during the sampling period, so they all appear in this diagram. The period doubling bifurcations are visually seen as the branching points of those curves.

But what are those noisy, crowded, random-looking parts at $r > 1.7$? That is chaos, the hallmark of nonlinear dynamics. We will discuss what is going on there in the next chapter.

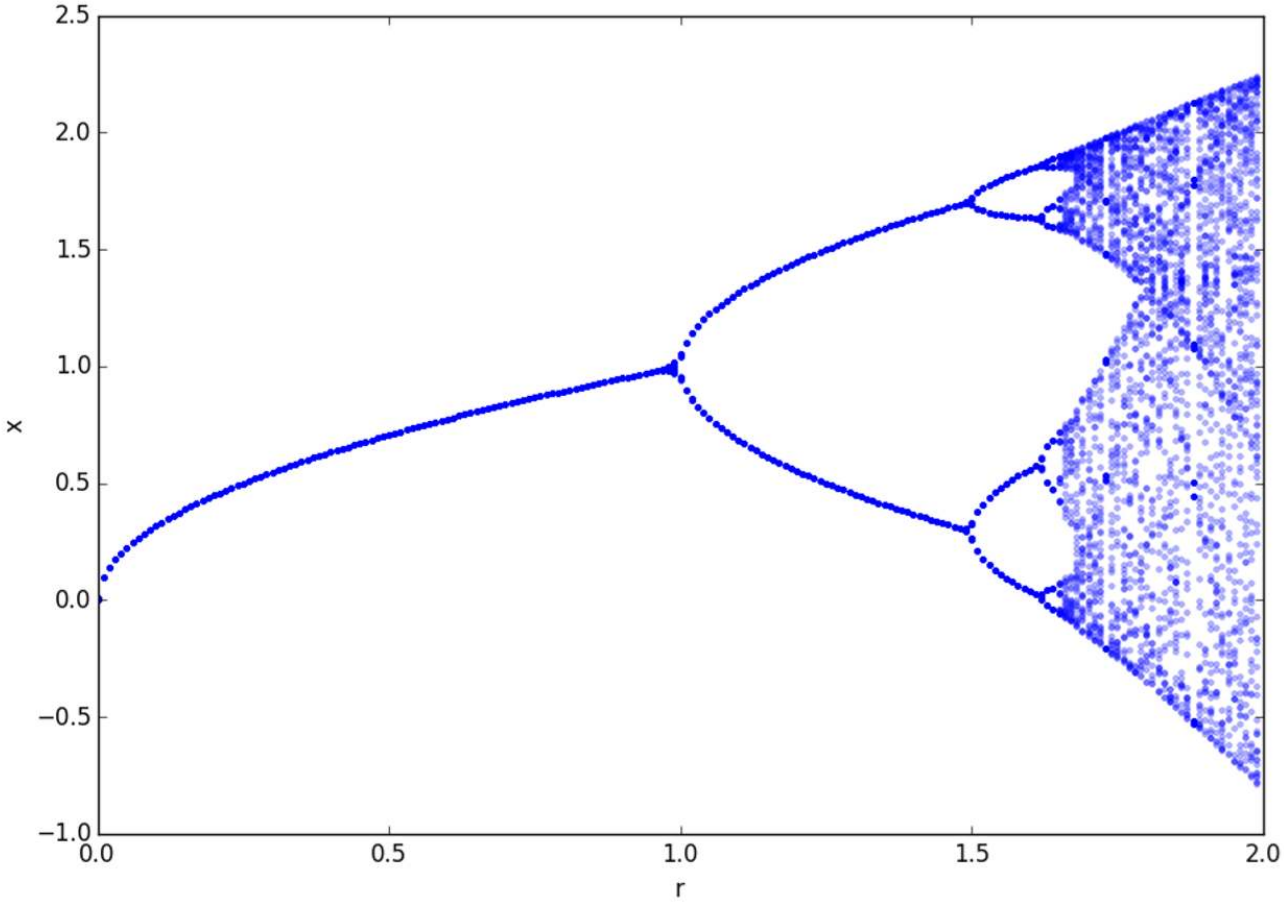


Figure 8.4.3: *Visual output of Code 8.4, showing the numerically constructed bifurcation diagram of Equation 8.4.3.*

? Exercise 8.4.2: Logistic map

Period-doubling bifurcations and chaos are not just for abstract, contrived mathematical equations, but they can occur in various models of real-world biological, ecological, social, and engineering phenomena. The simplest possible example would be the logistic map we introduced in Section 5.5:

$$x_t = r x_{t-1}(1 - x_{t-1}) \tag{8.4.9}$$

This is a mathematical model of population dynamics, where x_t represents the population of a species that reproduce in discrete (non-overlapping) generations. This model was used by British/Australian mathematical ecologist Robert May in his influential 1976 Nature paper [32] to illustrate how a very simple mathematical model could produce astonishingly complex behaviors.

- Conduct a bifurcation analysis of this model to find the critical thresholds of r at which bifurcations occur.
- Study the stability of each equilibrium point in each parameter range and summarize the results in a table.
- Simulate the model with several selected values of r to confirm the results of analysis.
- Draw a bifurcation diagram of this model for $0 < r < 4$.

? Exercise 8.4.3: Stability analysis of periodic trajectories

The stability of a period-2 trajectory of a discrete-time model $x_t = F(x_{t-1})$ can be studied by the stability analysis of another model made of a composite function of F:

$$y_r = G(y_{r-1}) = F(F(t_{r-1})) \tag{8.4.10}$$

This is because the period-2 trajectory in F corresponds to one of the equilibrium points of $G(\circ) = F(F(\circ))$. If such an equilibrium point of G is being destabilized so that $dG/dx \approx -1$, it means that the period-2 trajectory in question is losing the stability, and thus another period-doubling bifurcation into a period-4 trajectory is about to occur. Using this technique, analytically obtain the critical threshold of r in Equation 8.4.3 at which the second period-doubling bifurcation occurs (from period 2 to period 4). Then, draw cobweb plots of $y_t = G(y_{t-1})$ for several values of r near the critical threshold to see what is happening there.

This page titled 8.4: Bifurcations in Discrete-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

9: Chaos

9.1: Chaos in Discrete-Time Models

9.2: Characteristics of Chaos

9.3: Lyapunov Exponent

9.4: Chaos in Continuous-Time Model

This page titled 9: Chaos is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

9.1: Chaos in Discrete-Time Models

Figure 8.4.3 showed a cascade of period-doubling bifurcations, with the intervals between consecutive bifurcation thresholds getting shorter and shorter geometrically as r increased. This cascade of period doubling eventually leads to the divergence of the period to infinity at $r \approx 1.7$ in this case,which indicates the onset of chaos. In this mysterious parameter regime,the system loses any finite-length periodicity, and its behavior looks essentially random. Figure 9.1.1 shows an example of such chaotic behavior of Eq.(8.4.3) with $r = 1.8$.

So what is chaos anyway? It can be described in a number of different ways, as follows:

Properties of Chaos

- Chaos is a long-term behavior of a nonlinear dynamical system that never falls in any static or periodic trajectories.
- Chaos looks like a random fluctuation, but still occurs in completely deterministic, simple dynamical systems.
- Chaos exhibits sensitivity to initial conditions.
- Chaos occurs when the period of the trajectory of the system's state diverges to infinity.
- Chaos occurs when no periodic trajectories are stable.

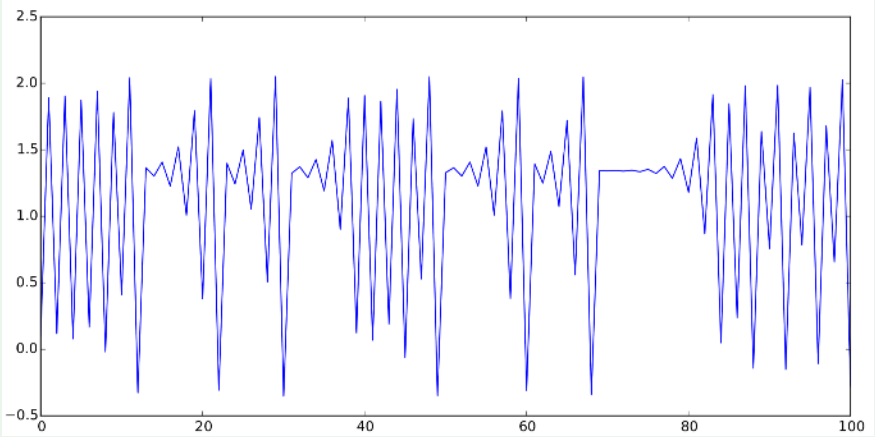


Figure 9.1.1: Example of chaotic behavior, generated using Eq. (8.4.3) with r = 1.8 *and the initial condition* \(x_0\) = 0.1.

- Chaos is a prevalent phenomenon that can be found everywhere in nature, as well as in social and engineered environments.

The sensitivity of chaotic systems to initial conditions is particularly well known under the moniker of the "*butterfly effect*," which is a metaphorical illustration of the chaotic nature of the weather system in which "a flap of a butterfly's wings in Brazil could set off a tornado in Texas." The meaning of this expression is that, in a chaotic system, a small perturbation could eventually cause very large-scale difference in the long run. Figure 9.1.2 shows two simulation runs of Eq. (*8.4.3*) with $r = 1.8$ and two slightly different initial conditions, $x_0 = 0.1$ and $x_0 = 0.100001$. The two simulations are fairly similar for the first several steps, because the system is fully deterministic (this is why weather forecasts for just a few days work pretty well). But the "flap of the butterfly's wings" (the 0.000001 difference) grows eventually so big that it separates the long-term fates of the two simulation runs. Such extreme sensitivity of chaotic systems makes it practically impossible for us to predict exactly their long-term behaviors (this is why there are no two-month weather forecasts[1]).

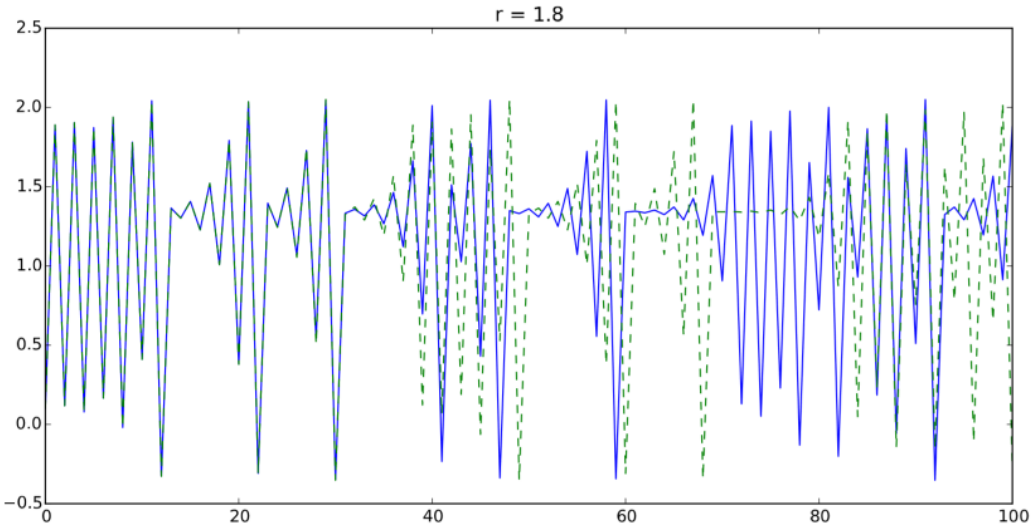


Figure **9.1.2**: *Example of the "butterfly effect," the extreme sensitivity of chaotic systems to initial conditions. The two curves show time series generated using Eq. (8.4.3) with* $r = 1.8$*; one with* $x_0 = 0.1$ *and the other with* $x_0 = 0.100001$.

? Exercise 9.1.2

There are many simple mathematical models that exhibit chaotic behavior. Try simulating each of the following dynamical systems (shown in Fig. 9.1.3). If needed, explore and find the parameter values with which the system shows chaotic behaviors.

- Logistic map: $x_t = rx_{t-1}(1 - x_{t-1})$
- Cubic map: $x_t = x_{t-1}^3 - rx_{t-1}$
- Sinusoid map: $x_t = rsinx_{t-1}$
- Saw map: $x_t =$ fractional part of $2x_{t-1}$

Note: The saw map may not show chaos if simulated on a computer,but it will show chaos if it is manually simulated on a cobweb plot. This issue will be discussed later.

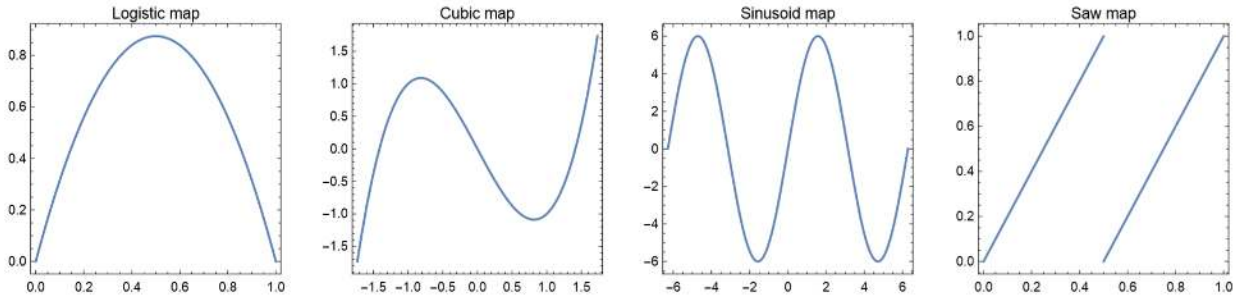


Figure **9.1.3**: *Simple maps that show chaotic behavior (for Exercise 9.1.1).*

[1]But this doesn't necessarily mean we can't predict climate change over longer time scales. What is not possible with a chaotic system is the prediction of the exact long-term behavior, e.g., when, where, and how much it will rain over the next 12 months. It is possible, though, to model and predict long-term changes of a system's statistical properties, e.g., the average temperature of the global climate, because it can be described well in a much simpler, non-chaotic model. We shouldn't use chaos as an excuse to avoid making predictions for our future!

This page titled 9.1: Chaos in Discrete-Time Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

9.2: Characteristics of Chaos

It is helpful to realize that there are two dynamical processes always going on in any kind of chaotic systems: *stretching and folding* [33]. Any chaotic system has a dynamical mechanism to stretch, and then fold, its phase space, like kneading pastry dough (Figure **9.2.1**). Imagine that you are keeping track of the location of a specific grain of flour in the dough while a pastry chef kneads the dough for a long period of time. Stretching the dough magnifies the tiny differences in position at microscopic scales to a larger, visible one, while folding the dough always keeps its extension within a finite, confined size. Note that folding is the primary source of nonlinearity that makes long-term predictions so hard—if the chef were simply stretching the dough all the time (which would look more like making ramen), you would still have a pretty good idea about where your favorite grain of flour would be after the stretching was completed. This stretching-and-folding view allows us to make another interpretation of chaos:

Stretching and Folding Phase Space

Chaos can be understood as a dynamical process in which microscopic information hidden in the details of a system's state is dug out and expanded to a macroscopically visible scale (*stretching*), while the macroscopic information visible in the current system's state is continuously discarded (*folding*).

Jul 24, 2018, 8:45 PM

http://beyondmicrofoundations.blogsp...2012-csss.html

https://elmer.unibas.ch/pendulum/lab.htm

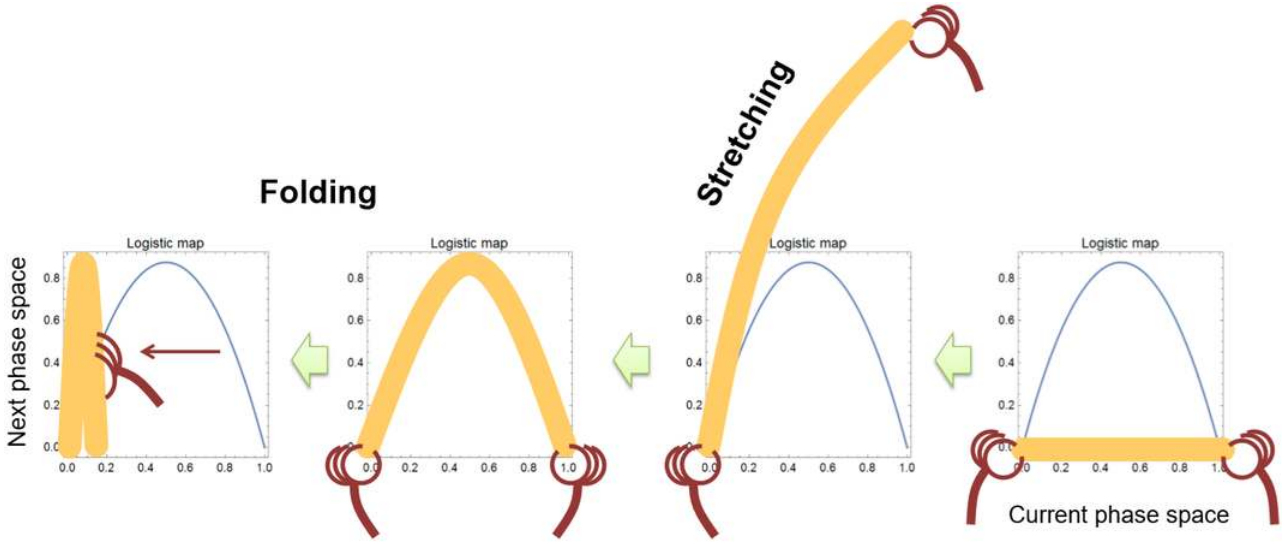


Figure **9.2.1**: Schematic illustration of strechting and folding taking place in a chaotic iterative map. The current phase space (right) is first stretched (middle right) and then folded (middle left and left) to generate the next phase spectra in each iteration.

This kind of information flow-based explanation of chaos is quite helpful in understanding the essence of chaos from a multiscale perspective. This is particularly clear when you consider the saw map discussed in the previous exercise:

$$x_t = \text{fractional part of } 2x_{t-1} \tag{9.2.1}$$

If you know binary notations of real numbers, it should be obvious that this iterative map is simply shifting the bit string in x always to the left, while forgetting the bits that came before the decimal point. And yet, such a simple arithmetic operation can still create chaos, if the initial condition is an irrational number (Figure **9.2.2**)! This is because an irrational number contains an infinite length of digits and chaos continuously digs them out to produce a fluctuating behavior at a visible scale.

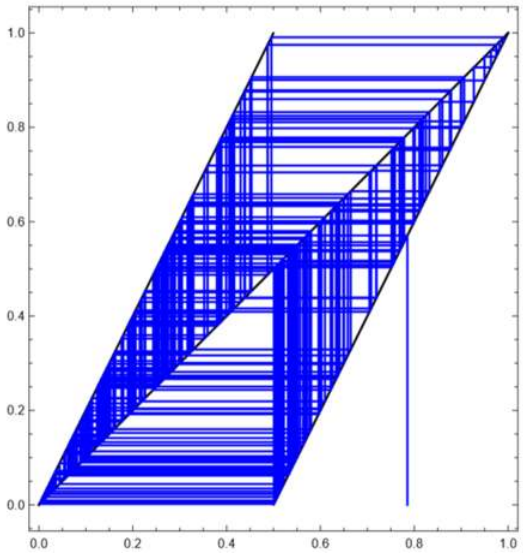


Figure **9.2.1**: Cobweb plot for the saw map with $\pi/4$ as the initial condition.

? Exercise 9.2.1

The saw map can also show chaos even from a rational initial condition, if its behavior is manually simulated by hand on a cobweb plot. Explain why.

This page titled 9.2: Characteristics of Chaos is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

9.3: Lyapunov Exponent

Finally, I would like to introduce one useful analytical metric that can help characterize chaos. It is called the Lyapunov exponent, which measures how quickly an infinitesimally small distance between two initially close states grows over time:

$$F^t(x_0+\varepsilon) - F^t(x_0) \approx \varepsilon e^{\lambda t} \tag{9.3.1}$$

The left hand side is the distance between two initially close states after t steps, and the right hand side is the assumption that the distance grows exponentially over time. The exponent λ measured for a long period of time (ideally $t \to \infty$) is the Lyapunov exponent. If $\lambda > 0$, small distances grow indefinitely over time, which means the stretching mechanism is in effect. Or if $\lambda < 0$, small distances don't grow indefinitely, i.e., the system settles down into a periodic trajectory eventually. Note that the Lyapunov exponent characterizes only stretching, but as we discussed before, stretching is not the only mechanism of chaos. You should keep in mind that the folding mechanism is not captured in this Lyapunov exponent.

We can do a little bit of mathematical derivation to transform Equation 9.3.1 into a more easily computable form:

$$e^{\lambda t} \approx \frac{|F^t(x_0+\varepsilon) - F^t(x_0)|}{\varepsilon} \tag{9.3.2}$$

$$\lambda = \lim_{t\to\infty,\varepsilon\to 0} \frac{1}{t}\log\frac{|F^t(x_0+\varepsilon) - F^t(x_0)|}{\varepsilon} \tag{9.3.3}$$

$$= \lim_{t\to\infty 1,\varepsilon\to 0} \frac{1}{t}\log\left|\frac{dF}{dx}|_{x=x_0}\right| \tag{9.3.4}$$

(applying the chain rule of differentiation...)

$$\lambda = \lim_{t\to\infty} \frac{1}{t}\log\left|\frac{dF}{dx}|_{x=F^{t-1}(x_0)=x_{t-1}} \cdot \frac{dF}{dx}|_{x=F^{t-2}(x_0)=x_{t-2}} \cdots \frac{dF}{dx}|_{x=x_0}\right| \tag{9.3.5}$$

$$\lim_{t\to\infty} \frac{1}{t}\sum_{i=0}^{t-1}\log\left|\frac{dF}{dx}|_{x=x_i}\right| \tag{9.3.6}$$

The final result is quite simple—the Lyapunov exponent is a time average of $log|\frac{dF}{dx}|$ at every state the system visits over the course of the simulation. This is very easy to compute numerically. Here is an example of computing the Lyapunov exponent of Eq. 8.4.3 over varying r:

Code 9.1: Lyapunov-exponent.py

```
from pylab import *

def initialize():
    global x, result
    x = 0.1
    result = [logdFdx(x)]

def observe():
    global x, result
    result.append(logdFdx(x))

def update():
    global x, result
    x = x + r - x**2
```

```
def logdFdx(x):
    return log(abs(1 - 2*x))

def lyapunov_exponent():
    initialize()
    for t in xrange(100):
        update()
        observe()
    return mean(result)

rvalues = arange(0, 2, 0.01)
lambdas = [lyapunov_exponent() for r in rvalues]
plot(rvalues, lambdas)
plot([0, 2], [0, 0])

xlabel('r')
ylabel('Lyapunov exponent')
show()
```

Figure 9.3.1 shows the result. By comparing this figure with the bifurcation diagram (Fig. 8.4.3), you will notice that the parameter range where the Lyapunov exponent takes positive values nicely matches the range where the system shows chaotic behavior. Also, whenever a bifurcation occurs (e.g., $r = 1$, 1.5, etc.), the Lyapunov exponent touches the $\lambda = 0$ line, indicating the criticality of those parameter values. Finally, there are several locations in the plot where the Lyapunov exponent diverges to negative infinity (they may not look so, but they are indeed going infinitely deep). Such values occur when the system converges to an extremely stable equilibrium point with $\frac{dF^t}{dx}|_{x=x_0} \approx 0$ for certain t. Since the definition of the Lyapunov exponent contains logarithms of this derivative, if it becomes zero, the exponent diverges to negative infinity as well.

? Exercise 9.3.1

Plot the Lyapunov exponent of the logistic map (Eq. (\ref{8.4.9})) for $0 < r < 4$, and compare the result with its bifurcation diagram.

? Exercise 9.3.2

Plot the bifurcation diagram and the Lyapunov exponent of the following discrete-time dynamical system for $r > 0$:

$$x_t = cos^2(rx_{t-1}) \tag{9.3.7}$$

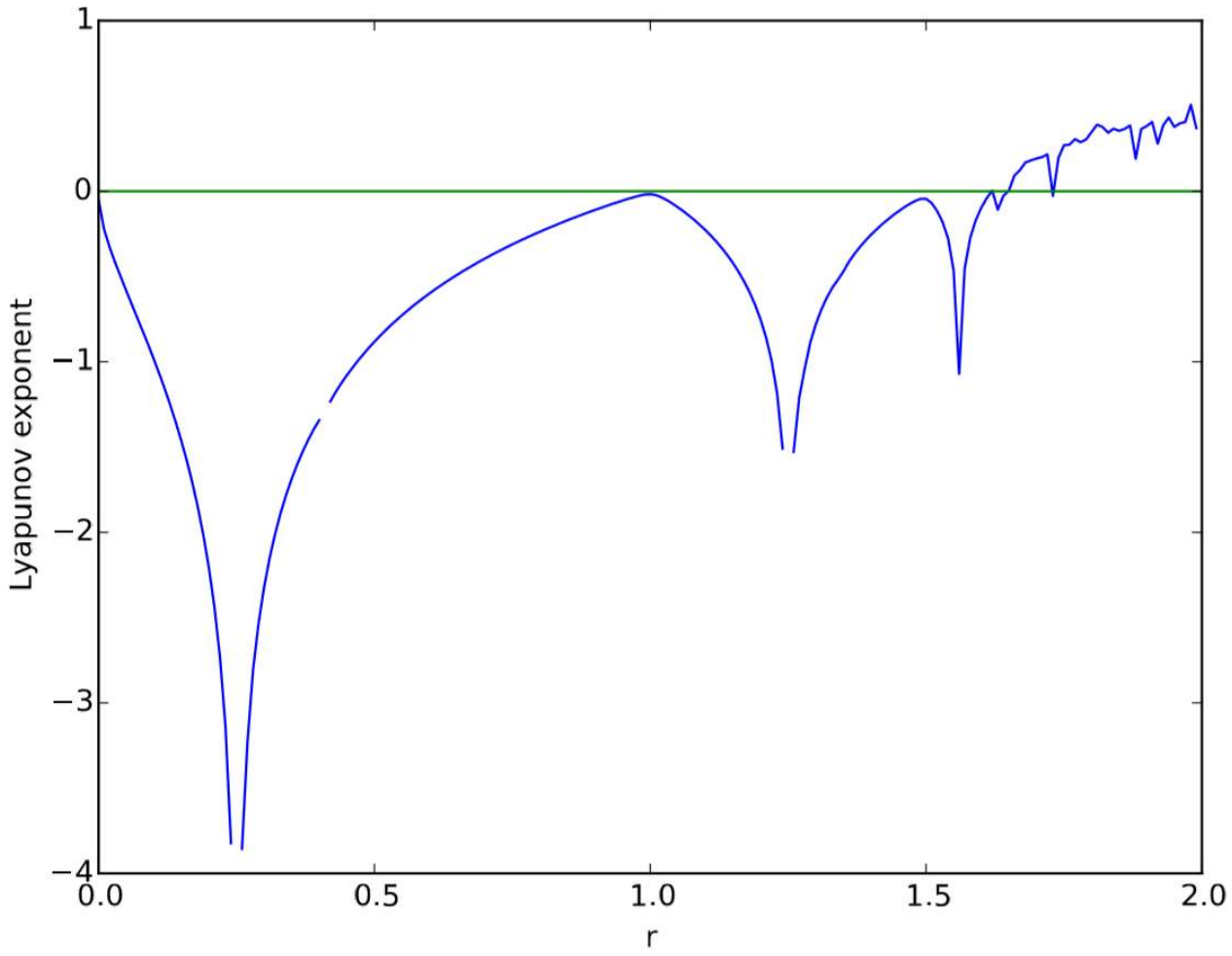


Figure **9.3.1**: Visual output of Code 9.1, showing the Lyapunov exponent of Eq. (8.4.3) *over varying* r. *Compare this with* *Fig. 8.4.3*.

Then explain in words how its dynamics change over r.

This page titled 9.3: Lyapunov Exponent is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

9.4: Chaos in Continuous-Time Model

As we reviewed above, chaos is really easy to show in discrete-time models. But the discovery of chaos was originally made with continuous-time dynamical systems, i.e., differential equations. Edward Lorenz, an American mathematician and meteorologist, and one of the founders of chaos theory, accidentally found chaotic behavior in the following model (called the *Lorenz equations*) that he developed to study the dynamics of atmospheric convection in the early 1960s [5]:

$$\frac{dx}{dt} = s(y - x) \tag{9.4.1}$$

$$\frac{dy}{dt} = rx - y - xz \tag{9.4.2}$$

$$\frac{dz}{dt} = xy - bz \tag{9.4.3}$$

Here s, r, and b are positive parameters.This model is known to be one of the first that can show chaos in continuous time. Let's simulate this model with $s = 10$, $r = 30$, and $b = 3$, for example:

Code 9.2: Lorenz-equations.py

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

s = 10.
r = 30.
b = 3.
Dt = 0.01

def initialize():
    global x, xresult, y, yresult, z, zresult, t, timesteps
    x = y = z = 1.
    xresult = [x]
    yresult = [y]
    zresult = [z]
```

```
    t = 0.
    timesteps = [t]

def observe():
    global x, xresult, y, yresult, z, zresult, t, timesteps
    xresult.append(x)
    yresult.append(y)
    zresult.append(z)
    timesteps.append(t)

def update():
    global x, xresult, y, yresult, z, zresult, t, timesteps
    nextx = x + (s * (y - x)) * Dt
    nexty = y + (r * x - y - x * z) * Dt
    nextz = z + (x * y - b * z) * Dt
```

```
    x, y, z = nextx, nexty, nextz
    t = t + Dt

initialize()
while t < 30.:
    update()
    observe()

subplot(3, 1, 1)
plot(timesteps, xresult)
xlabel('t')
ylabel('x')

subplot(3, 1, 2)
plot(timesteps, yresult)
```

```
xlabel('t')
ylabel('y')

subplot(3, 1, 3)
plot(timesteps, zresult)
xlabel('t')
ylabel('z')
```

```
figure()
ax = gca(projection='3d')
ax.plot(xresult, yresult, zresult, 'b')

show()
```

This code produces two outputs: one is the time series plots of `x`, `y`, and `z` (Fig. 9.4.1), and the other is the trajectory of the system's state in a 3-D phase space (Fig. 9.4.2). As you can see in Fig. 9.4.1, the behavior of the system is highly unpredictable, but there is definitely some regularity in it too. `x` and `y` tend to stay on either the positive or negative side, while showing some oscillations with growing amplitudes. When the oscillation becomes too big, they are thrown to the other side. This continues indefinitely, with occasional switching of sides at unpredictable moments. In the meantime, `z` remains positive all the time, with similar oscillatory patterns.

Plotting these three variables together in a 3-D phase space reveals what is called the *Lorenz attractor* (Fig. 9.4.2). It is probably the best-known example of *strange attractors*, i.e., attractors that appear in phase spaces of chaotic systems.

Just like any other attractors, strange attractors are sets of states to which nearby trajectories are attracted. But what makes them really "strange" is that, even if they look like a bulky object, their "*volume*" is zero relative to that of the phase space, and thus they have a *fractal dimension*, i.e., a dimension of an object that is not integer-valued. For example, the Lorenz attractor's fractal dimension is known to be about 2.06, i.e., it is pretty close to a 2-D object but not quite. In fact, any chaotic system has a strange attractor with fractal dimension in its phase space. For example, if you carefully look at the intricate patterns in the chaotic regime of Fig. 8.4.3, you will see fractal patterns there too.

? Exercise 9.4.1

Draw trajectories of the states of the Lorenz equations in a 3-D phase space for several different values of r while other parameters are kept constant. See how the dynamics of the Lorenz equations change as you vary r.

? Exercise 9.4.2

Obtain the equilibrium points of the Lorenz equations as a function of r, while keeping $s = 10$ and $b = 3$. Then conduct a bifurcation analysis on each equilibrium point to find the critical thresholds of r at which a bifurcation occurs.

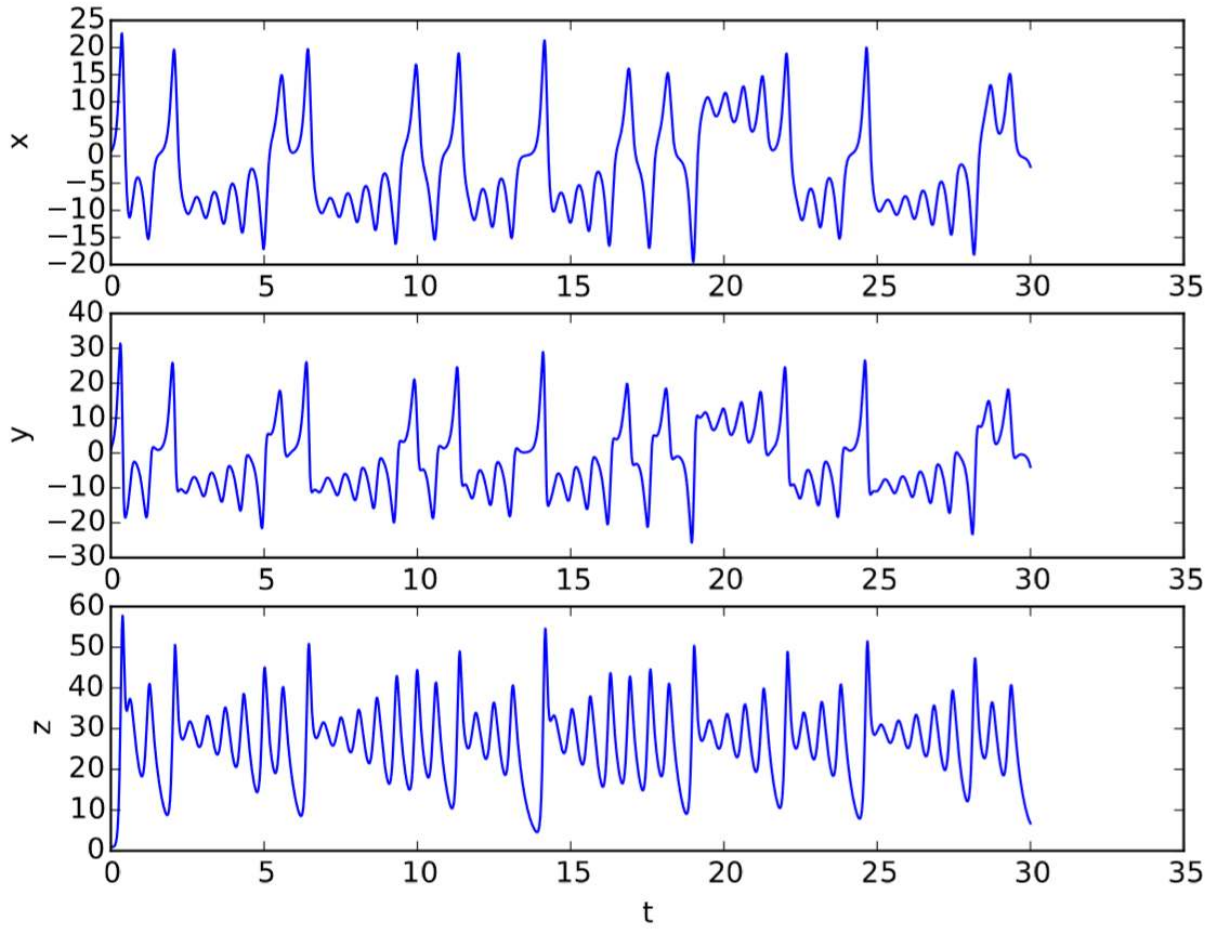


Figure **9.4.1**: *Visual output of Code 9.2 (1): Time series plots of* x, y, *and* z.

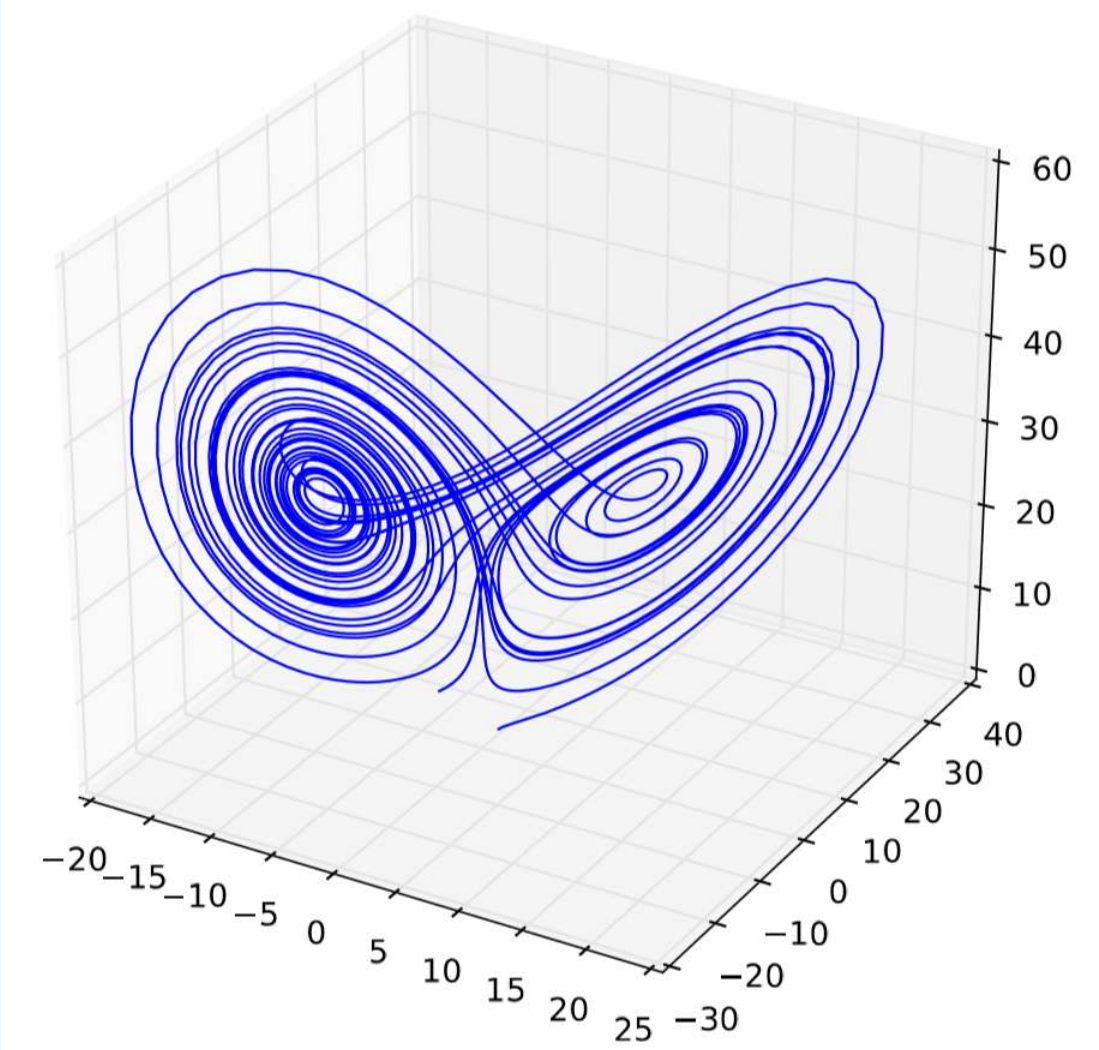


Figure **9.4.2**: *Visual output of Code 9.2 (2): Trajectory of the state of the Lorenz equations in a 3-D phase space.*

Compare your result with numerical simulation results obtained in the previous exercise.

By the way, I said before that any chaotic system has two dynamical processes: stretching and folding. Where do these processes occur in the Lorenz attractor? It is not so straightforward to fully understand its structure, but the stretching occurs where the trajectory is circling within one of the two "wings" of the attractor. The spirals seen on those wings are unstable ones going outward, so the distance between initially close states are expanded as they circle around the spiral focus. In the meantime, the folding occurs at the center of the attractor, where two "sheets" of trajectories meet. Those two sheets actually never cross each other, but they keep themselves separated from each other, forming a "wafer sandwich" made of two thin layers, whose right half goes on to circle in the right wing while the left half goes on to the left wing. In this way, the "dough" is split into two, each of which is stretched, and then the two stretched doughs are stacked on top of each other to form a new dough that is made of two layers again. As this process continues, the final result, the Lorenz attractor, acquires infinitely many, recursively formed layers in it, which give it the name of a "strange" attractor with a fractal dimension.

? Exercise 9.4.3

Plot the Lorenz attractor in several different perspectives (the easiest choice would be to project it onto the $x-y$, $y-z$, and $x-z$ planes) and observe its structure. Interpret its shape and the flows of trajectories from a stretching-and-folding viewpoint.

I would like to bring up one more important mathematical fact before we close this chapter:

> In order for continuous-time dynamical systems to be chaotic, the dimensions of the system's phase space must be at least 3-D. In contrast, discrete-time dynamical systems can be chaotic regardless of their dimensions

The Lorenz equations involved three variables, so it was an example of continuous-time chaotic systems with minimal dimensionality.

This fact is derived from the *Poincar´e-Bendixson theorem* in mathematics, which states that no strange attractor can arise in continuous 2-D phase space. An intuitive explanation of this is that, in a 2-D phase space, every existing trajectory works as a "wall" that you can't cross, which imposes limitations on where you can go in the future. In such an increasingly constraining environment, it is not possible to maintain continuously exploring dynamics for an indefinitely long period of time.

? Exercise 9.4.4

Gather a pen and a piece of blank paper. Start drawing a continuous curve on the paper that represents the trajectory of a hypothetical dynamical system in a 2-D phase space. The shape of the curve you draw can be arbitrary, but with the following limitations:

- You can't let the pen go off the paper. The curve must be drawn in one continuous stroke.
- The curve can't merge into or cross itself.
- You can't draw curves flowing in opposing directions within a very tiny area (this violates the assumption of phase space continuity).

Keep drawing the curve as long as you can, and see what happens. Discuss the implications of the result for dynamical systems. Then consider what would happen if you drew the curve in a 3-D space instead of 2-D.

? Exercise 9.4.5

Let zi denote the value of the i-th peak of $z(t)$ produced by the Lorenz equations. Obtain time series data $z_1, z_2, z_3, \ldots$ from numerical simulation results. Plot z_t against z_{t-1}, like in a cobweb plot, and see what kind of structure you find there. Do this visualization for various values of r, while keeping $s = 10$ and $b = 3$, and compare the results with the results of the bifurcation analysis obtained in Exercise 9.4.1.

As reviewed through this and previous chapters, bifurcations and chaos are the most distinctive characteristics of nonlinear systems. They can produce unexpected system behaviors that are often counter-intuitive to our everyday understanding of nature. But once you realize the possibility of such system behaviors and you know how and when they can occur, your view of the world becomes a more informed, enriched one. After all, bifurcations and chaos are playing important roles in our physical, biological, ecological, and technological environments (as well as inside our bodies and brains). They should thus deserve our appreciation.

This chapter concludes our journey through systems with a small number of variables. We will shift gears and finally go into the realm of complex systems that are made of a large number of variables in the next chapter.

This page titled 9.4: Chaos in Continuous-Time Model is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

10: Interactive Simulations of Complex Systems

10.1: Simulation of Systems with a Large Number of Variables

10.2: Interactive Simulation with PyCX

10.3: Interactive Parameter Control in PyCX

10.4: Simulation without PyCX

This page titled 10: Interactive Simulations of Complex Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

10.1: Simulation of Systems with a Large Number of Variables

We are finally moving into the modeling and analysis of *complex systems*. The number of variables involved in a model will jump drastically from just a few to tens of thousands! What happens if you have so many dynamical components, and moreover, if those components interact with each other in nontrivial ways? This is the core question of complex systems. Key concepts of complex systems, such as emergence and self-organization, all stem from the fact that a system is made of a massive amount of interactive components, which allows us to study its properties at various scales and how those properties are linked across scales.

Modeling and simulating systems made of a large number of variables pose some practical challenges. First, we need to know how to specify the dynamical states of so many variables and their interaction pathways, and how those interactions affect the states of the variables over time. If you have empirical data for all of these aspects, lucky you— you could just use them to build a fairly detailed model (which might not be so useful without proper abstraction, by the way). However, such detailed information may not be readily available, and if that is the case, you have to come up with some reasonable assumptions to make your modeling effort feasible. The modeling frameworks we will discuss in the following chapters (cellular automata, continuous field models, network models, and agent-based models) are, in some sense, the fruit that came out of researchers' collective effort to come up with "best practices" in modeling complex systems, especially with the lack of detailed information available (at least at the time when those frameworks were developed). It is therefore important for you to know explicit/implicit model assumptions and limitations of each modeling framework and how you can go beyond them to develop your own modeling framework in both critical and creative ways.

Another practical challenge in complex systems modeling and simulation is visualization of the simulation results. For systems made of a few variables, there are straightforward ways to visualize their dynamical behaviors, such as simple time series plots, phase space plots, cobweb plots, etc., which we discussed in the earlier chapters. When the number of variables is far greater, however, the same approaches won't work. You can't discern thousands of time series plots, or you can't draw a phase space of one thousand dimensions. A typical way to address this difficulty is to define and use a metric of some global characteristics of the system, such as the average state of the system, and then plot its behavior. This is a reasonable approach by all means, but it loses a lot of information about the system's actual state.

An alternative approach is to visualize the system's state at each time point in detail, and then *animate* it over time, so that you can see the behavior of the system without losing information about the details of its states. This approach is particularly effective if the simulation is *interactive*, i.e., if the simulation results are visualized on the fly as you operate the simulator. In fact, most complex systems simulation tools (e.g., NetLogo, Repast) adopt such *interactive simulation* as their default mode of operation. It is a great way to explore the system's behaviors and become "experienced" with various dynamics of complex systems.

This page titled 10.1: Simulation of Systems with a Large Number of Variables is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

10.2: Interactive Simulation with PyCX

We can build an interactive, dynamic simulation model in Python relatively easily, using PyCX's"pycxsimulator.py" file, which is available from `http://sourceforge.net/projects/ pycx/files/` (it is also directly linked from the file name above if you are reading this electronically). This file implements a simple interactive graphical user interface (GUI) for your own dynamic simulation model, which is still structured in the three essential components—initialization, observation, and updating—just like we did in the earlier chapters.

To use pycxsimulator.py, you need to place that file in the directory where your simulation code is located. Your simulation code should be structured as follows:

Code 10.1: interactive-template.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

# import necessary modules
# define model parameters

def initialize():
    global # list global variables
    # initialize system states

def observe():
    global # list global variables
    cla() # to clear the visualization space
    # visualize system states

def update():
    global # list global variables
    # update system states for one discrete time step

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

The first three lines and the last two lines should always be in place; no modification is needed.

Let's work on some simple example to learn how to use pycxsimulator.py. Here we build a model of a bunch of particles that are moving around randomly in a twodimensional space. We will go through the process of implementing this simulation model step by step.

First, we need to import the necessary modules and define the parameters. In this particular example, a Gaussian random number generator is useful to simulate the random motion of the particles. It is available in the `random` module of Python. There are

various parameters that are conceivable for this example. As an example, let's consider the number of particles and the standard deviation of Gaussian noise used for random motion of particles, as model parameters. We write these in the beginning of the code as follows:

Code 10.2:

```
import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise
```

Next is the initialization function. In this model, the variables needed to describe the state of the system are the positions of particles. We can store their x– and y– coordinates in two separate lists, as follows:

Code 10.3:

```
def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))
```

Here we generate `n` particles whose initial positions are sampled from a Gaussian distribution with mean 0 and standard deviation 1.

Then visualization. This is fairly easy. We can simply plot the positions of the particles as a scatter plot. Here is an example:

Code 10.4:

```
def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')
```

The last option, '.', in the `plot` function specifies that it draws a scatter plot instead of a curve. Finally, we need to implement the updating function that simulates the random motion of the particles. There is no interaction between the particles in this particular simulation model, so we can directly update `xlist` and `ylist` without using things like `nextxlist` or `nextylist` :

Code 10.5:

```
def update():
    global xlist, ylist
```

```
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)
```

Here, a small Gaussian noise with mean 0 and standard deviation `sd` is added to the x– and y–coordinates of each particle in every time step. Combining these components, the completed code looks like this:

Code 10.6: random-walk-2D.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise

def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))

def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')

def update():
    global xlist, ylist
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

Run this code, and you will find two windows popping up (Fig. 10.1.1; they may appear overlapped or they may be hidden under other windows on your desktop)[12].

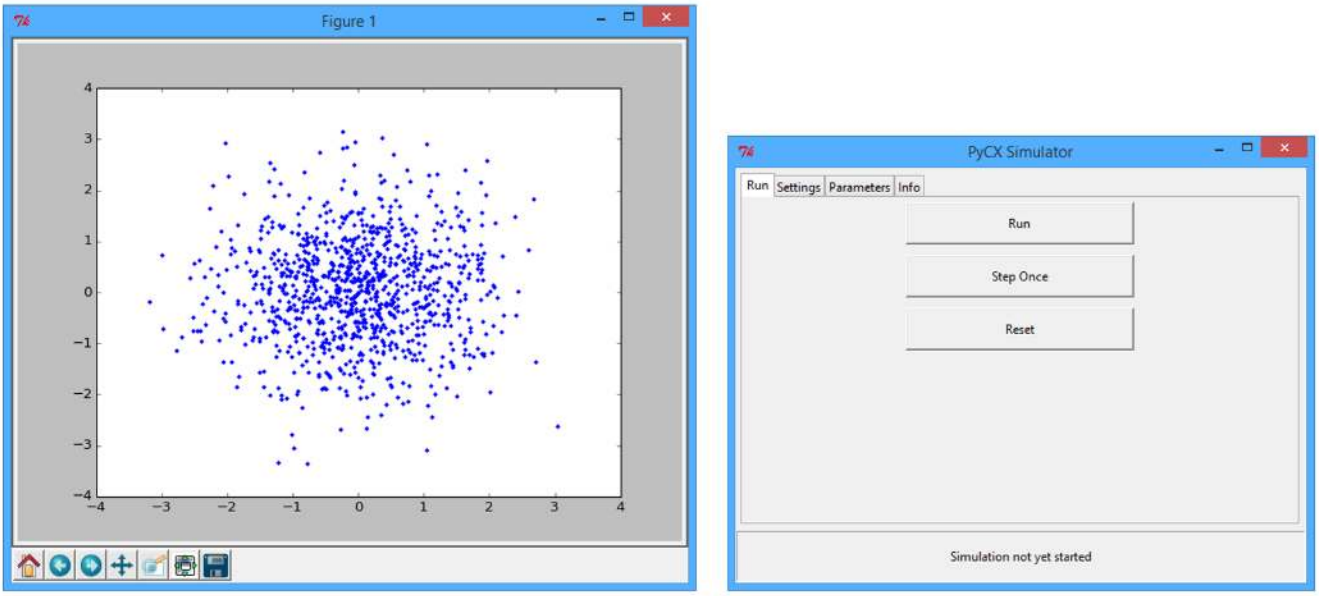


Figure 10.2.1: *Graphical user interface (GUI) generated by pycxsimulator.py. Left: Visualization window. Right: Control window.*

This interface is very minimalistic compared to other software tools, but you can still do basic interactive operations. Under the "Run" tab of the control window (which shows up by default), there are three self-explanatory buttons to run/pause the simulation, update the system just for one step, and reset the system's state. When you run the simulation, the system's state will be updated dynamically and continuously in the other visualization window. To close the simulator, close the control window.

Under the "Settings" tab of the control window, you can change how many times the system is updated before each visualization (default = 1) and the length of waiting time between the visualizations (default = 0 ms). The other two tabs ("Parameters" and "Info") are initially blank. You can include information to be displayed under the "Info" tab as a "docstring" (string placed as a first statement) in the initialize function. For example:

Code 10.7:

```
def initialize():
    '''
This is my first PyCX simulator code.
It simulates random motion of n particles.
Copyright 2014 John Doe
    '''
    global xlist, ylist
    ...
```

This additional docstring appears under the "Info" tab as shown in Fig. 10.1.2.

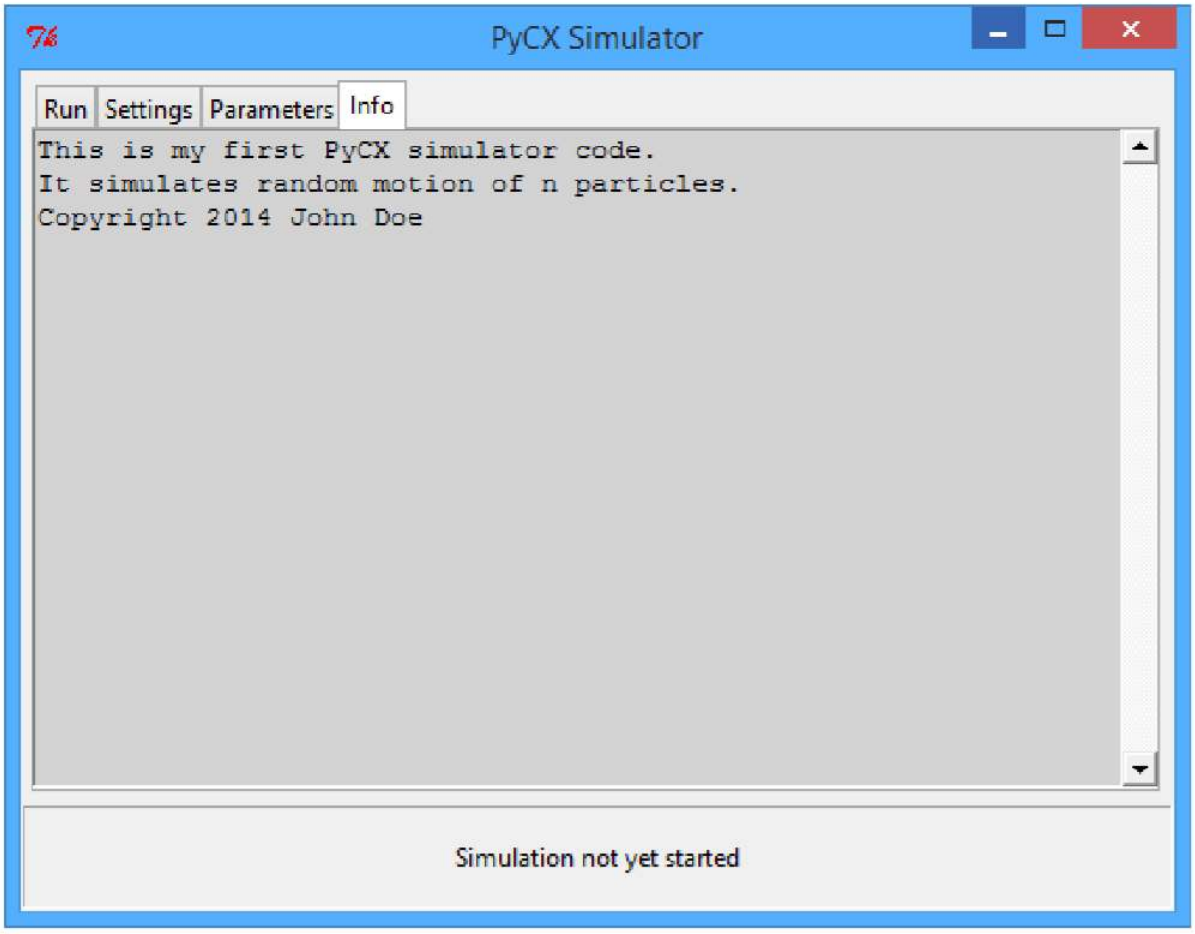


Figure **10.2.2**: *Information about the simulation added as a "docstring" in the initialization function.*

? Exercise 10.2.1

The animation generated by Code 10.6 frequently readjusts the plot range so that the result is rather hard to watch. Also, the aspect ratio of the plot is not 1, which makes it hard to see the actual shape of the particles' distribution. Search matplotlib's online documentation (`http://matplotlib.org/`) to find out how to fix these problems.

[1]If you are using Anaconda Spyder, make sure you run the code in a plain Python console (not an IPython console). You can open a plain Python console from the "Consoles" menu.

[2]If you are using Enthought Canopy and can't run the simulation, try the following:

1. Go to "Edit"→"Preferences"→"Python" tab in Enthought Canopy.
2. Uncheck the "Use PyLab" check box, and click "OK."
3. Choose "Run"→"Restart kernel."
4. Run your code.

If it still doesn't work, re-check the "Use PyLab" check box, and try again.

This page titled 10.2: Interactive Simulation with PyCX is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

10.3: Interactive Parameter Control in PyCX

In Code 10.6, the parameter values are all directly given in the code and are not change able from the control window. PyCX has a feature, however,by which you can create interactive parameter setters (thanks to Przemysław Szufel and Bogumił Kami´nski at the Warsaw School of Economics who developed this nice feature!). A parameter setter should be defined as a function in the following format:

Code 10.8:

```
def <parameter setter name> (val = <parameter name>):
    '''
    <explanation of parameter>
    <this part will be displayed when you mouse-over on parameter setter>
    '''
    global <parameter name>
    <parameter name> = int(val) # or float(val), str(val), etc.
    return val
```

This may look a bit confusing, but all you need to do is to fill in the <...> parts. Note that the int function in the code above may need to be changed according to the type of the parameter (`float` for real-valued parameters, `str` for string-valued ones, etc.). Once you define your own parameter setter functions, you can include them as an option when you call the pycxsimulator.GUI() function as follows:

Code 10.9:

```
pycxsimulator.GUI(parameterSetters=<list of parameter setters>).start...
```

Here is an example of a parameter setter implementation that allows you to interactively change the number of particles:

Code 10.10: random-walk-2D-pSetter.py

```
def num_particles (val = n):
    '''
    Number of particles.
    Make sure you change this parameter while the simulation is not running,
    and reset the simulation before running it. Otherwise it causes an error!
    '''
    global n
    n = int(val)
    return val

import pycxsimulator
pycxsimulator.GUI(parameterSetters = [num_particles]).start(func=[initialize
, observe, update])
```

Once you apply this modification to Code 10.6 and run it, the new parameter setter appears under the “Parameters” tab of the control window (Fig. 10.4.1). You can enter a new value into the input box, and then click either “Save parameters to the running

model" or "Save parameters to the model and reset the model," and the new parameter value is reflected in the model immediately.

Exercise 10.3.1

To the code developed above, add one more parameter setter for sd (standard deviation of Gaussian noise for random motion of particles).

This page titled 10.3: Interactive Parameter Control in PyCX is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

10.4: Simulation without PyCX

Finally, I would like to emphasize an important fact: The PyCX simulator file used in this chapter was used only for creating a GUI, while the core simulation model was still fully implemented in your own code. This means that your simulation model is completely independent of PyCX, and once the interactive exploration and model verification is over, your simulator can "graduate" from PyCX and run on its own.

For example, here is a revised version of Code 10.6, which automatically generates a series of image files *without using PyCX at all*. You can generate an animated movie from the saved image files using, e.g., Windows Movie Maker. This is a nice example that illustrates the main purpose of PyCX—to serve as a stepping stone for students and

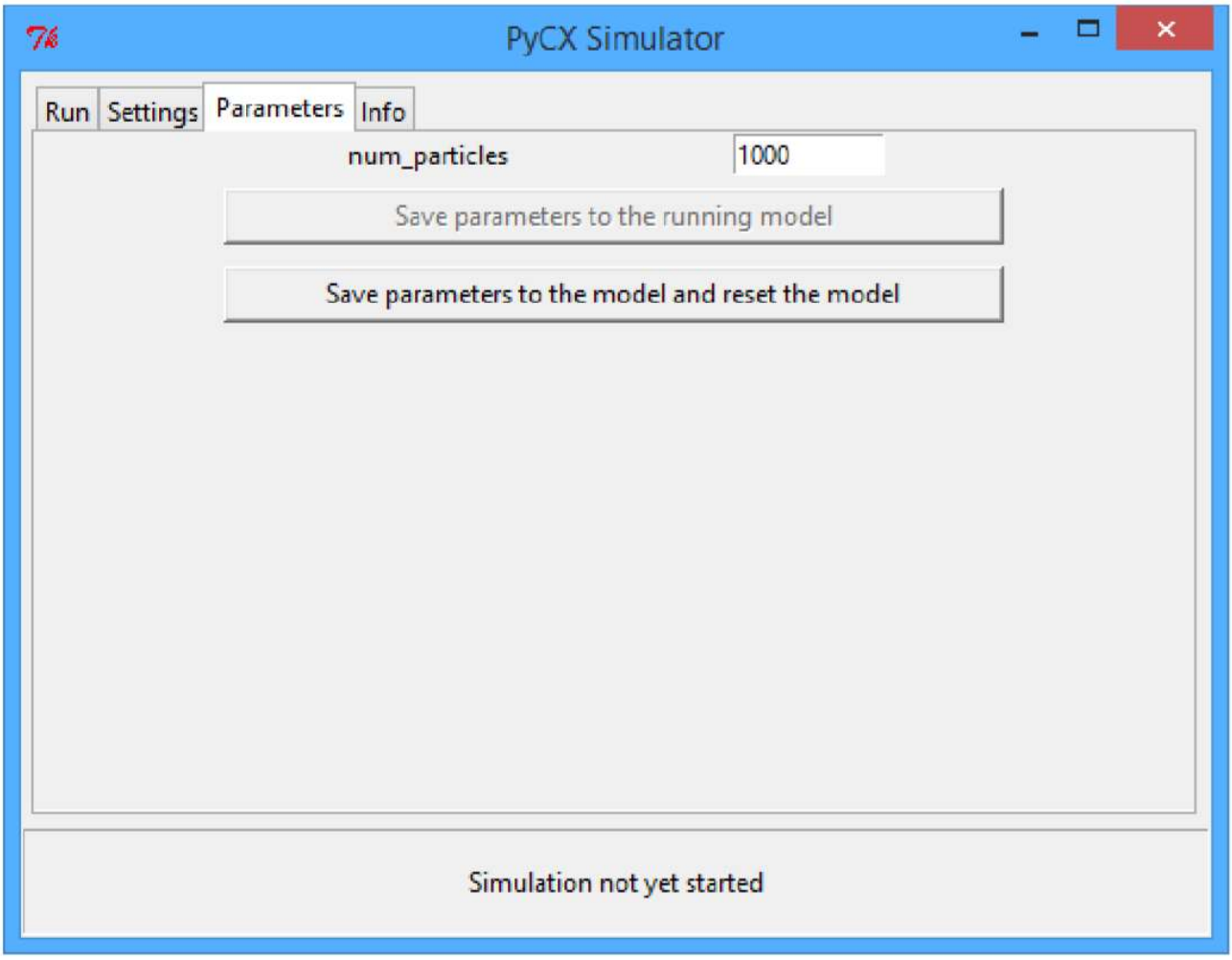


Figure 10.4.1: *New parameter setter for the number of particles implemented in Code 10.10.*

researchers in learning complex systems modeling and simulation, so that it eventually becomes unnecessary once they have acquired sufficient programming skills.

Code 10.11: random-walk-2D-standalone.py

```
from pylab import *

import random as rd
n = 1000 # number of particles
sd = 0.1 # standard deviation of Gaussian noise

def initialize():
    global xlist, ylist
    xlist = []
    ylist = []
    for i in xrange(n):
        xlist.append(rd.gauss(0, 1))
        ylist.append(rd.gauss(0, 1))
```

```
def observe():
    global xlist, ylist
    cla()
    plot(xlist, ylist, '.')
    savefig(str(t) + '.png')

def update():
    global xlist, ylist
    for i in xrange(n):
        xlist[i] += rd.gauss(0, sd)
        ylist[i] += rd.gauss(0, sd)

t = 0
initialize()
```

```
observe()
for t in xrange(1, 100):
    update()
    observe()
```

? Exercise 10.4.1

Develop your own interactive simulation code. Explore various features of the plotting functions and the PyCX simulator's GUI. Then revise your code so that it automatically generates a series of image files without using PyCX. Finally, create an animated movie file using the image files generated by your own code. Enjoy!

This page titled 10.4: Simulation without PyCX is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

11: Cellular Automata I - Modeling

11.1: Definition of Cellular Automata

11.2: Examples of Simple Binary Cellular Automata Rules

11.3: Simulating Cellular Automata

11.4: Extensions of Cellular Automata

11.5: Examples of Biological Cellular Automata Models

This page titled 11: Cellular Automata I - Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

11.1: Definition of Cellular Automata

"*Automaton*" (plural: "automata") is a technical term used in computer science and mathematics for a theoretical machine that changes its internal state based on inputs and its previous state. The state set is usually defined as finite and discrete, which often causes nonlinearity in the system's dynamics.

Cellular automata (CA) [18] are a set of such automata arranged along a regular spatial grid, whose states are simultaneously updated by a uniformly applied *state-transition function* that refers to the states of their neighbors. Such simultaneous updating is also called *synchronous updating* (which could be loosened to be asynchronous; to be discussed later). The original idea of CA was invented in the 1940s and 1950s by John von Neumann and his collaborator Stanisław Ulam. They invented this modeling framework, which was among the very first to model complex systems, in order to describe self-reproductive and evolvable behavior of living systems [11].

Because CA are very powerful in describing highly nonlinear spatio-temporal dynamics in a simple, concise manner, they have been extensively utilized for modeling various phenomena, such as molecular dynamics, hydrodynamics, physical properties of materials, reaction-diffusion chemical processes, growth and morphogenesis of a living organism, ecological interaction and evolution of populations, propagation of traffic jams, social and economical dynamics, and so forth. They have also been utilized for computational applications such as image processing, random number generation, and cryptography.

There are some technical definitions and terminologies you need to know to discuss CA models, so here comes a barrage of definitions.

Mathematically speaking, CA are defined as a spatially distributed dynamical system where both time and space are discrete. A CA model consists of identical automata (cells or sites) uniformly arranged on lattice points in a D-dimensional discrete space (usually $D = 1, 2,$ or 3). Each automaton is a dynamical variable, and its temporal change is given by

$$s_{t+1}(x) = F(s_t(x + x_0), s_t(x + x_1), \ldots, s_t(x + x_{n-1})), \tag{11.1.1}$$

where $s_t(x)$ is the state of an automaton located at x at time t, F is the *state-transition function*, and $N = x_0, x_1, \ldots, x_{n-1}$ is the neighborhood. The idea that the same statetransition function and the same neighborhood apply uniformly to all spatial locations is the most characteristic assumption of CA. When von Neumann and Ulam developed this modeling framework, researchers generally didn't have explicit empirical data about how things were connected in real-world complex systems. Therefore, it was a reasonable first step to assume spatial regularity and homogeneity (which will be extended later in network models).

s_t is a function that maps spatial locations to states, which is called a *configuration* of the CA at time t. A configuration intuitively means the spatial pattern that the CA display at that time. These definitions are illustrated in Fig. 11.1.1.

Neighborhood N is usually set up so that it is centered around the focal cell being updated ($x_0 = 0$) and spatially localized ($|x_i - x_0| \leq r$ for $i = 1, 2, \ldots, n - 1$), where r is called a radius of N. In other words, a cell's next state is determined locally according to its own current state and its local neighbors' current states. A specific arrangement of states within the neighborhood is called a *situation* here. Figure 11.1.2 shows typical examples of neighborhoods often used for two-dimensional CA. In CA with von Neumann neighborhoods (Fig. 11.1.2, left), each cell changes its state according to the states of its upper, lower, right, and left neighbor cells as well as itself. With Moore neighborhoods (Fig. 11.1.2, right), four diagonal cells are added to the neighborhood.

The state-transition function is applied uniformly and simultaneously to all cells in the space. The function can be described in the form of a look-up table (as shown in Fig. 11.1.1), some mathematical formula, or a more high-level algorithmic language.

If a state-transition function always gives an identical state to all the situations that are identical to each other when rotated, then the CA model has *rotational symmetry*. Such symmetry is often employed in CA with an aim to model physical phenomena. Rotational symmetry is called *strong* if all the states of the CA are orientation-free and if the rotation of a situation doesn't involve any rotation of the states themselves (Fig. 11.1.3, left). Otherwise, it is called *weak* (Fig. 11.1.3, right). In CA with weak rotational symmetry, some states are oriented, and the rotation of a situation requires rotational adjustments of those states as well. Von Neumann's original CA model adopted weak rotational symmetry.

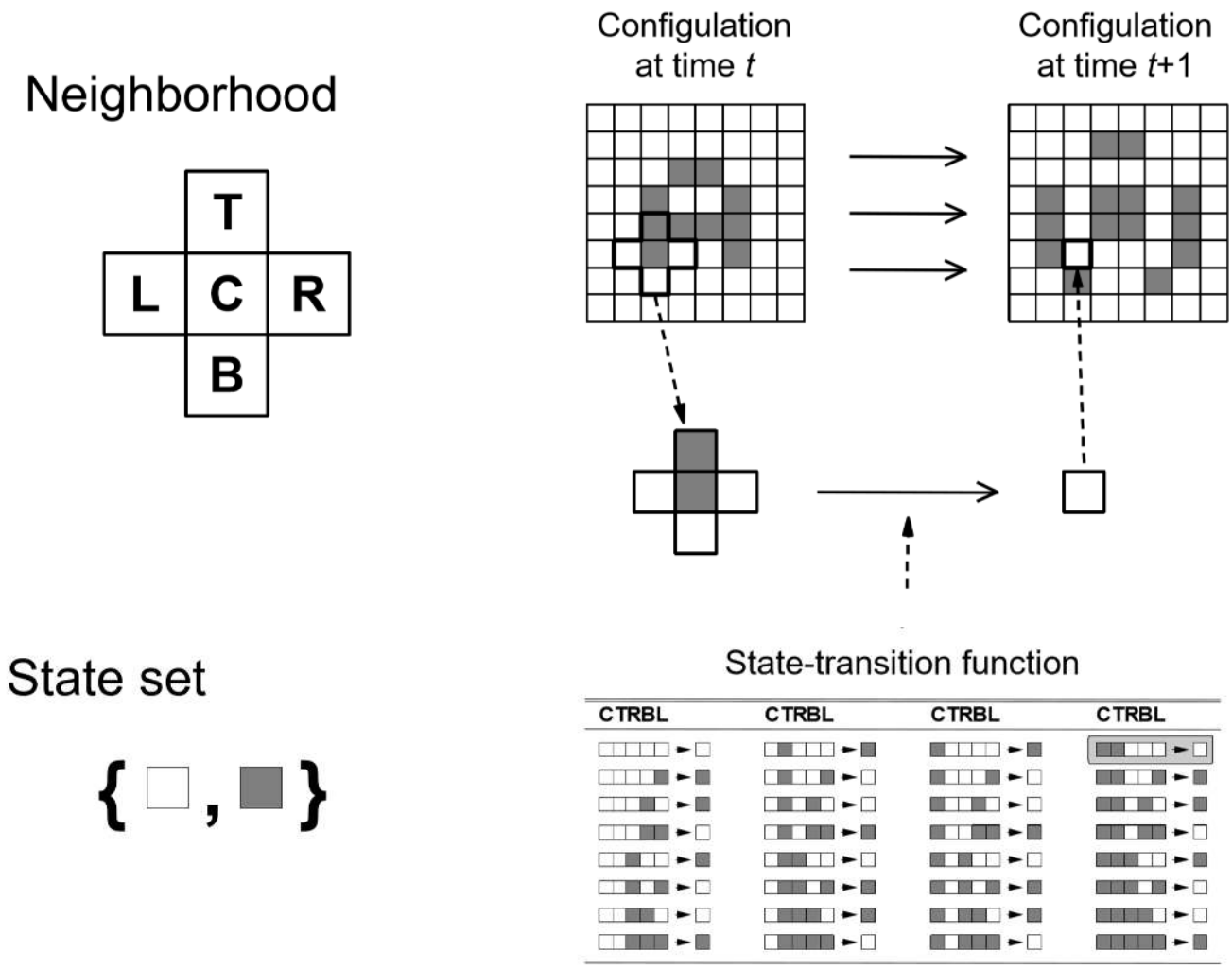


Figure 11.1.1: Schematic illustration of how cellular automata work.

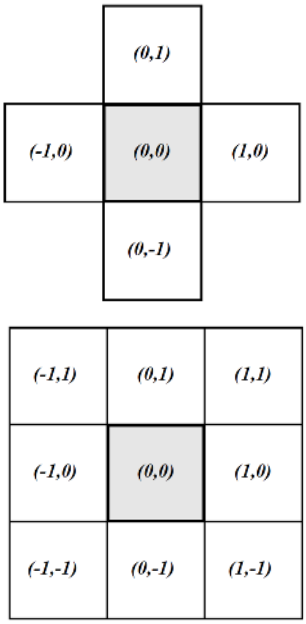


Figure 11.1.2: : Examples of neighborhoods often used for two-dimensional CA. Left: von Neumann neighborhood. Right: Moore neighborhood.

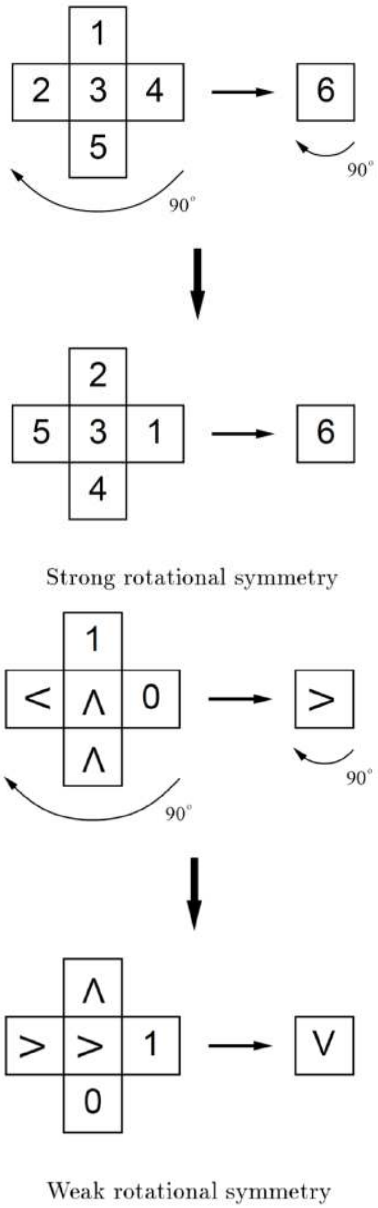


Figure **11.1.3**:Schematic illustration of rotational symmetry in two-dimensional CA with von Neumann neighborhoods.

If a state-transition function depends only on the sum of the states of cells within the neighborhood, the CA is called totalistic. By definition, such state-transition functions are rotationally symmetric. The totalistic assumption makes the design of CA models much simpler, yet they can still produce a wide variety of complex dynamics [34].

The states of CA are usually categorized as either *quiescent* or *non-quiescent*. A cell in a quiescent state remains in the same state if all of its neighbors are also in the same quiescent state. Many CA models have at least one such quiescent state, often represented by either “0” or “ ” (blank). This state symbolizes a “vacuum” in the CA universe. Non-quiescent states are also called active states, because they can change dynamically and interact with nearby states. Such active states usually play a primary role in producing complex behaviors within CA.

Because CA models have a spatial extension as well as a temporal extension, you need to specify spatial boundary conditions in addition to initial conditions in order to study their behaviors. There are several commonly adopted boundary conditions:

No boundaries

This assumes that the space is infinite and completely filled with the quiescent state

Periodic boundaries

This assumes that the space is “wrapped around” each spatial axis, i.e., the cells at one edge of a finite space are connected to the cells at the opposite edge. Examples include a ring for 1-D CA and a torus for 2-D CA

Cut-off boundaries

This assumes that cells at the edge of a finite space don’t have neighbors beyond the boundaries. This necessarily results in fewer numbers of neighbors, and therefore, the state-transition function must be designed so that it can handle such cases.

Fixed boundaries

This assumes that cells at the edge of a finite space have fixed states that will never change. This is often the easiest choice when you implement a simulation code of a CA model

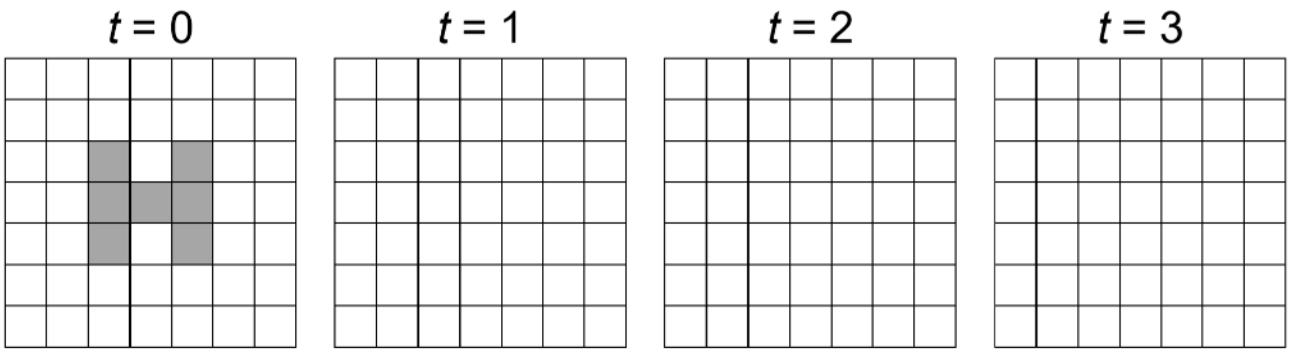
? Exercise 11.1.1

Shown below is an example of a one-dimensional totalistic CA model with radius 1 and a periodic boundary condition. White means 0, while gray means 1. Each cell switches to *round(S/3)* in every time step, where S is the local sum of states within its neighborhood. Complete the time evolution of this CA.



? Exercise 11.1.2

Shown below is an example of a two-dimensional totalistic CA model with von Neumann neighborhoods and with no boundary (infinite space) conditions. White means 0 (= quiescent state), while gray means 1. Each cell switches to *round(S/5)* in every time step, where S is the local sum of the states within its neighborhood. Complete the time evolution of this CA.



This page titled 11.1: Definition of Cellular Automata is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

11.2: Examples of Simple Binary Cellular Automata Rules

Majority rule The two exercises in the previous section were actually examples of CA with a state-transition function called the *majority rule* (a.k.a. *voting rule*). In this rule, each cell switches its state to a local majority choice within its neighborhood. This rule is so simple that it can be easily generalized to various settings, such as multi-dimensional space, multiple states, larger neighborhood size, etc. Note that all states are quiescent states in this CA. It is known that this CA model self-organizes into geographically separated patches made of distinct states, depending on initial conditions. Figure 11.4 shows an example of a majority rule-based 2-D CA with binary states (black and white), each of which is present at equal probability in the initial condition.

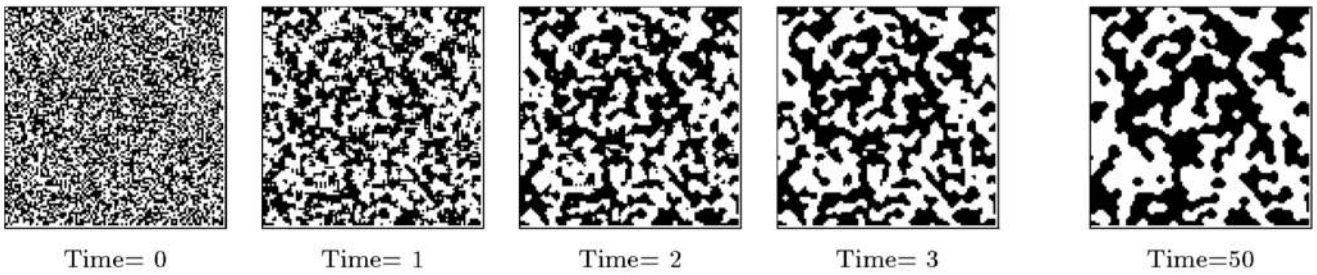


Figure **11.2.1**: *Typical behavior of binary CA with a Moore neighborhood governed by the majority rule.*

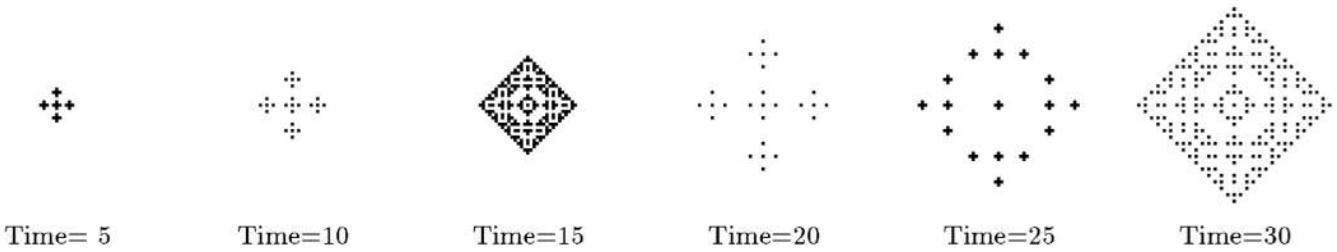
Parity rule The *parity rule*, also called the *XOR (exclusive OR) rule*, is another well known state-transition function for CA. Its state-transition function is defined mathematically as

$$s_{t+1}(x) = \sum_{i=0}^{n-1} s_t(x + x_i) \qquad (mod\ k), \tag{11.2.1}$$

where k is the number of states ($k = 2$ for binary CA). It is known that, in this CA, any arbitrary pattern embedded in the initial configuration replicates itself and propagates over the space indefinitely. Figure 11.2.2 shows examples of such behaviors. In fact, this self-replicative feature is universal for all CA with parity rules, regardless of the numbers of states (k) or the neighborhood radius (r). This is because, under this rule, the growth pattern created from a single active cell (Fig. 11.2.2, top row) doesn't interact with other growth patterns that are created from other active cells (i.e., they are "independent" from each other). Any future pattern from any initial configuration can be predicted by a simple superposition of such growth patterns.

Game of Life The final example is the most popular 2-D binary CA, named the "*Game of Life*," created by mathematician John Conway. Since its popularization by Martin Gardner in *Scientific American* in the early 1970s [35, 36], the Game of Life has attracted a lot of interest from researchers and mathematics/computer hobbyists all over the world, because of its fascinating, highly nontrivial behaviors. Its state-transition function uses the Moore neighborhood and is inspired by the birth and death of living organisms and their dependency on population density, as follows:

- A dead (quiescent) cell will turn into a living (active) cell if and only if it is surrounded by exactly three living cells.



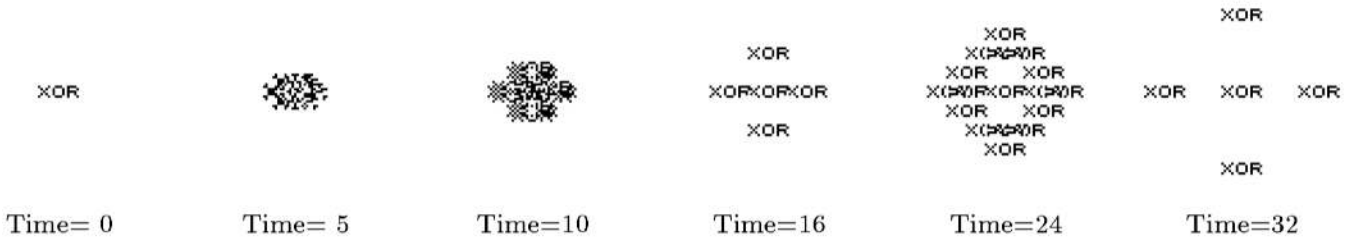


Figure **11.2.2**: *Typical behaviors of binary CA with a von Neumann neighborhood governed by the parity (XOR) rule.*

- A living cell will remain alive if and only if it is surrounded by two or three other living cells. Otherwise it will die.

The Game of Life shows quite dynamic, almost life-like behaviors (Fig. 11.3.1). Many intriguing characteristics have been discovered about this game, including its statistical properties, computational universality, the possibility of the emergence of self-replicative creatures within it, and so on. It is often considered one of the historical roots of *Artificial Life*[1], an interdisciplinary research area that aims to synthesize living systems using non-living materials. The artificial life community emerged in the 1980s and grew together with the complex systems community, and thus these two communities are closely related to each other. Cellular automata have been a popular modeling framework used by artificial life researchers to model self-replicative and evolutionary dynamics of artificial organisms [37, 38, 39, 40].

[1] http://alife.org/

This page titled 11.2: Examples of Simple Binary Cellular Automata Rules is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

11.3: Simulating Cellular Automata

Despite their capability to represent various complex nonlinear phenomena, CA are relatively easy to implement and simulate because of their discreteness and homogeneity.

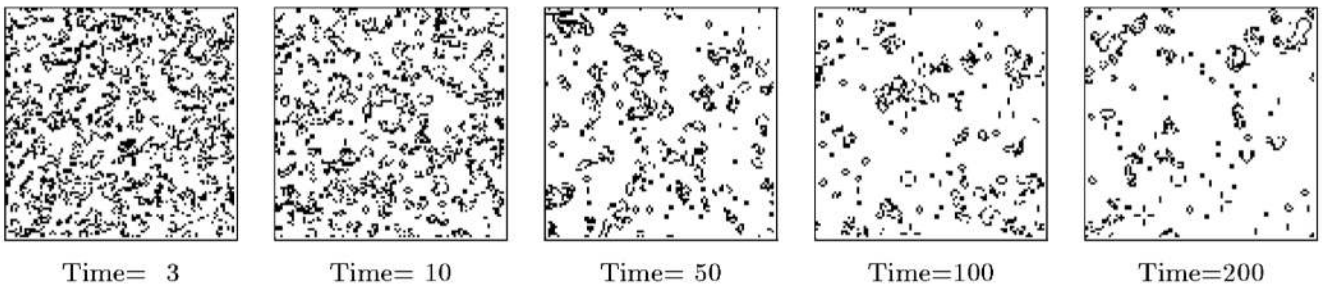


Figure **11.3.1**:: *Typical behavior of the most well-known binary CA, the Game of Life.*

There are existing software tools[2] and online interactive demonstrations[3] already available for cellular automata simulation, but it is nonetheless helpful to learn how to develop a CA simulator by yourself. Let's do so in Python, by working through the following example step by step. The CA model we plan to implement here is a binary CA model with the *droplet rule* [4]. Its state-transition function can be understood as a model of panic propagation among individuals sitting in a gym after a fire alarm goes off. Here is the rule (which uses the Moore neighborhoods):

- A normal individual will get panicky if he or she is surrounded by four or more panicky individuals.
- A panicky individual will remain panicky if he or she is surrounded by three or more panicky individuals. Otherwise he or she will revert back to normal.

Note that this rule can be further simplified to the following single rule:

- If there are four or more panicky individuals within the neighborhood, the central cell will become panicky; otherwise it will become normal.

Here are other model assumptions:

- Space: 2-D, $n \times n$ ($n = 100$ for the time being)
- Boundary condition: periodic
- Initial condition: Random (panicky individuals with probability p; normal ones with probability $1 - p$)

We will use pycxsimulator.py for dynamic CA simulations. Just as before, we need to design and implement the three essential components—initialization, observation, and updating.

To implement the initialization part, we have to decide how we are going to represent the states of the system. Since the configuration of this CA model is a regular two-dimensional grid, it is natural to use Python's `array` data structure. We can initialize it with randomly assigned states with probability p. Moreover, we should actually prepare two such arrays, one for the current time step and the other for the next time step, in order to avoid any unwanted conflict during the state updating process. So here is an example of how to implement the initialization part:

Code 11.1:

```
n = 100 # size of space: n x n
p = 0.1 # probability of initially panicky individuals

def initialize():
    global config, nextconfig
    config = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
            config[x, y] = 1 if random() < p else 0
    nextconfig = zeros([n, n])
```

Here, `zeros([a, b])` is a function that generates an all-zero array with `a` rows and `b` columns. While `config` is initialized with randomly assigned states (1 for panicky individuals, 0 for normal), `nextconfig` is not, because it will be populated with the next states at the time of state updating.

The next thing is the observation. Fortunately, pylab has a built-in function `imshow` that is perfect for our purpose to visualize the content of an array:

Code 11.2:

```
def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)
```

Note that the `cmap` option in `imshow` is to specify the color scheme used in the plot. Without it, pylab uses dark blue and red for binary values by default, which are rather hard to see.

The updating part requires some algorithmic thinking. The state-transition function needs to count the number of panicky individuals within a neighborhood. How can we do this? The positions of the neighbor cells within the Moore neighborhood around a position (x, y) can be written as

$$(x', y') | x - 1 \leq x' \leq x + 1, y - 1 \leq y' \leq y + 1. \tag{11.3.1}$$

This suggests that the neighbor cells can be swept through using nested `for` loops for relative coordinate variables, say, dx and dy, each ranging from `-1` to `+1`. Counting panicky individuals (1's) using this idea can be implemented in Python as follows:

Code 11.3:

```
        count = 0
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                count += config[(x + dx) % n, (y + dy) % n]
```

Here, `dx` and `dy` are relative coordinates around (x, y), each varying from `-1` to `+1`. They are added to `x` and `y`, respectively, to look up the current state of the cell located at $(x + dx, y + dy)$ in `config`. The expression (...) % n means that the

value inside the parentheses is contained inside the `[0,n - 1]` range by the mod operator (%). This is a useful coding technique to implement periodic boundary conditions in a very simple manner.

The counting code given above needs to be applied to all the cells in the space, so it should be included in another set of nested loops for `x` and `y` to sweep over the entire space. For each spatial location, the counting will be executed, and the next state of the cell will be determined based on the result. Here is the completed code for the updating:

Code 11.4:

```
def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            count = 0
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    count += config[(x + dx) % n, (y + dy) % n]
```

```
            nextconfig[x, y] = 1 if count >= 4 else 0
    config, nextconfig = nextconfig, config
```

Note the swapping of `config` and `nextconfig` at the end. This is precisely the same technique that we did for the simulation of multi-variable dynamical systems in the earlier chapters.

By putting all the above codes together, the completed simulator code in its entirety looks like this:

Code 11.5: panic-ca.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of space: n x n
p = 0.1 # probability of initially panicky individuals

def initialize():
    global config, nextconfig
    config = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
```

```
            config[x, y] = 1 if random() < p else 0
    nextconfig = zeros([n, n])

def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)

def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            count = 0
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    count += config[(x + dx) % n, (y + dy) % n]
            nextconfig[x, y] = 1 if count >= 4 else 0
    config, nextconfig = nextconfig, config

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

When you run this code, you see the results like those shown in Fig. 11.3.2. As you can see, with the initial configuration with $p = 0.1$, most of the panicky states disappear quickly, leaving only a few self-sustaining clusters of panicky people. They may look like condensed water droplets, hence the name of the model (droplet rule).

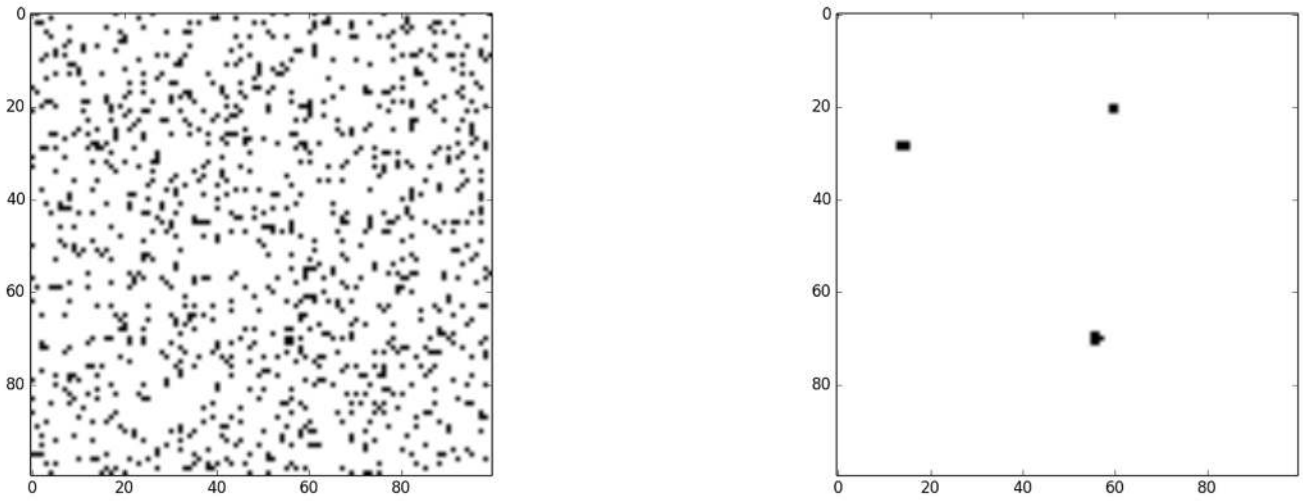


Figure **11.3.2**: *Visual output of Code 11.5. Left: Initial configuration with* $p = 0.1$. *Right: Final configuration after 20 time steps.*

? Exercise 11.3.1:

Modify Code 11.5 to implement a simulator of the Game of Life CA. Simulate the dynamics from a random initial configuration. Measure the density of state 1's in the configuration at each time step, and plot how the density changes over time. This can be done by creating an empty list in the `initialize` function, and then making the measurement and appending the result to the list in the `observe` function. The results stored in the list can be plotted manually after the simulation, or they could be plotted next to the visualization using pylab's subplot function during the simulation.

? Exercise 11.3.2:

Modify Code 11.5 to implement a simulator of the majority rule CA in a two-dimensional space. Then answer the following questions by conducting simulations:

- What happens if you change the ratio of binary states in the initial condition?
- What happens if you increase the radius of the neighborhoods?
- What happens if you increase the number of states?

The second model revision in the previous exercise (increasing the radius of neighborhoods) for the majority rule CA produces quite interesting spatial dynamics. Specifically, the boundaries between the two states tend to straighten out, and the characteristic scale of spatial features continuously becomes larger and larger over time (Fig. 11.3.3). This behavior is called *coarsening* (to be more specific, non-conserved coarsening). It can be seen in many real-world contexts, such as geographical distributions of distinct cultural/political/linguistic states in human populations, or incompatible genetic types in animal/plant populations.



Figure **11.3.3**: *Coarsening behavior observed in a binary CA with the majority rule in a* `100 × 100` *2-D spatial grid with periodic boundary conditions. The radius of the neighborhoods was set to 5. Time flows from left to right.*

What is most interesting about this coarsening behavior is that, once clear boundaries are established between domains of different states, the system's macroscopic behavior can be described using emergent properties of those boundaries, such as their surface tensions and direction of movement[41,42]. The final fate of the system is determined not by the relative frequencies of the two states in the space, but by the topological features of the boundaries. For example, if a big "island" of white states is surrounded by a thin "ring" of black states, the latter minority will eventually dominate, no matter how small its initial proportion is (even though this is still driven by the majority rule!). This is an illustrative example that shows how important it is to consider emergent macroscopic properties of complex systems, and how counter-intuitive their behaviors can be sometimes.

Here is one more interesting fact about CA dynamics. The droplet/panic model discussed above has an interesting property: When you increase the initial density of panicky people (e.g., $p = 0.3$), the result changes dramatically. As seen in Fig. 11.3.4, the initially formed clusters tend to attach to each other, which makes their growth unstoppable. The whole space will eventually be filled up with all panicky people, which could be a disaster if this was a real situation.

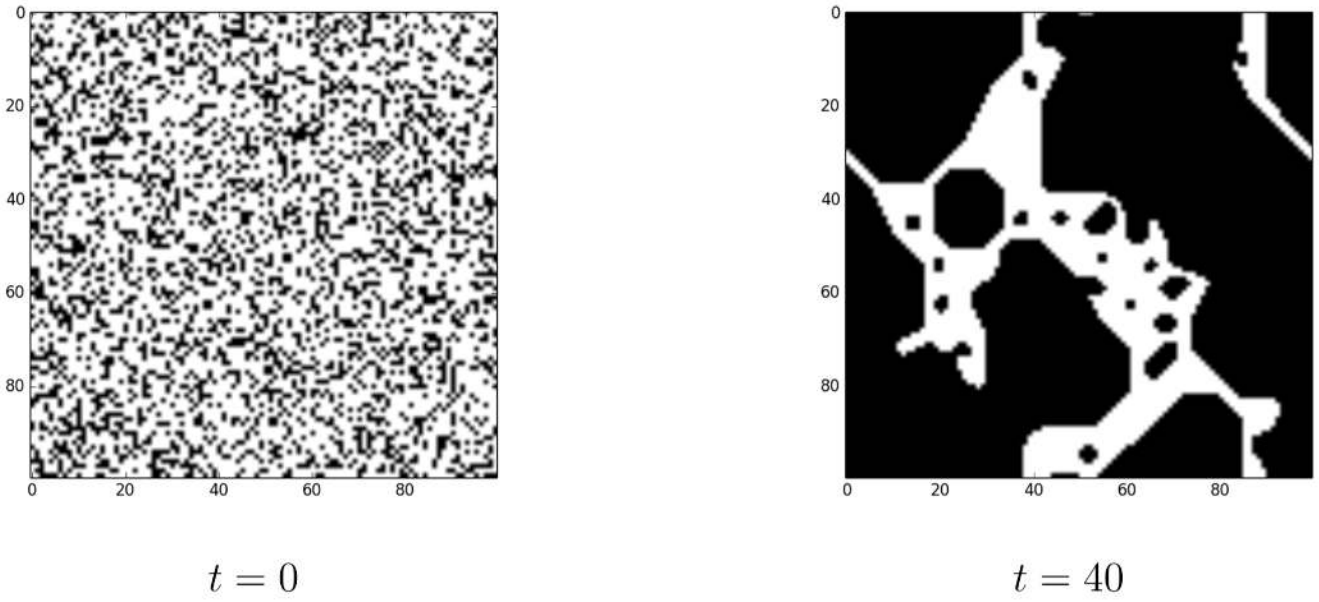


Figure **11.3.4**: *Another visual output of Code 11.5. Left: Initial configuration with* $p = 0.3$. *Right: Configuration after 40 time steps.*

You can explore the value of p to find out that the transition between these two distinct behaviors takes place at a rather narrow range of p. This is an example of a *phase transition*, which is defined informally as follows:

> A *phase transition* is a transition of macroscopic properties of a collective system that occurs when its environmental or internal conditions are varied

A familiar example of phase transitions is the transition between different phases of matter, i.e., solid, liquid, and gas, which occur when temperature and/or pressure are varied. Phase transitions can be characterized by measuring what physicists call order

parameters[4] that represent how ordered a macroscopic state of the system is. A phase transition can be understood as a bifurcation observed in the macroscopic properties (i.e., order parameters) of a collective system.

? Exercise 11.3.3:

Implement an interactive parameter setter for p in Code 11.5. Then conduct systematic simulations with varying p, and identify its critical value below which isolated clusters are formed but above which the whole space is filled with panic.

[2]Most notable is Golly (http://golly.sourceforge.net/).

[3]For example, check out Wolfram Demonstrations Project(demonstrations.wolfram.com/)and Shodor.org's interactive activities (http://www.shodor.org/interactivate/activities/).

[4]Note that the word"parameter" in this context means an outcome of a measurement, and not a condition or input as in "model parameters."

This page titled 11.3: Simulating Cellular Automata is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

11.4: Extensions of Cellular Automata

So far, we discussed CA models in their most conventional settings. But there are several ways to "break" the modeling conventions, which could make CA more useful and applicable to real-world phenomena. Here are some examples.

- **Stochastic cellular automata:** A state-transition function of CA doesn't have to be a rigorous mathematical function. It can be a computational process that produces the output probabilistically. CA with such probabilistic state-transition rules are called *stochastic CA*, which play an important role in mathematical modeling of various biological, social, and physical phenomena. A good example is a CA model of epidemiological processes where infection of a disease takes place stochastically (this will be discussed more in the following section).
- **Multi-layer cellular automata:** States of cells don't have to be scalar. Instead, each spatial location can be associated with several variables (i.e., vectors). Such vector-valued configurations can be considered a superposition of multiple layers, each having a conventional scalar-valued CA model. Multi-layer CA models are useful when multiple biological or chemical species are interacting with each other in a space-time. This is particularly related to reaction-diffusion systems that will be discussed in later chapters.
- **Asynchronous cellular automata:** Synchronous updating is a signature of CA models, but we can even break this convention to make the dynamics asynchronous. There are several asynchronous updating mechanisms possible, such as random updating (a randomly selected cell is updated at each time step), sequential updating (cells are updated in a predetermined sequential order), state-triggered updating (certain states trigger updating of nearby cells), etc. It is often argued that synchronous updating in conventional CA models is too artificial and fragile against slight perturbations in updating orders, and in this sense,the behaviors of a synchronous CA models are deemed more robust and applicable to real-world problems. Moreover, there is a procedure to create asynchronous CA that can robustly emulate the behavior of any synchronous CA [43].

This page titled 11.4: Extensions of Cellular Automata is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

11.5: Examples of Biological Cellular Automata Models

In this final section, I provide more examples of cellular automata models, with a particular emphasis on biological systems. Nearly all biological phenomena involve some kind of spatial extension, such as excitation patterns on neural or muscular tissue, cellular arrangements in an individual organism's body, and population distribution at ecological levels. If a system has a spatial extension, nonlinear local interactions among its components may cause *spontaneous pattern formation*, i.e., self-organization of static or dynamic spatial patterns from initially uniform conditions. Such self-organizing dynamics are quite counter-intuitive, yet they play essential roles in the structure and function of biological systems.

In each of the following examples, I provide basic ideas of the state-transition rules and what kind of patterns can arise if some conditions are met. I assume Moore neighborhoods in these examples (unless noted otherwise), but the shape of the neighborhoods is not so critically important. Completed Python simulator codes are available from http://sourceforge.net/projects/pycx/files/, but you should try implementing your own simulator codes first.

Turing patterns

Animal skin patterns are a beautiful example of pattern formation in biological systems. To provide a theoretical basis of this intriguing phenomenon, British mathematician Alan Turing (who is best known for his fundamental work in theoretical computer science and for his code-breaking work during World War II) developed a family of models of spatio-temporal dynamics of chemical reaction and diffusion processes [44]. His original model was first written in a set of coupled ordinary differential equations on compartmentalized cellular structures, and then it was extended to *partial differential equations* (PDEs) in a continuous space. Later, a much simpler CA version of the same model was proposed by David Young [45]. We will discuss Young's simpler model here.

Assume a two-dimensional space made of cells where each cell can take either a passive (0) or active (1) state. A cell becomes activated if there are a sufficient number of active cells within its local neighborhood. However, other active cells outside this neighborhood try to suppress the activation of the focal cell with relatively weaker influences than those from the active cells in its close vicinity. These dynamics are called *short-range activation* and *long-range inhibition*. This model can be described mathematically as follows:

$$N_a = x' \mid |x'| \leq R_a \tag{11.5.1}$$

$$N_i = x' \mid |x'| \leq R_i \tag{11.5.2}$$

$$a_t(x) = w_a \sum_{x' \epsilon N_a} s_t(x + x') - w_i \sum_{x' \epsilon N_i} s_t(x + x') \tag{11.5.3}$$

$$x_{t+1}(x) = \begin{cases} 1 & \text{if } a_t(x) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{11.5.4}$$

Here, R_a and R_i are the radii of neighborhoods for activation (N_a) and inhibition (N_i), respectively ($R_a < R_i$), and w_a and w_i are the weights that represent their relative strengths. $a_t(x)$ is the result of two neighborhood countings, which tells you whether the short-range activation wins ($a_t(x) > 0$) or the long-range inhibition wins ($a_t(x) \leq 0$)at location x. Figure 11.10 shows the schematic illustration of this state-transition function, as well as a sample simulation result you can get if you implement this model successfully.

? Exercise 11.5.1:

Implement a CA model of the Turing pattern formation in Python. Then try the following:

- What happens if R_a and R_i are varied?
- What happens if w_a and w_i are varied?

Waves in excitable media

Neural and muscle tissues made of animal cells can generate and propagate electrophysiological signals. These cells can get excited in response to external stimuli coming from nearby cells, and they can generate *action potential* across their cell membranes that

will be transmitted as a stimulus to other nearby cells. Once excited, the cell goes through a *refractory period* during which it doesn't respond to any further stimuli. This causes the directionality of signal propagation and the formation of "traveling waves" on tissues.

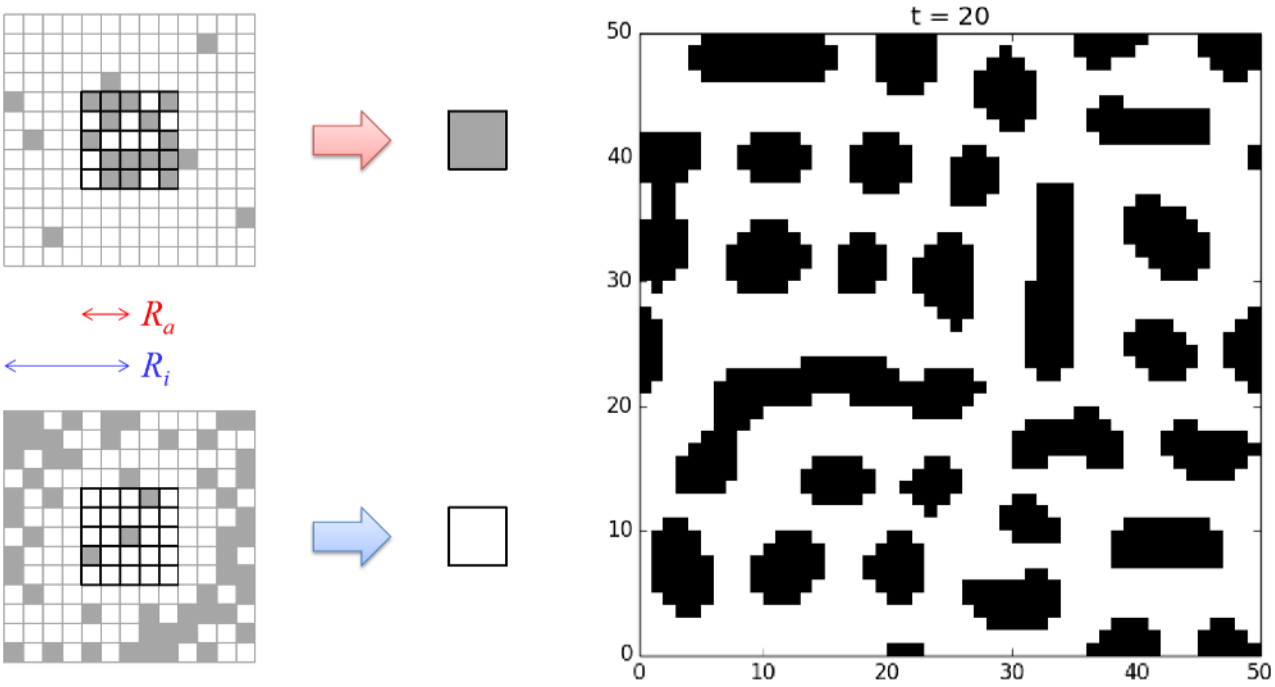


Figure 11.5.1: *Visual output of Code 11.5. Left: Initial configuration with* $p = 0.1$. *Right: Final configuration after 20 time steps.*

This kind of spatial dynamics, driven by propagation of states between adjacent cells that are physically touching each other, are called *contact processes*. This model and the following two are all examples of contact processes.

A stylized CA model of this excitable media can be developed as follows. Assume a two-dimensional space made of cells where each cell takes either a normal (0;quiescent), excited (1), or refractory (2, 3, ...k) state. A normal cell becomes excited stochastically with a probability determined by a function of the number of excited cells in its neighborhood. An excited cell becomes refractory (2) immediately, while a refractory cell remains refractory for a while (but its state keeps counting up) and then it comes back to normal after it reaches k. Figure 11.5.2 shows the schematic illustration of this state-transition function, as well as a sample simulation result you can get if you implement this model successfully. These kinds of uncoordinated traveling waves of excitation (including spirals) are actually experimentally observable on the surface of a human heart under cardiac arrhythmia.

? Exercise 11.5.2:

Implement a CA model of excitable media in Python. Then try the following:

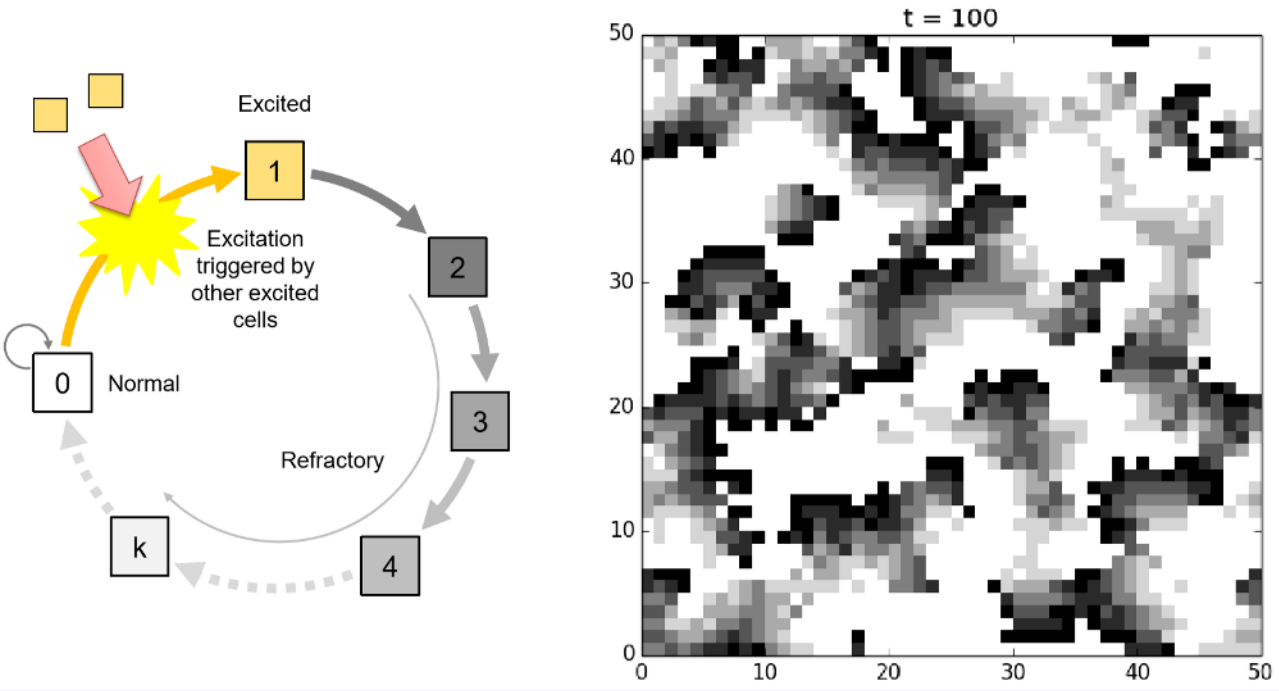


Figure **11.5.2**: *CA model of excitable media. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.*

- What happens if the excitation probability is changed?
- What happens if the length of the refractory period is changed?

Host-pathogen model

A spatially extended host-pathogen model, studied in theoretical biology and epidemiology, is a nice example to demonstrate the subtle relationship between these two antagonistic players. This model can also be viewed as a spatially extended version of the Lotka-Volterra (predator-prey) model, where each cell at a particular location represents either an empty space or an individual organism, instead of population densities that the variables in the Lotka-Volterra model represented.

Assume a two-dimensional space filled with empty sites (0; quiescent) in which a small number of host organisms are initially populated. Some of them are "infected" by pathogens. A healthy host (1; also quiescent) without pathogens will grow into nearby empty sites stochastically. A healthy host may get infected by pathogens with a probability determined by a function of the number of infected hosts (2) in its neighborhood. An infected host will die immediately (or after some period of time). Figure 11.5.3 shows the schematic illustration of this state-transition function, as well as a sample simulation result you could get if you implement this model successfully. Note that this model is similar to the excitable media model discussed above, with the difference that the healthy host state (which corresponds to the excited state in the previous example) is quiescent and thus can remain in the space indefinitely.

The spatial patterns formed in this model are quite different from those in the previously discussed models. Turing patterns and waves in excitable media have clear characteristic length scales (i.e., size of spots or width of waves), but the dynamic patterns formed in the host-pathogen model lacks such characteristic length scales. You will see a number of dynamically forming patches of various sizes, from tiny ones to very large continents.

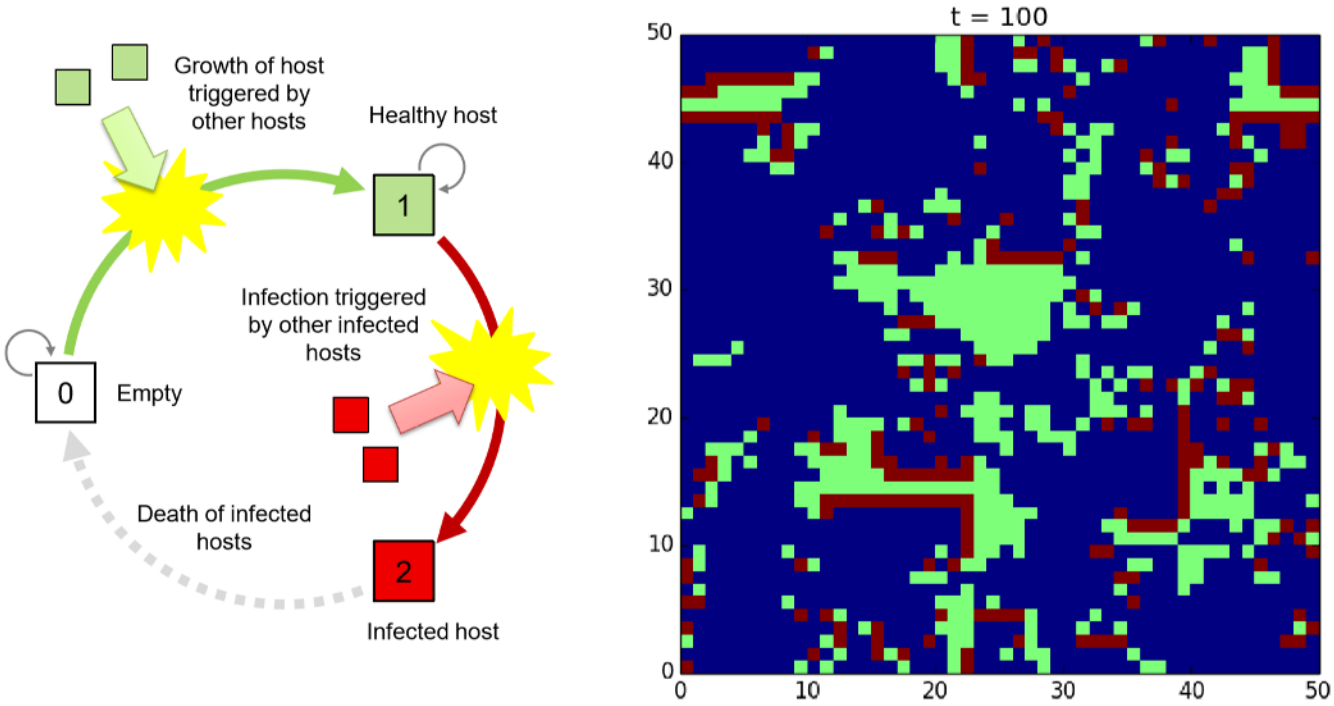


Figure 11.5.3: *CA model of host-pathogen systems. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.*

? Exercise 11.5.3:

Implement a CA model of host-pathogen systems in Python. Then try the following:

- What happens if the infection probability is changed?
- In what conditions will both hosts and pathogens co-exist? In what conditions can hosts exterminate pathogens? In what conditions will both of them become extinct?
- Plot the populations of healthy/infected hosts over time, and see if there are any temporal patterns you can observe.

Epidemic/forest fire model

The final example, the epidemic model, is also about contact processes similar to the previous two examples. One difference is that this model focuses more on static spatial distributions of organisms and their influence on the propagation of an infectious disease within a single epidemic event. This model is also known as the "forest fire" model, so let's use this analogy.

Assume there is a square-shaped geographical area, represented as a CA space, in which trees (1) are distributed with some given probability, p. That is, $p = 0$ means there are no trees in the space, whereas $p = 1$ means trees are everywhere with no open space left in the area. Then, you set fire (2) to one of the trees in this forest to see if the fire you started eventually destroys the entire forest (don't do this in real life!!). A tree will catch fire if there is at least one tree burning in its neighborhood, and the burning tree will be charred (3) completely after one time step.

Figure 11.5.4 shows the schematic illustration of this state-transition function, as well as a sample simulation result you could get if you implement this model successfully. Note that this model doesn't have cyclic local dynamics; possible state transitions are always one way from a tree (1) to a burning tree (2) to being charred (3), which is different from the previous two examples. So the whole system eventually falls into a static final configuration with no further changes possible. But the total area burned in the final configuration greatly depends on the density of trees p. If you start with a sufficiently large value of p, you will see that a significant portion of the forest will be burned down eventually. This phenomenon is called *percolation* in statistical physics, which intuitively means that something found a way to go through a large portion of material from one side to the other.

? Exercise 11.5.4:

Implement a CA model of epidemic/forest fire systems in Python. Then try the following:

- Compare the final results for different values of p.
- Plot the total burned area as a function of p.
- Plot the time until the fire stops spreading as a function of p.

When you work on the above tasks, make sure to carry out multiple independent simulation runs for each value of p and average the results to obtain statistically

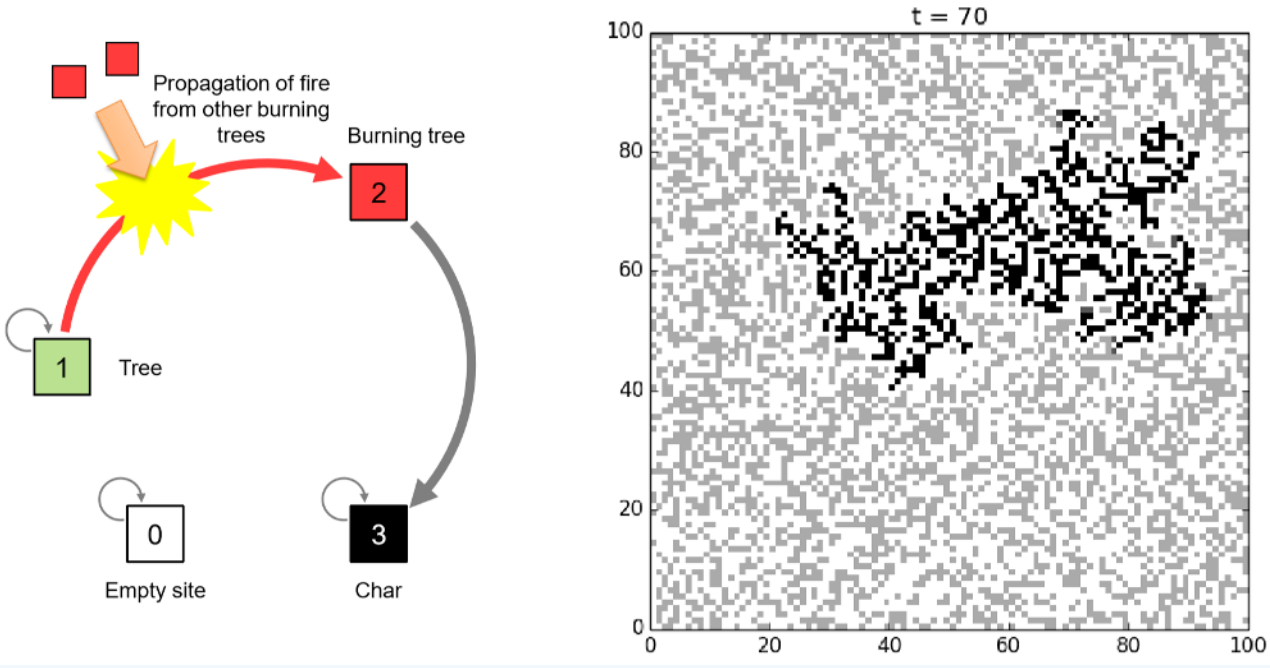


Figure **11.5.3**: *CA model of host-pathogen systems. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.*

more reliable results. Such simulation methods based on random sampling are generally called *Monte Carlo simulations*. In Monte Carlo simulations, you conduct many replications of independent simulation runs for the same experimental settings, and measure the outcome variables from each run to obtain a statistical distribution of the measurements. Using this distribution (e.g., by calculating its average value) will help you enhance the accuracy and reliability of the experimental results.

If you have done the implementation and experiments correctly, you will probably see another case of phase transition in the exercise above. The system's response shows a sharp transition at a critical value of p, above which percolation occurs but below which it doesn't occur. Near this critical value, the system is very sensitive to minor perturbations, and a number of intriguing phenomena (such as the formation of self-similar *fractal* patterns) are found to take place at or near this transition point, which are called *critical behaviors*. Many complex systems, including biological and social ones, are considered to be utilizing such critical behaviors for their self-organizing and information processing purposes. For example, there is a conjecture that animal brains tend to dynamically maintain critical states in their neural dynamics in order to maximize their information processing capabilities. Such *self-organized criticality* in natural systems has been a fundamental research topic in complex systems science.

This page titled 11.5: Examples of Biological Cellular Automata Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

12: Cellular Automata II - Analysis

12.1: Sizes of Rule Space and Phase Space

12.2: Phase Space Visualization

12.3: Mean-Field Approximation

12.4: Renormalization Group Analysis to Predict Percolation Thresholds

---

This page titled 12: Cellular Automata II - Analysis is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

12.1: Sizes of Rule Space and Phase Space

One of the unique features of typical CA models is that time, space, and states of cells are all discrete. Because of such discreteness, the number of all possible state-transition functions is finite, i.e., there are only a finite number of "universes" possible in a given CA setting. Moreover, if the space is finite, all possible configurations of the entire system are also enumerable. This means that, for reasonably small CA settings, one can conduct an exhaustive search of the entire rule space or phase space to study the properties of all the "parallel universes." Stephen Wolfram did such an exhaustive search for a binary CA rule space to illustrate the possible dynamics of CA [34, 46].

Let's calculate how large a rule space/phase space of a given CA setting can be. Here we assume the following:

- Dimension of space: D
- Length of space in each dimension: L
- Radius of neighborhood: R
- Number of states for each cell: K

To calculate the number of possible rules, we first need to know the size of the neighborhood of each cell. For each dimension, the length of each side of a neighborhood is given by $2r+1$, including the cell itself. In a D-dimensional space, this is raised to the power of D, assuming that the neighborhood is a D-dimensional (hyper)cube. So, the size (volume) of the neighborhood is given by

$$n = (2r+1)^D \tag{12.1.1}$$

Each of the n cells in a neighborhood can take one of k states. Therefore, the total number of local situations possible is given by

$$m = k^n = k^{2r+1^D} \tag{12.1.2}$$

Now we can calculate the number of all possible state-transition functions. A function has to map each of the m situations to one of the k states. Therefore, the number of possible mappings is given by

$$R = k^m = k^{k^n} = k^{k^{(2r+1)^D}}. \tag{12.1.3}$$

For example, a one-dimensional ($D=1$) binary ($k=2$) CA model with radius 1 ($r=1$) has $2^{(2\times1+1)^1} = 2^3 = 8$ different possible situations, and thus there are $2^8 = 256$ state-transition functions possible in this CA universe[1]. This seems reasonable, but be careful — this number quickly explodes to astronomical scales for larger k, r, or D.

? Exercise 12.1.1:

Calculate the number of possible state-transition functions for a two-dimensional CA model with two states and Moore neighborhoods (i.e., $r=1$).

? Exercise 12.1.2:

Calculate the number of possible state-transition functions for a three-dimensional CA model with three states and 3-D Moore neighborhoods (i.e., $r=1$).

You must have faced some violently large numbers in these exercises. Various symmetry assumptions (e.g., rotational symmetries, reflectional symmetries, totalistic rules, etc.) are often adopted to reduce the size of the rule spaces of CA models.

How about the size of the phase space? It can actually be much more manageable compared to the size of rule spaces, depending on how big L is. The total number of cells in the space (i.e., the volume of the space) is given by

$$V = L^D. \tag{12.1.4}$$

Each cell in this volume takes one of the k states, so the total number of possible configurations (i.e., the size of the phase space) is simply given by

$$S = k^V = k^{L^D}. \tag{12.1.5}$$

For example,a one-dimensional binary CA model with $L = 10$ has $2^{10^1} = 1024$ possible configurations, a two-dimensional binary CA model with $L = 10$ has ($2^{10^2} \approx 1.27 \times 10^{30}$ possible configurations. The latter is large, but it isn't so huge compared to the size of the rule spaces you saw in the exercises above.

? Exercise 12.1.3:

Calculate the number of all possible configurations of a two-dimensional, three-state CA model with $L = 100$.

[1]There is a well-known rule-numbering scheme for this particular setting proposed by Wolfram, but we don't discuss it in detail in this textbook.

This page titled 12.1: Sizes of Rule Space and Phase Space is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

12.2: Phase Space Visualization

If the phase space of a CA model is not too large, you can visualize it using the technique we discussed in Section 5.4. Such visualizations are helpful for understanding the overall dynamics of the system, especially by measuring the number of separate basins of attraction, their sizes, and the properties of the attractors. For example, if you see only one large basin of attraction, the system doesn't depend on initial conditions and will always fall into the same attractor. Or if you see multiple basins of attraction of about comparable size, the system's behavior is sensitive to the initial conditions. The attractors may be made of a single state or multiple states forming a cycle, which determines whether the system eventually becomes static or remains dynamic (cyclic) indefinitely.

Let's work on an example. Consider a one-dimensional binary CA model with neighborhood radius $r = 2$. We assume the space is made of nine cells with periodic boundary conditions(i.e., the space is a ring made of nine cells). In this setting, the size of its phase space is just $2^9 = 512$, so this is still easy to visualize.

In order to enumerate all possible configurations, it is convenient to define functions that map a specific configuration of the CA to a unique configuration ID number, and vice versa. Here are examples of such functions:

Code 12.1:

```
def config(x):
    return [1 if x & 2**i > 0 else 0 for i in range(L - 1, -1, -1)]

def cf_number(cf):
    return sum(cf[L - 1 - i] * 2**i for i in range(L))
```

Here `L` and `i` are the size of the space and the spatial position of a cell, respectively. These functions use a typical binary notation of an integer as a way to create mapping between a configuration and its unique ID number (from 0 to $2^L - 1$; 511 in our example), arranging the bits in the order of their significance from left to right. For example, the configuration [0, 1, 0, 1, 1, 0, 1, 0, 1] is mapped to $2^7 + 2^5 + 2^4 + 2^2 + 2^0 = 181$. The function `config` receives a non-negative integer x and returns a list made of 0 and 1, i.e., a configuration of the CA that corresponds to the given number. Note that "&" is a logical AND operator, which is used to check if x's `i` -th bit is 1 or not. The function `cf_number` receives a configuration of the CA, `cf` , and returns its unique ID number (i.e., `cf_number` is an inverse function of `config`).

Next, we need to define an updating function to construct a one-step trajectory of the CA model. This is similar to what we usually do in the implementation of CA models:

Code 12.2:

```
def update(cf):
    nextcf = [0] * L
    for x in range(L):
        count = 0
        for dx in range(-r, r + 1):
            count += cf[(x + dx) % L]
        nextcf[x] = 1 if count > (2 * r + 1) * 0.5 else 0
    return nextcf
```

In this example, we adopt the majority rule as the CA's state-transition function, which is written in the second-to-last line. Specifically, each cell turns to 1 if more than half of its local neighbors (there are $2r + 1$ such neighbors including itself) had state 1's, or it otherwise turns to 0. You can revise that line to implement different state-transition rules as well.

Now we have all the necessary pieces for phase space visualization. By plugging the above codes into Code 5.5 and making some edits, we get the following:

Code 12.3: ca-graph-based-phasespace.py

```
from pylab import *
import networkx as nx

g = nx.DiGraph()

r = 2
L = 9
```

```
def config(x):
    return [1 if x & 2**i > 0 else 0 for i in range(L - 1, -1, -1)]

def cf_number(cf):
    return sum(cf[L - 1 - i] * 2**i for i in range(L))

def update(cf):
    nextcf = [0] * L
    for x in range(L):
        count = 0
        for dx in range(-r, r + 1):
            count += cf[(x + dx) % L]
        nextcf[x] = 1 if count > (2 * r + 1) * 0.5 else 0
    return nextcf
```

```
for x in xrange(2**L):
    g.add_edge(x, cf_number(update(config(x))))

ccs = [cc for cc in nx.connected_components(g.to_undirected())]
n = len(ccs)
w = ceil(sqrt(n))
h = ceil(n / w)
for i in xrange(n):
    subplot(h, w, i + 1)
    nx.draw(nx.subgraph(g, ccs[i]), with_labels = True)

show()
```

The result is shown in Fig. 12.2.1. From this visualization, we learn that there are two major basins of attraction with 36 other minor ones. The inside of those two major basins of attraction is packed and quite hard to see, but if you zoom into their central parts using pylab's interactive zoom-in feature (available from the magnifying glass button on the plot window), you will find that

their attractors are “0” (= [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], all zero) and “511” (= [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], all one). This means that this system has a tendency to converge to a consensus state, either 0 or 1, sensitively depending on the initial condition. Also, you can see that there are a number of states that don’t have any predecessors. Those states that can’t be reached from any other states are called *“Garden of Eden”* states in CA terminology.

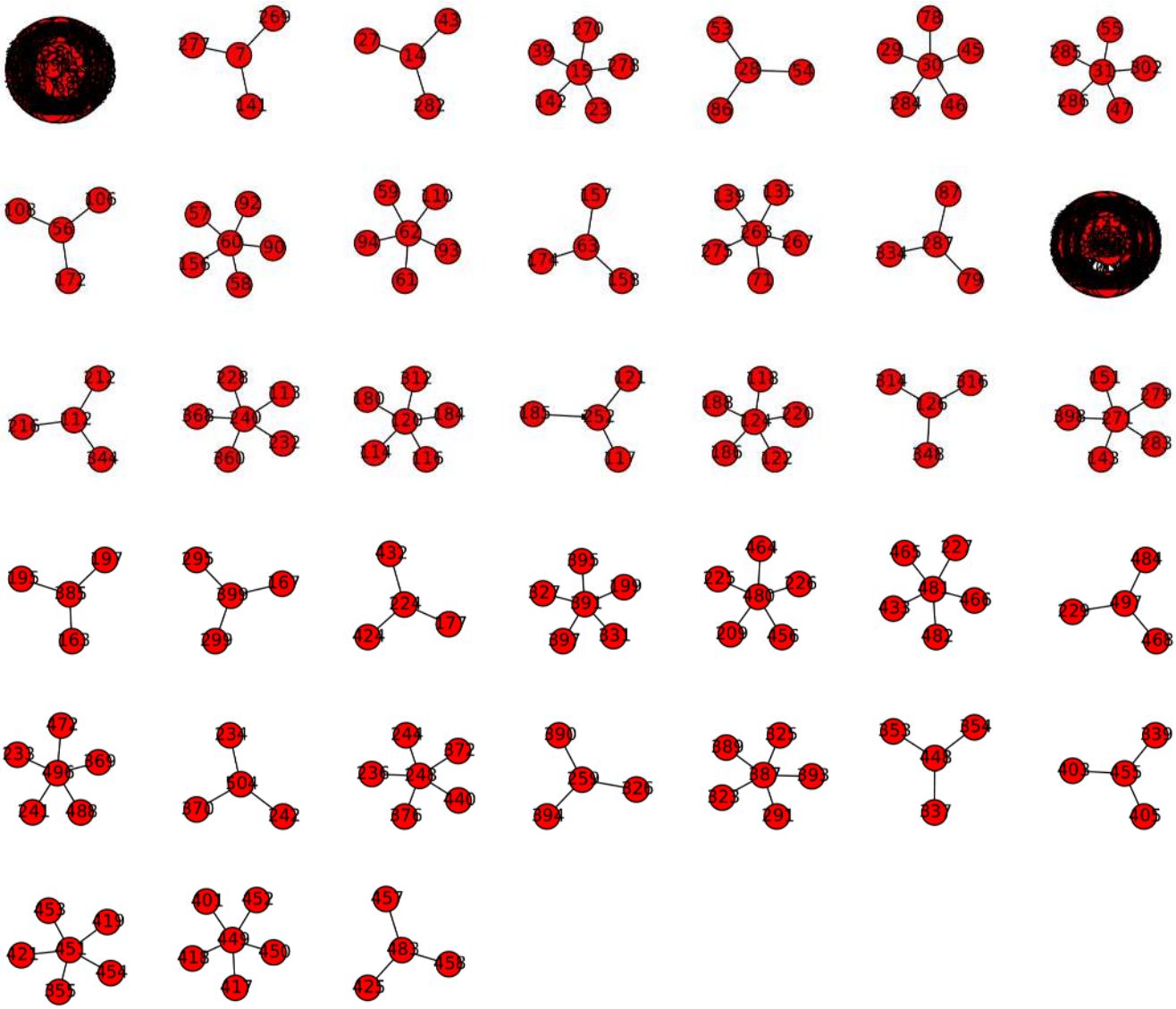


Figure **12.2.1**: *Graph-based phase space of the 1-D binary CA model with the majority rule* $(r = 2, L = 9)$ *drawn with Code 12.3.*

? Exercise 12.2.1:

Measure the number of states in each of the basins of attraction shown in Fig. 12.2.1, and draw a pie chart to show the relative sizes of those basins. Look up matplotlib’s online references to find out how to draw a pie chart. Then discuss the findings.

? Exercise 12.2.2:

Modify Code 12.3 to change the state-transition function to the “minority”rule, so that each cell changes its state to a local minority state. Visualize the phase space of this model and discuss the differences between this result and Fig. 12.2.1

The technique we discussed in this section is still naïve and may not work for more complex CA models. If you want to do more advanced phase space visualizations of CA and other discrete dynamical systems, there is a free software tool named “Discrete

Dynamics Lab" developed by Andrew Wuensche [47], which is available from http://www.ddlab.com/.

This page titled 12.2: Phase Space Visualization is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

12.3: Mean-Field Approximation

Behaviors of CA models are complex and highly nonlinear, so it isn't easy to analyze their dynamics in a mathematically elegant way. But still, there are some analytical methods available. *Mean-field approximation* is one such analytical method. It is a powerful analytical method to make a rough prediction of the macroscopic behavior of a complex system. It is widely used to study various kinds of large-scale dynamical systems, not just CA. Having said that, it is often misleading for studying complex systems (this issue will be discussed later), but as a first step of mathematical analysis, it isn't so bad either.

In any case, the primary reason why it is so hard to analyze CA and other complex systems is that they have a large number of dynamical variables. Mean-field approximation drastically reduces this high dimensionality down to just a few dimensions (!) by reformulating the dynamics of the system in terms of the "average state" of the system (Fig.12.3.2). Then the dynamics are re-described in terms of how each individual cell interacts with this average state and how the average state itself changes over time. In so doing, it is assumed that every cell in the entire space chooses its next state independently, according to the probabilities determined by the average state of the system. This hypothetical, homogeneous, probabilistic space is called the *mean field*, from which the name of the approximation method was derived. Averaging the whole system is equivalent to randomizing or ignoring spatial relationships among components, so you can say that mean-field approximation is a technique to approximate spatial dynamics by non-spatial ones.

Let's work on an example. Consider applying the mean-field approximation to a 2-D binary CA model with the majority rule on Moore neighborhoods. When we apply meanfield approximation to this model, the size of the space no longer matters, because, no

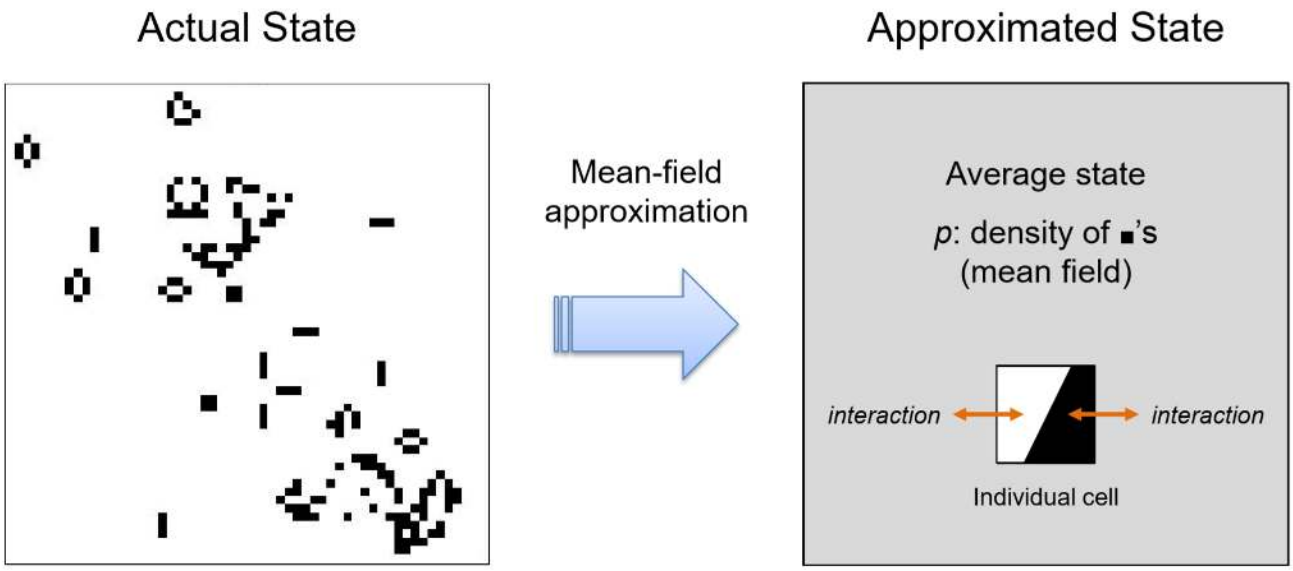


Figure **12.3.1**: *Basic idea of the mean-field approximation.*

matter how large the space is, the system's state is approximated just by one variable: the density of 1's, p_t. This is the mean field, and now our task is to describe its dynamics in a difference equation.

When we derive a new difference equation for the average state, we no longer have any specific spatial configuration; everything takes place probabilistically. Therefore, we need to enumerate all possible scenarios of an individual cell's state transition, and then calculate the probability for each scenario to occur.

Table 12.3.1 lists all possible scenarios for the binary CA with the majority rule. The probability of each state transition event is calculated by (probability for the cell to take the "Current state") × (probability for the eight neighbors to be in any of the "Neighbors' states"). The latter is the sum of (number of ways to arrange k 1's in 8 cells)×(probability for k cells to be 1)×(probability for $8-k$ cells to be 0) over the respective range of k. You may have learned about this kind of combinatorial calculation of probabilities in discrete mathematics and/or probability and statistics.

? Exercise 12.3.1:

Confirm that the probabilities listed in the last column of Table 12.3.1 are a valid probability distribution, i.e., that the sum of them is 1.

To write a difference equation of p_t, there are only two scenarios we need to take into account: the second and fourth ones in Table 12.3.1, whose next state is 1. This is

Table **12.3.1**: *Possible scenarios of state transitions for binary CA with the majority rule.*

Current state	Neighbors' states	Next state	Probability of this transition
0	Four 1's or fewer	0	$(1-p)\sum_{k=0}^{4}\binom{8}{k}p^k(1-p)^{(8-k)}$
0	Five 1's or more	1	$(1-p)\sum_{k=5}^{8}\binom{8}{k}p^k(1-p)^{(8-k)}$
1	Three 1's or fewer	0	$p\sum_{k=0}^{3}\binom{8}{k}p^k(1-p)^{(8-k)}$
1	Four 1's or more	1	$p\sum_{k=4}^{8}\binom{8}{k}p^k(1-p)^{(8-k)}$

because the next value of the average state, p_{t+1}, is the probability for the next state to be 1. Therefore, we can write the following difference equation (the subscript of p_t is omitted on the right hand side for simplicity):

$$p_{t+1} = (1-p)\sum_{k=5}^{8}\binom{8}{k}p^k(1-p)^{(8-k)} + p\sum_{k=4}^{8}\binom{8}{k}p^k(1-p)^{(8-k)} \tag{12.3.1}$$

$$= \sum_{k=5}^{8}\binom{8}{k}p^k(1-p)^{(8-k)} + \binom{8}{4}p^4(1-p)^4 \tag{12.3.2}$$

$$= \binom{8}{5}p^5(1-p)^3 + \binom{8}{6}p^6(1-p)^2 + \binom{8}{7}p^7(1-p) + \binom{8}{8}p^8 + 70p^5(1-p)^4 \tag{12.3.3}$$

$$= 56p5(1-p)^3 + 28p^6(1-p)^2 + 8p^7(1-p) + p^8 + 70p^5(1-p)^4 \tag{12.3.4}$$

$$= 70p^9 - 315p^8 + 540p^7 - 420p^6 + 126p^5 \tag{12.3.5}$$

This result may still seem rather complicated, but it is now nothing more than a one-dimensional nonlinear iterative map, and we already learned how to analyze its dynamics in Chapter 5. For example, we can draw a *cobweb plot* of this iterative map by replacing the function `f(x)` in Code 5.4 with the following (you should also change `xmin` and `xmax` to see the whole picture of the cobweb plot):

Code 12.4: cobweb-plot-for-mfa.py

```
def f(x):
    return 70*x**9 - 315*x**8 + 540*x**7 - 420*x**6 + 126*x**5
```

This produces the cobweb plot shown in Fig. 12.3.2. This plot clearly shows that there are three equilibrium points ($p = 0$, $1/2$, and 1), $p = 0$ and 1 are stable while $p = 1/2$ is unstable, and the asymptotic state is determined by whether the initial value is below or above $1/2$. This prediction makes some sense in view of the nature of the state-transition function (the majority rule); interaction with other individuals will bring the whole system a little closer to the majority choice, and eventually everyone will agree on one of the two choices.

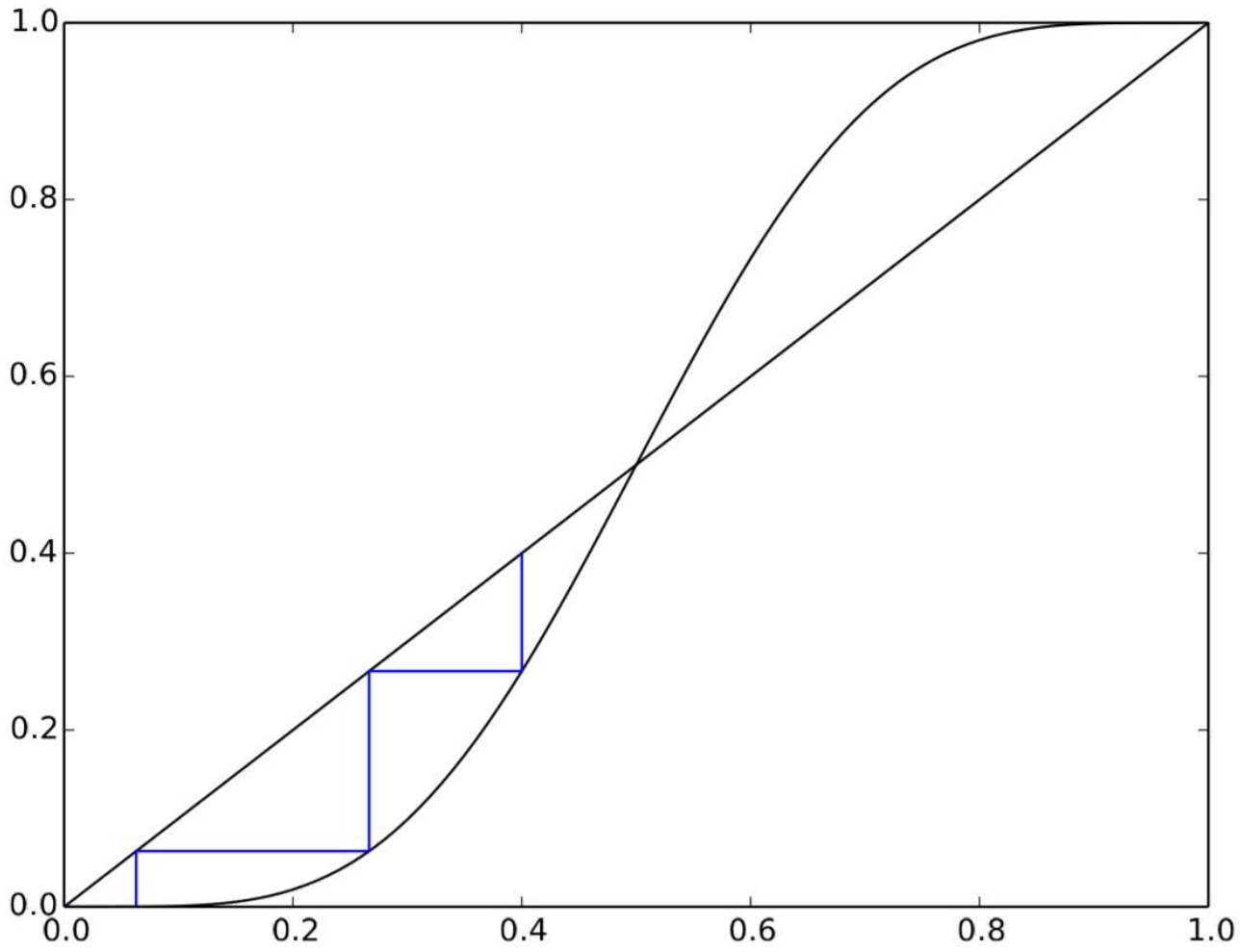


Figure* 12.3.1: Cobweb plot of Equation* 12.3.5*.*

However, we should note that the prediction made using the mean-field approximation above doesn't always match what actually happens in spatially explicit CA models. In simulations, you often see clusters of cells with the minority state remaining in space, making it impossible for the whole system to reach a unanimous consensus. This is because, after all, mean-field approximation is no more than an approximation. It produces a prediction that holds only in an ideal scenario where the spatial locality can be ignored and every component can be homogeneously represented by a global average, which, unfortunately, doesn't apply to most real-world spatial systems that tend to have nonhomogeneous states and/or interactions. So you should be aware of when you can apply mean-field approximation, and what are its limitations, as summarized below:

> Mean-field approximation is a technique that ignores spatial relationships among components. It works quite well for systems whose parts are fully connected or randomly interacting with each other. *It doesn't work if the interactions are local or non-homogeneous, and/or if the system has a non-uniform pattern of states.* In such cases, you could still use mean-field approximation as a preliminary, "zeroth-order" approximation, but you should not derive a final conclusion from it.

In some sense, mean-field approximation can serve as a reference point to understand the dynamics of your model system. If your model actually behaves the same way as predicted by the mean-field approximation, that probably means that the system is not quite complex and that its behavior can be understood using a simpler model.

? Exercise 12.3.2:

Apply mean-field approximation to the Game of Life 2-D CA model. Derive a difference equation for the average state density p_t, and predict its asymptotic behavior. Then compare the result with the actual density obtained from a simulation result. Do they match or not? Why?

This page titled 12.3: Mean-Field Approximation is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

12.4: Renormalization Group Analysis to Predict Percolation Thresholds

The next analytical method is for studying critical thresholds for percolation to occur in spatial contact processes, like those in the epidemic/forest fire CA model discussed in Section 11.5. The percolation threshold may be estimated analytically by a method called *renormalization group analysis*. This is a serious mathematical technique developed and used in quantum and statistical physics, and covering it in depth is far beyond the scope of this textbook (and beyond my ability anyway). Here, we specifically focus on the basic idea of the analysis and how it can be applied to specific CA models.

In the previous section, we discussed mean-field approximation, which defines the average property of a whole system and then describes how it changes over time. Renormalization group analysis can be understood in a similar lens—it defines a certain property of a "portion" of the system and then describes how it changes over "scale," not time. We still end up creating an iterative map and then studying its asymptotic state to understand macroscopic properties of the whole system, but the iterative map is iterated over spatial scales.

I am well aware that the explanation above is still vague and somewhat cryptic, so let's discuss the detailed steps of how this method works. Here are typical steps of how renormalization group analysis is done:

1. Define a property of a "portion" of the system you are interested in. This property must be definable and measurable for portions of any size, like a material property of matter. For analyzing percolation, it is typically defined as the probability for a portion to conduct a fire or a disease from one side to another side through it.
2. Calculate the property at the smallest scale, p_1. This is usually at the single-cell level, which should be immediately obtainable from a model parameter.
3. Derive a mathematical relationship between the property at the smallest scale, p_1, and the same property at a one-step larger scale, $p_2 = \Phi(p_1)$. This derivation is done by using single cells as building blocks to describe the process at a larger scale (e.g., two-by-two blocks)
4. Assume that the relationship derived above can be applied to predict the property at even larger scales, and then study the asymptotic behavior of the iterative map $p_{s+1} = \Phi(p_s)$ when scale s goes to infinity.

Let's see what these steps actually look like with a specific forest fire CA model with Moore neighborhoods. The essential parameter of this model is, as discussed in Section 11.5, the density f trees in a space. This tree density was called p in the previous chapter, but let's rename it q here, in order to avoid confusion with the property to be studied in this analysis. The key property we want to measure is whether a portion of the system (a block of the forest in this context) can conduct fire from one side to another. This probability is defined as a function of scale s, i.e., the length of an edge of the block (Fig. 12.4.1). Let's call this the conductance probability for now.

The conductance probability at the lowest level, p_1, is simply the probability for a tree to be present in a single cell (i.e.,a 1×1 block). If there is a tree, the fire will be conducted, but if not, it won't. Therefore:

$$p_1 = q \tag{12.4.1}$$

Next, we calculate the conductance probability at a little larger scale,p_2. In so doing, we use p_1 as the basic property of a smaller-sized building block and enumerate what are their arrangements that conduct the fire across a two-cell gap (Fig. 12.4.2).

As you see in the figure, if all four cells are occupied by trees, the fire will definitely reach the other end. Even if there are only three or two trees within the 4 area, there are

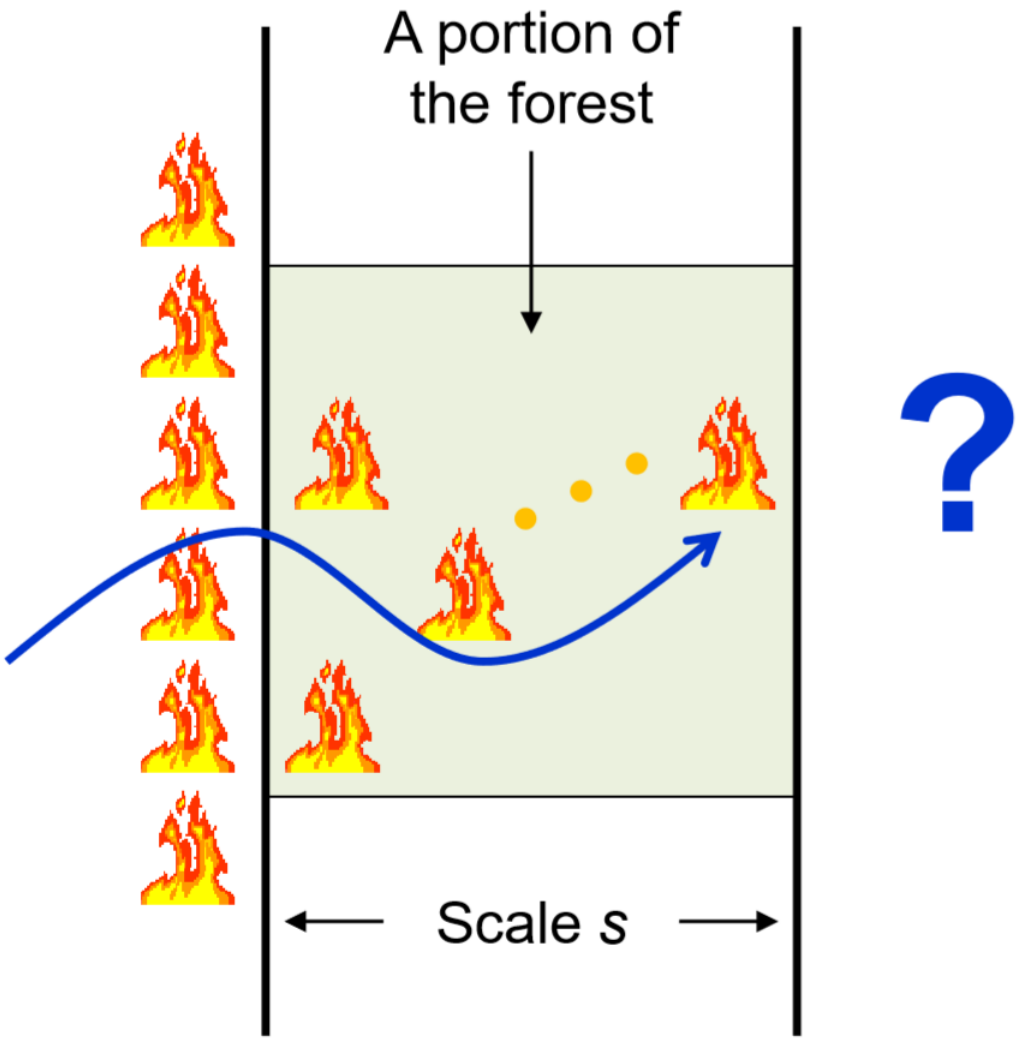


Figure 12.4.1: *Key property to be analyzed in the renormalization group analysis of the epidemic/forest fire CA model, i.e., the probability for a portion of the forest (of scale s) to conduct the fire from one side to another.*

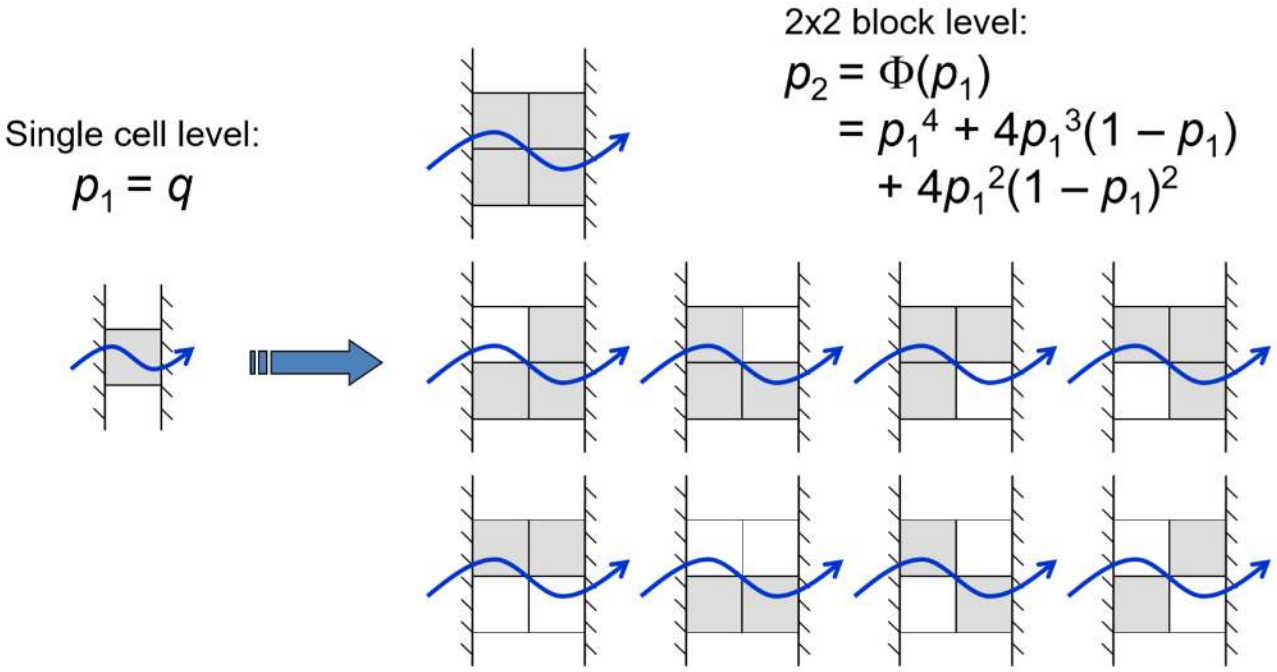


Figure 12.4.2: *How to estimate the conductance probability p_2 at scale $s = 2$ using p_1.*

four different ways to transmit the fire through the area. But if there is only one tree, there is no way to transmit the fire, so there is no need to consider such cases. This is the exhaustive list of possible situations where the fire will be conducted through a scale-2 block. We can calculate the probability for each situation and add them up, to obtain the following relationship between p_1 and p_2:

$$p_2 = \Phi p_1 = p_1^4 + 4p_1^3(1 - p_1) + 4_1^2(1 - p_1)^2 \tag{12.4.2}$$

And this is the place where a very strong assumption/approximation comes in. Let's assume that the relationship above applies to 4×4 blocks (Fig. 12.4.3), 8×8 blocks, etc., all the way up to infinitely large portions of the forest, so that

$$p_{n+1} = \Phi p_s = p_s^4 + 4p_s^3(1 - p_s) + 4p_s^2(1 - p_s)^2 \text{ for all s.} \tag{12.4.3}$$

Of course this is not correct, but some kind of approximation needs to be made in order to study complex systems analytically.

? Exercise 12.4.1:

Why is it not correct to assume that the relationship between the 1×1 blocks and the 2×2 blocks can be applied to 2×2 and 4×4 (and larger)? Look at Fig. 12.4.3 carefully and figure out some configurations that violate this assumption.

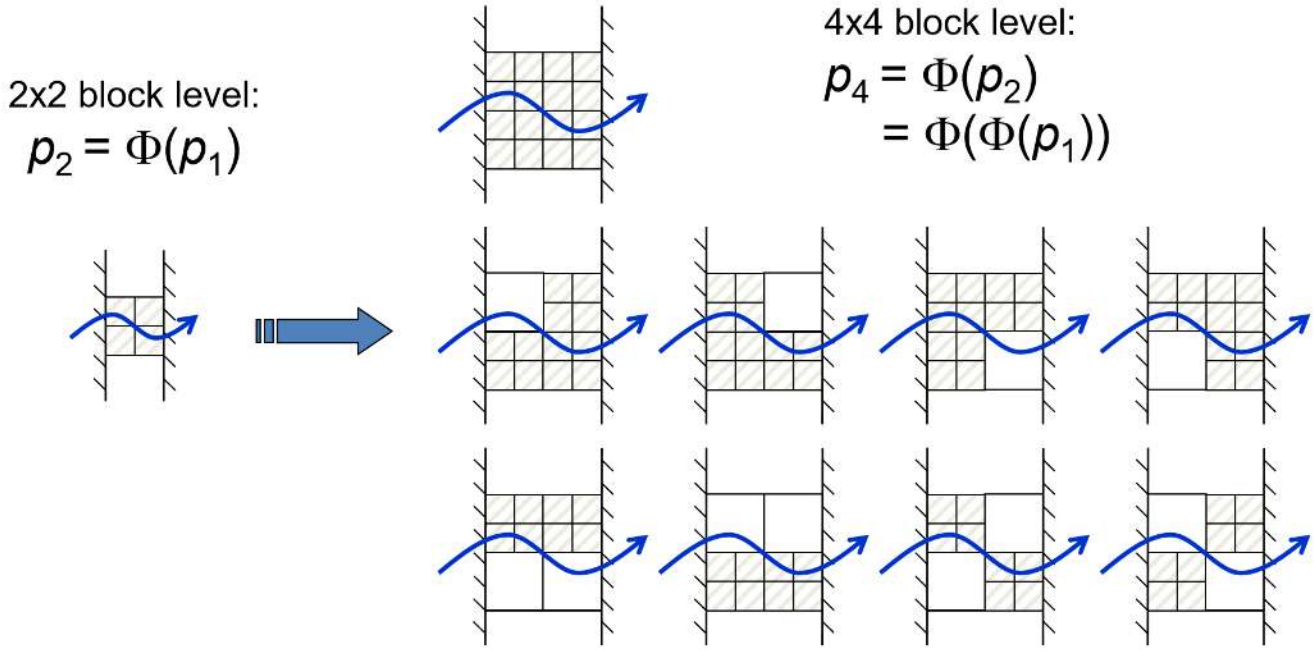


Figure **12.4.3**: *How to estimate the conductance probability p_4 at scale $s = 4$ using p_2.*

Anyway, now we have a relatively simple iterative map of p_s, i.e., the conductance probability for a portion of the forest at scale s. Its asymptotic value, p_∞, tells us whether the fire will propagate over indefinite lengths of distance. You can draw a cobweb plot of Eq.12.4.3 again using Code 5.4, by replacing the function `f(x)` with the following(again, make sure to also change `xmin` and `xmax` to see the entire cobweb plot):

Code 12.5: cobweb-plot-for-rga.py

```
def f(x):
    return x**4 + 4*x**3*(1-x) + 4*x**2*(1-x)**2
```

This will produce the cobweb plot shown in Fig. 12.4.4. This looks quite similar to what we had from the mean-field approximation before, but note that this is not about dynamics over time, but about relationships between scales. It is clearly observed in the figure that there are two asymptotic states possible: $p_\infty = 0$ and $p_\infty = 1$. The former means that the conductance probability is 0 at the macroscopic scale, i.e., percolation doesn't occur, while the latter means that the conductance probability is 1 at the macroscopic scale, i.e., percolation does occur.

Which way the system goes is determined by where you start plotting the cob webplot, i.e., $p_1 = q$, which is the tree density in the forest. The critical percolation threshold, p_c, is seen in Fig. 12.4.4 as an unstable equilibrium point around $p = 0.4$. The exact value can be obtained analytically as follows:

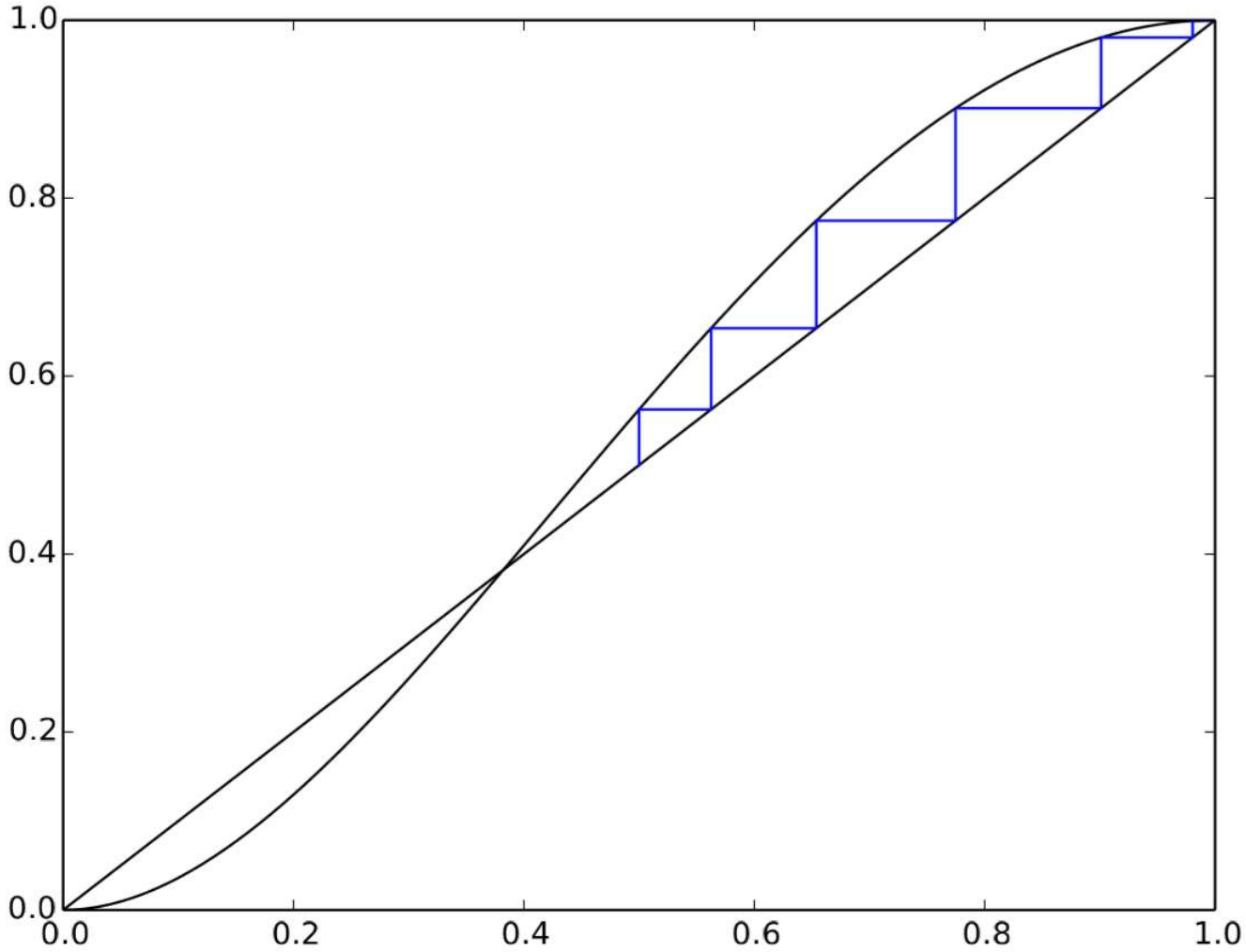


Figure 12.4.4: *Cobweb plot of Eq.* 12.4.3.

$$p_c = \psi(p_c) = p_c^4 + 4p_c^3(1-p_c) + 4p_c^2(1-p_c)^2 \tag{12.4.4}$$

$$0 = p_c(p_c^3 + 4p_c^2(1-p_c) + 4p - c(1-p_c)^2 - 1) \tag{12.4.5}$$

$$0 = p_c(1-p_c)(-p_c^2 - pc - 1 + 4p_c^2 + 4p_c(1-p_c)) \tag{12.4.6}$$

$$0 = p_c(1-p_c)(-1 + 3p_c - p_c^2) \tag{12.4.7}$$

$$p_c = 0, 1, \frac{3 \pm \sqrt{5}}{2} \tag{12.4.8}$$

Among those solutions, $p_c = (3 - \sqrt{5})/2 \approx 0.382$ is what we are looking for.

So,the bottom lineis,if the tree density in the forest is below 38%,the burned area will remain small, but if it is above 38%, almost the entire forest will be burned down. You can check if this prediction made by the renormalization group analysis is accurate or not by carrying out systematic simulations. You should be surprised that this prediction is pretty good; the percolation indeed occurs for densities above about this threshold!

? Exercise 12.4.2:

Estimate the critical percolation threshold for the same forest fire model but with von Neumann neighborhoods. Confirm the analytical result by conducting simulations.

? Exercise 12.4.3:

What would happen if the space of the forest fire propagation were 1-D or 3-D? Conduct the renormalization group analysis to see what happens in those cases.

This page titled 12.4: Renormalization Group Analysis to Predict Percolation Thresholds is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

13: Continuous Field Models I - Modeling

13.1: Continuous Field Models with Partial Differential Equations

13.2: Fundamentals of Vector Calculus

13.3: Visualizing Two-Dimensional Scalar and Vector Field

13.4: Modeling Spatial Movement

13.5: Simulation of Continuous Field Models

13.6: Reaction-Diffusion Systems

This page titled 13: Continuous Field Models I - Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.1: Continuous Field Models with Partial Differential Equations

Spatio-temporal dynamics of complex systems can also be modeled and analyzed using *partial differential equations* (PDEs), i.e., differential equations whose independent variables include not just time, but also space. As a modeling framework, PDEs are much older than CA. But interestingly, the applications of PDEs to describe self-organizing dynamics of spatially extended systems began about the same time as the studies of CA. As discussed in Section 11.5, Turing's monumental work on the chemical basis of morphogenesis [44] played an important role in igniting researchers' attention to the PDE-based continuous field models as a mathematical framework to study self-organization of complex systems.

There are many different ways to formulate a PDE-based model, but here we stick to the following simple first-order mathematical formulation:

$$\frac{\partial f}{\partial t} = F(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \ldots, x, t) \tag{13.1.1}$$

Now the partial derivatives (e.g., $\partial f/\partial t$) have begun to show up in the equations, but don't be afraid; they are nothing different from ordinary derivatives (e.g., df/dt). Partial derivatives simply mean that the function being differentiated has more than one independent variable (e.g., x, t) and that the differentiation is being done while other independent variables are kept as constants. The above formula is still about instantaneous change of something over time (as seen on the left hand side), which is consistent with what we have done so far, so you will find this formulation relatively easy to understand and simulate.

Note that the variable x is no longer a state variable of the system, but instead, it represents a position in a continuous space. In the meantime, the state of the system is now represented by a function, or a field $f(x,t)$, which is defined over a continuous space as well as over time (e.g., Fig. 13.1). The value of function f could be scalar or vector. Examples of such continuous fields include population density (*scalar field*) and flows of ocean currents (*vector field*).

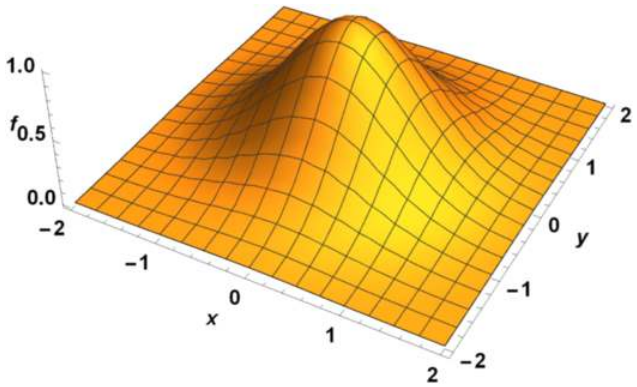


Figure **13.1.1**: Example of a spatial function (field) defined over a two-dimensional space. The function $f(x,y) = e^{-(x^2+y^2)}$ is shown here.

In Equation 13.1.1, the right hand side may still contain space x and time t), which means that this may be a non-autonomous equation. But as we already know, non-autonomous equations can be converted into autonomous equations using the trick discussed in Section 6.2. Therefore, we will focus on autonomous models in this chapter.

As you can see on the right hand side of Equation 13.1.1, the model can now include *spatial derivatives* of the field, which gives information about how the system's state is shaped spatially. The interaction between spatial derivatives (shape, structure) and temporal derivatives (behavior, dynamics) makes continuous field models particularly interesting and useful.

In a sense, using a continuous function as the system state means that the number of variables (i.e., the degrees of freedom) in the system is now *infinite*. You must be proud to see this milestone; we have come a long way to reach this point, starting from a single variable dynamical equation, going through CA with thousands of variables, to finally facing systems with infinitely many variables!

But of course, we are not actually capable of modeling or analyzing systems made of infinitely many variables. In order to bring these otherwise infinitely complex mathematical models down to something manageable for us who have finite intelligence and

lifespan, we usually assume the *smoothness* of function f. This is why we can describe the shape and behavior of the function using well-defined derivatives, which may still allow us to study their properties using analytical means[1].

[1]But we should also beaware that not all physically important, interesting systems can be represented by smooth spatial functions. For example, electromagnetic and gravitational fields can have singularities where smoothness is broken and state values and/or their derivatives diverge to infinity. While such singularities do play an important role in nature, here we limit ourselves to continuous, smooth spatial functions only.

This page titled 13.1: Continuous Field Models with Partial Differential Equations is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.2: Fundamentals of Vector Calculus

In order to develop continuous field models, you need to know some basic mathematical concepts developed and used in vector calculus. A minimalistic quick review of those concepts is given in the following.

Definition: Contour

A *contour* is a set of spatial positions $\boldsymbol{x}$ that satisfy

$$f(x) = C \tag{13.2.1}$$

for a scalar field $\boldsymbol{f}$, where $\boldsymbol{C}$ is a constant (see an example in Figure 13.2.1). Contours are also called level curves or surfaces.

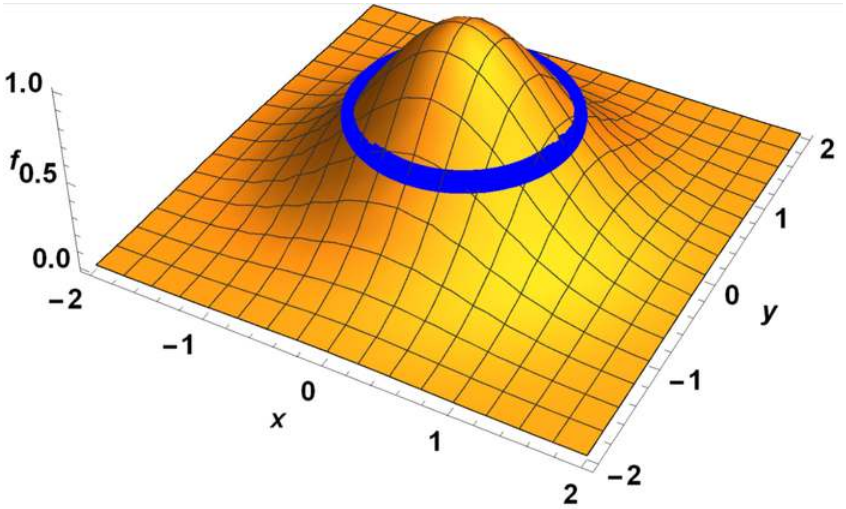


Figure 13.2.1: Contour $f(x, y) = 1/2$ (blue solid circle) of a spatial function $f(x, y) = e^{-(x^2+y^2)}$.

Definition: Gradient

A *gradient* of a scalar field $\boldsymbol{f}$ is a vector locally defined as

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{13.2.2}$$

Here, $\boldsymbol{n}$ is the number of dimensions of the space.

The symbol ∇ is called "*del*" or "*nabla*," which can be considered the following "vector" of differential operators:

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{pmatrix} \tag{13.2.3}$$

A gradient of $\boldsymbol{f}$ at position $\boldsymbol{x}$ is a vector pointing toward the direction of the steepest ascending slope of $\boldsymbol{f}$ at $\boldsymbol{x}$. The length of the vector represents how steep the slope is. A vector field of gradient ∇f defined over a scalar field $\boldsymbol{f}$ is often called a *gradient field*

(See an example in Figure **13.2.2**). The gradient is always perpendicular to the contour that goes through $\boldsymbol{x}$ (unless the gradient is a zero vector; compare Figs. 13.2.1 and 13.2.2).

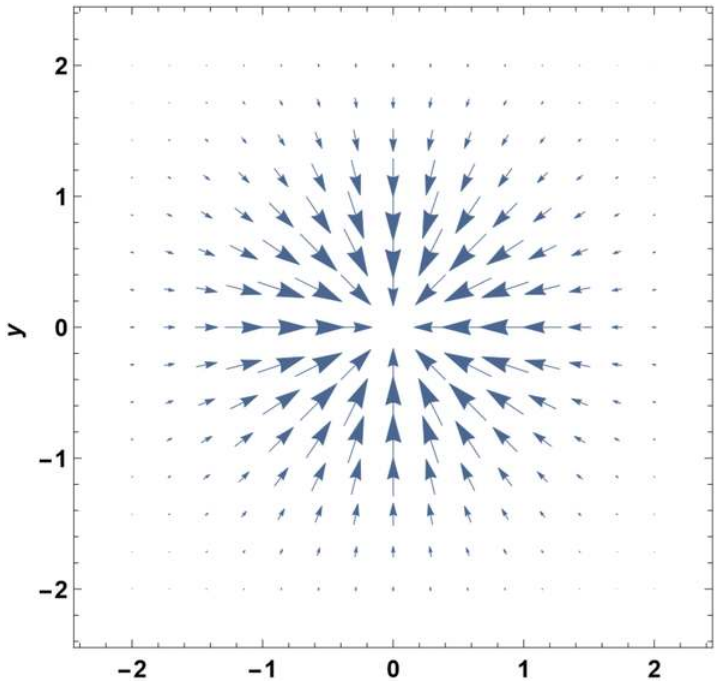


Figure **13.2.2**: Gradient field of a spatial function $f(x, y) = e^{-(x^2+y^2)}$.

Definition: Divergence

A *divergence* of a vector field $\boldsymbol{v}$ is a scalar field defined as

$$\nabla \cdot v = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial^1}{\partial x_2} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{pmatrix}^T \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \ldots + \frac{\partial v_n}{\partial x_n}. \tag{13.2.4}$$

Note the tiny dot between ∇ and $\boldsymbol{v}$, which indicates that this is an "inner product" of them. Don't confuse this divergence with the gradient discussed above!

The physical meaning of divergence is not so straightforward to understand, but anyway, it literally quantifies how much the vector field $\boldsymbol{v}$ is "diverging" from the location $\boldsymbol{x}$. Let's go through an example for a 2-D space. Assume that $\boldsymbol{v}$ is representing flows of some "stuff" moving around in a 2-D space. The stuff is made of a large number of particles (like a gas made of molecules), and the flow $\boldsymbol{v} = (\boldsymbol{v_x}, \boldsymbol{v_y})$ means how many particles are going through a cross-section of the space per area per unit of time. In this 2-D example, a cross-section means a line segment in the space, and the total area of the section is just the length of the line segment.

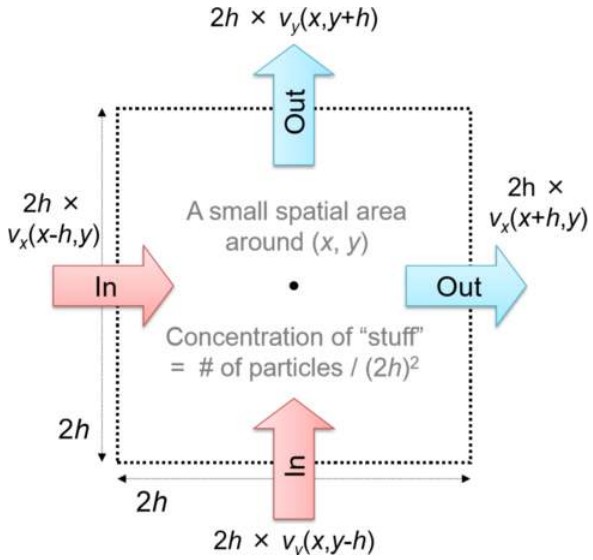


Figure **13.2.3**: Explanation of the physical meaning of divergence in a 2-D space.

Then, consider a temporal change of the number of particles, say N, within a very small spatial area around (x,y) Figure 13.2.3. The temporal change of N can be calculated by counting how many particles are coming into/going out of the area through each edge per unit of time, as

$$\frac{\partial N}{\partial t} = 2hv_x(x-h,y) + 2hv_y(x,y-h) - 2hv_x(x+h,y) - 2hv_y(x,y+h), \tag{13.2.5}$$

where h is the distance from (x,y) to each edge, and therefore the length of each edge is $2h$.

Actually, Figure 13.2.3 may be a bit misleading; the particles can go in and out through any edge in both directions. But we use the $+$ sign for $v_x(x-h,y)$ and $v_y(x,y-h)$ and the $-$ sign for $v_x(x+h,y)$ and $v_y(x,y+h)$ in Eq. (13.2.5), because the velocities are measured using the coordinate system where the rightward and upward directions are considered positive.

Since N depends on the size of the area, we can divide it by the area ($(2h)^2$ in this case) to calculate the change in terms of the concentration of the particles, $c = N/(2h)^2$, which won't depend on h:

$$\frac{\partial c}{\partial t} = \frac{2hv_x(x-h,y) + 2hv_y(x,y-h) - 2hv_x(x+h,y) - 2hv_y(x,y+h)}{(2h)^2} \tag{13.2.6}$$

$$= \frac{v_x(x-h,y) + v_y(x,y-h) - v_x(x+h,y) - v_y(x,y+h)}{2h} \tag{13.2.7}$$

If we make the size of the area really small ($h \to 0$), this becomes the following:

$$\frac{\partial c}{\partial t} = \lim_{h\to 0} \frac{v_x(x-h,y) + v_y(x,y-h) - v_x(x+h,y) - v_x(x+h,y) - v_y(x,y+h)}{2h} \tag{13.2.8}$$

$$= \lim_{h\to 0} \left\{ \left(-\frac{v_x(x+h,y)-v_x(x-h,y)}{2h}\right) + \left(-\frac{v_y(x,y+h)-v_y(x,y-h)}{2h}\right) \right\} \tag{13.2.9}$$

$$= -\frac{\partial v_x}{\partial x} - \frac{\partial v_y}{\partial y} \tag{13.2.10}$$

$$= -\nabla \cdot v \tag{13.2.11}$$

In natural words, this means that the temporal change of a concentration of the stuff is given by a negative divergence of the vector field v that describes its movement. If the divergence is positive, that means that the stuff is escaping from the local area. If the divergence is negative, the stuff is flowing into the local area. The mathematical derivation above confirms this intuitive understanding of divergence

? Exercise 13.2.1

Confirm that the interpretation of divergence above also applies to 3-D cases.

Definition: Laplacian

A *Laplacian* of a scalar field f is a scalar field defined as

$$\nabla^2 f = \nabla \cdot \nabla f = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{pmatrix}^T \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{pmatrix} f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \ldots + \frac{\partial^2 f}{\partial x_n^2}. \tag{13.2.12}$$

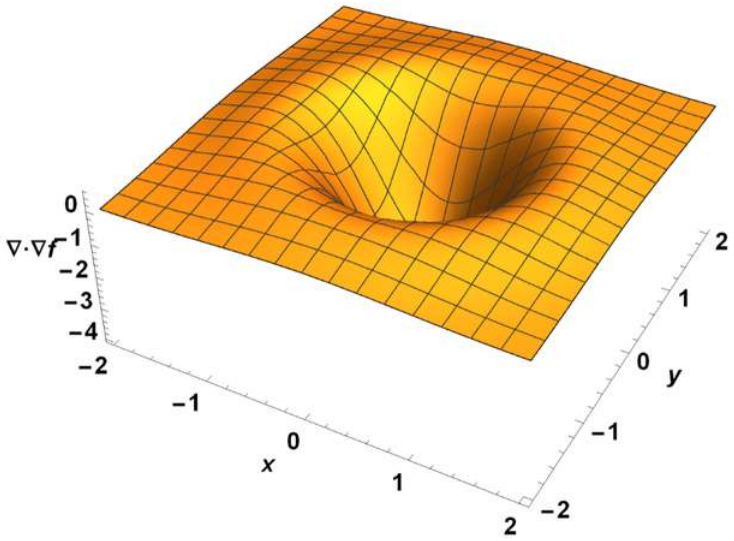


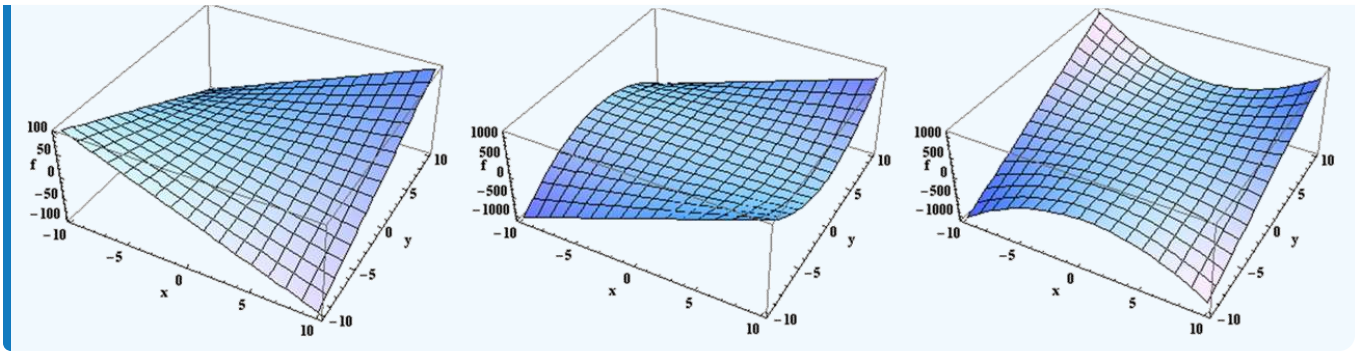
Figure **13.2.4**: Laplacian (i.e., divergence of a gradient field) of a spatial function $f(x, y) = e^{-(x^2+y^2)}$. Compare this figure with Fig. 13.1.

Sometimes the Laplacian operator is denoted by a right side up triangle Δ instead of ∇^2. This is so confusing, I know, but blame the people who invented this notation. In this textbook, we use ∇^2 instead of Δ to denote Laplacians, because ∇^2 is more intuitive to show it is a second-order differential operator, and also because Δ is already used to represent small quantities (e.g., Δx).

The gradient field of f shows a diverging pattern where the scalar field f is concave like a dip or a valley, or a converging pattern where f is convex like a hump or a ridge. Therefore, the Laplacian of f, which is the divergence of the gradient field of f, has a positive value where f is concave, or a negative value where f is convex. This is similar to the second-order derivative of a mathematical function; a concave function has a positive second-order derivative while a convex function has a negative one. The Laplacian is a generalization of the same concept, applicable to functions defined over a multidimensional domain.

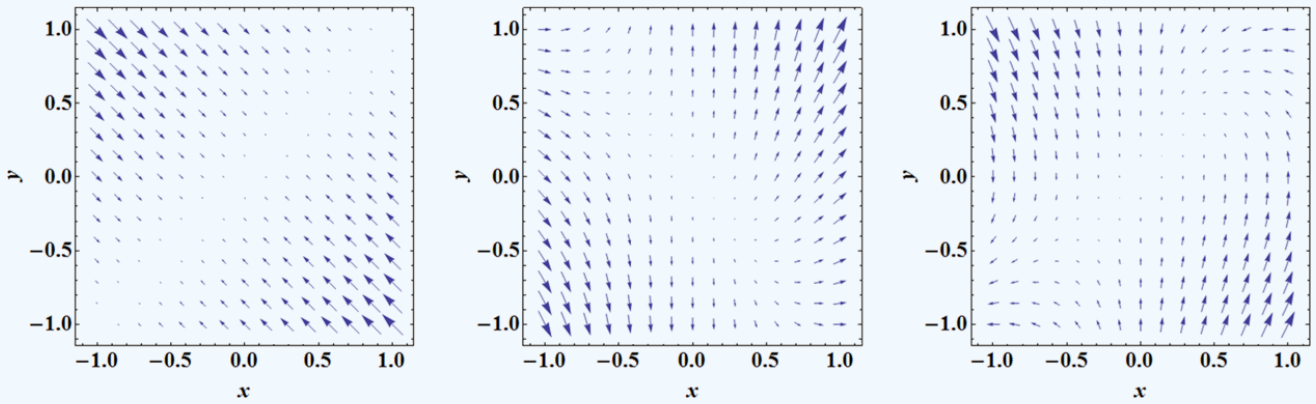
? Exercise 13.2.2

Which of the following surface plots correctly illustrates the shape of a scalar field $f(x, y) = xy(x - 1)$?



Exercise 13.2.3

Which of the following vector field plots correctly illustrates the flows given by a vector field $v(x, y) = (-xy, x - y)$?



Exercise 13.2.4

Calculate the gradient field and the Laplacian of each of the following:

a. $f(x, y) = x^2 + xy$
b. $f(x, y) = e^{x+y}$
c. $f(x, y, z) = \frac{x+y}{z}$

Exercise 13.2.5

Explain where in the $x - y$ space the surface defined by $f(x, y) = 3x^2 - xy^3$ switches between concave and convex.

Exercise 13.2.6

So far, the gradient is defined only for a scalar field, but it is natural to define a gradient for a vector field too. Propose a meaningful definition of ∇v and discuss its physical meaning.

This page titled 13.2: Fundamentals of Vector Calculus is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.3: Visualizing Two-Dimensional Scalar and Vector Field

Plotting scalar and vector fields in Python is straightforward, as long as the space is two-dimensional. Here is an example of how to plot a 3-D surface plot:

Code 13.1: plot-surface-3d.py

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

xvalues, yvalues = meshgrid(arange(-5, 5.5, 0.05), arange(-5, 5.5, 0.05))
zvalues = sin(sqrt(xvalues**2 + yvalues**2))

ax = gca(projection='3d')

ax.plot_surface(xvalues, yvalues, zvalues)

show()
```

The scalar field $f(x, y) = \sin\sqrt{x^2 + y^2}$ is given on the right hand side of the `zvalues` part. The result is shown in Fig. 13.3.1.

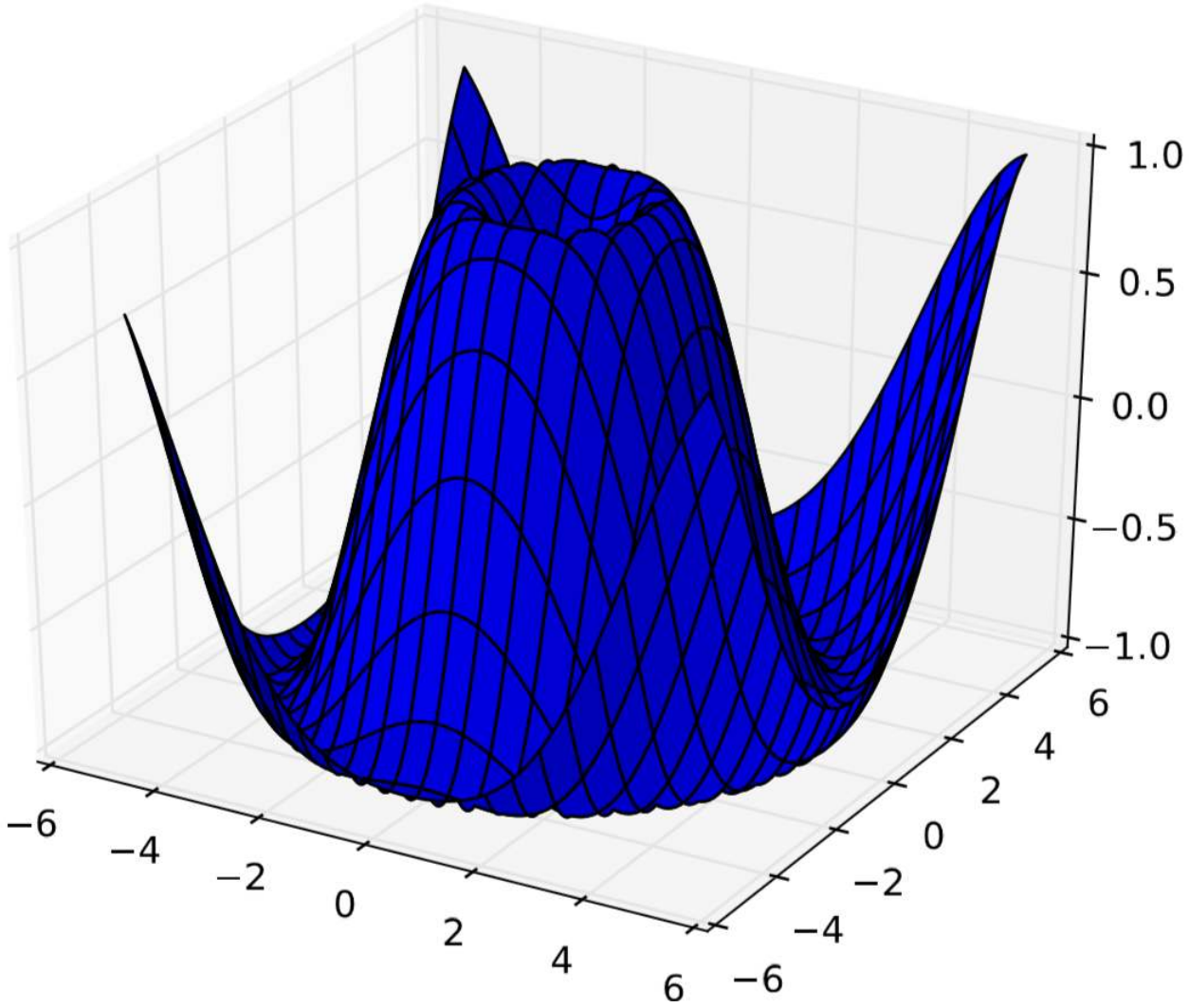


Figure **13.3.1**: *Scalar field visualized as a 3-D surface using Code 13.1.*

And here is how to draw a contour plot of the same scalar field:

Code 13.2: plot-contour.py

```
from pylab import *

xvalues, yvalues = meshgrid(arange(-5, 5.5, 0.05), arange(-5, 5.5, 0.05))
zvalues = sin(sqrt(xvalues**2 + yvalues**2))

cp = contour(xvalues, yvalues, zvalues)
clabel(cp)

show()
```

The `clabel` command is used here to add labels to the contours. The result is shown in Fig. 13.3.2.

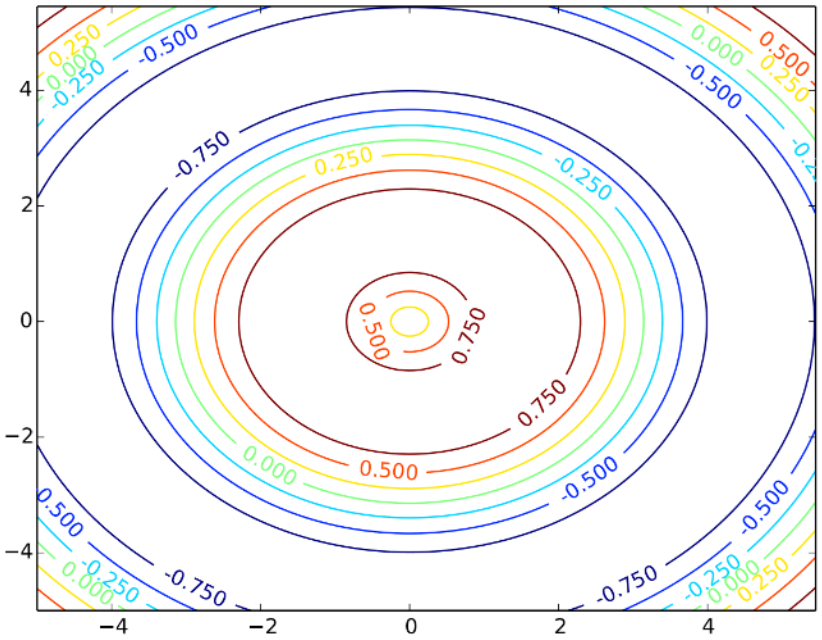


Figure **13.3.2**: *Scalar field visualized as a contour plot using Code 13.2*

If you want more color, you can use `imshow`, which we already used for CA:

Code 13.3: plot-imshow.py

```
from pylab import *

xvalues, yvalues = meshgrid(arange(-5, 5.5, 0.05), arange(-5, 5.5, 0.05))
zvalues = sin(sqrt(xvalues**2 + yvalues**2))

imshow(zvalues)

show()
```

The result is shown in Fig. 13.3.3. Colorful!

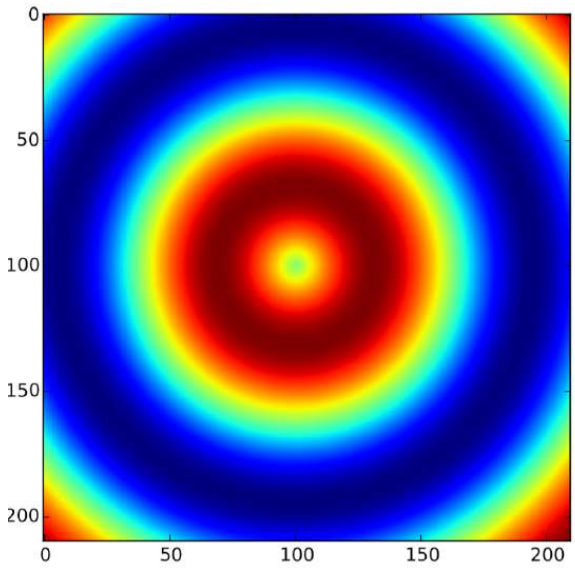


Figure **13.3.3**: *Scalar field visualized as a color image using Code 13.3.*

Finally, a two-dimensional vector field can be visualized using the streamplot function that we used in Section 7.2. Here is an example of the visualization of a vector field v = (vx,vy) = (2x,y−x), with the result shown in Fig. 13.3.4:

Code 13.4: plot-vector-field.py

```
from pylab import *


xvalues, yvalues = meshgrid(arange(-3, 3.1, 0.1), arange(-3, 3.1, 0.1))

vx = 2 * xvalues
vy = yvalues - xvalues

streamplot(xvalues, yvalues, vx, vy)

show()
```

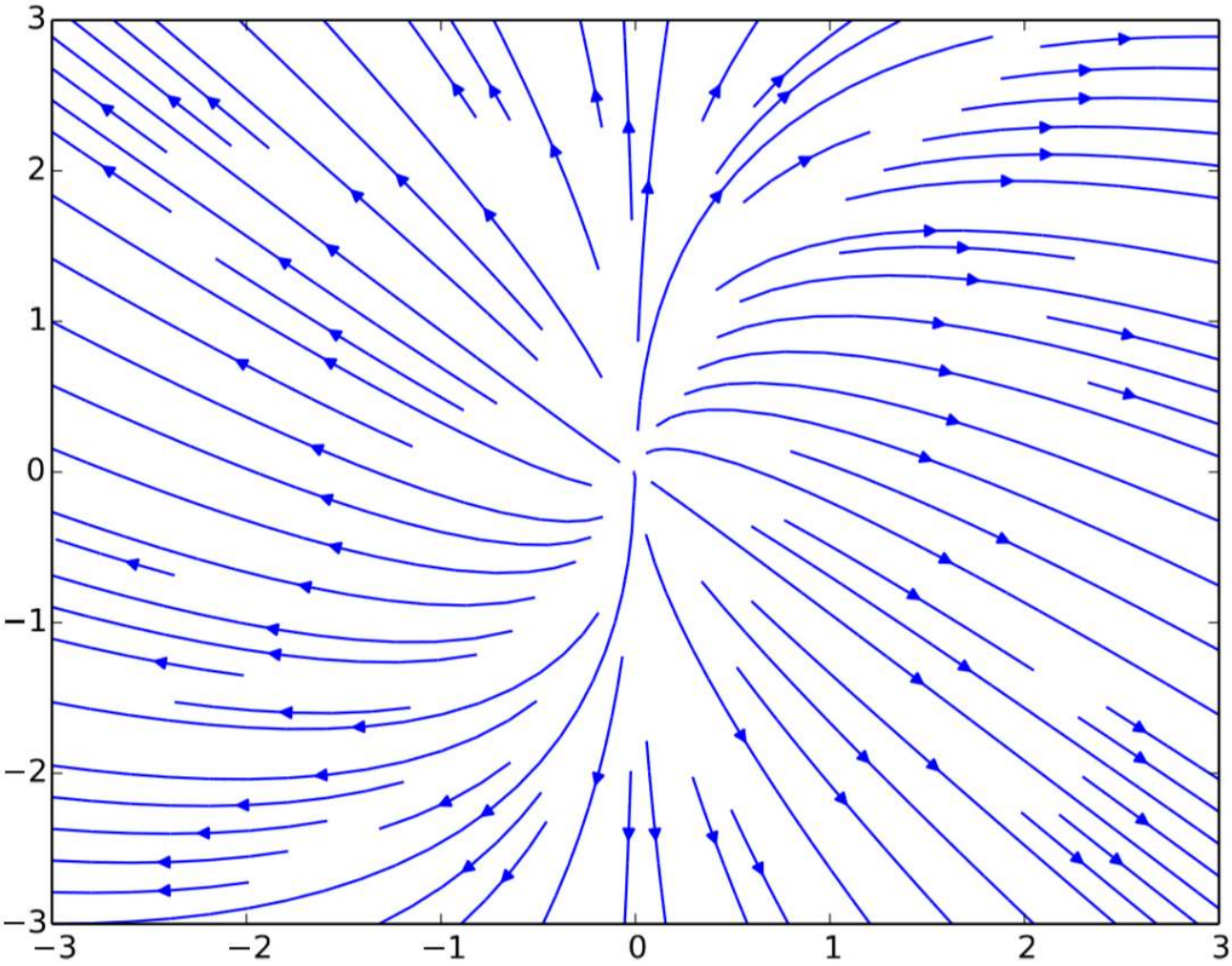


Figure **13.3.4**: *Vector field visualized using Code 13.4.*

? Exercise 13.3.1:

Plot the scalar field $f(x, y) = \sin(xy)$ for $-4 \leq x, y \leq 4$ using Python.

? Exercise 13.3.2:

Plot the gradient field of f$(x, y) = \sin(xy)$ for $-4 \leq x, y \leq 4$ using Python.

? Exercise 13.3.3:

Plot the Laplacian of $f(x, y) = \sin(xy)$ for $-4 \leq x, y \leq 4$ using Python. Compare the result with the outputs of the exercises above.

This page titled 13.3: Visualizing Two-Dimensional Scalar and Vector Field is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.4: Modeling Spatial Movement

Now we will discuss how to write a PDE-based mathematical model for a dynamical process that involves spatial movement of some stuff. There are many approaches to writing PDEs, but here in this textbook, we will use only one "template," called the transport equation. Here is the equation:

$$\frac{\partial c}{\partial t} = -\nabla \cdot J + s \tag{13.4.1}$$

Here, c is the system's state defined as a spatio-temporal function that represents the concentration of the stuff moving in the space. J is a vector field called the *flux* of c. The magnitude of the vector J at a particular location represents the number of particles moving through a cross-section of the space per area per unit of time at that location. Note that this is exactly the same as the vector field v we discussed when we tried to understand the meaning of divergence. Therefore, the first term of Equation 13.4.1 is coming directly from Equation (13.2.11). The second term, s, is often called a *source/sink* term, which is a scalar field that represents any increase or decrease of c taking place locally. This is often the result of influx or outflux of particles from/to the outside of the system. If the system is closed and the total amount of the stuff is conserved, you don't need the source/sink term, and the transport equation becomes identical to Equation (13.2.11).

There is another way of writing a transport equation. While Equation 13.4.1 explicitly describes the amount of particles' movement per area per unit of time in the form of flux J, another convenient way of describing the movement is to specify the velocity of the particles, and then assume that all the particles move at that velocity. Specifically:

$$\frac{\partial c}{\partial t} = -\nabla \cdot (cw) + s \tag{13.4.2}$$

Here, w is the vector field that represents the velocity of particles at each location. Since all the particles are moving at this velocity, the flux is given by $J = cw$. Which formulation you should use, Equation 13.4.1 or Equation 13.4.2, depends on the nature of the system you are going to model.

As you can see above, the transport equation is very simple. But it is quite useful as a starting point when you want to describe various kinds of spatial phenomena. One example is to derive a *diffusion equation*. How can you write a PDE-based model of diffusion dynamics by which the spatial distribution of things tends to smoothen over time (Figure 13.4.1)?

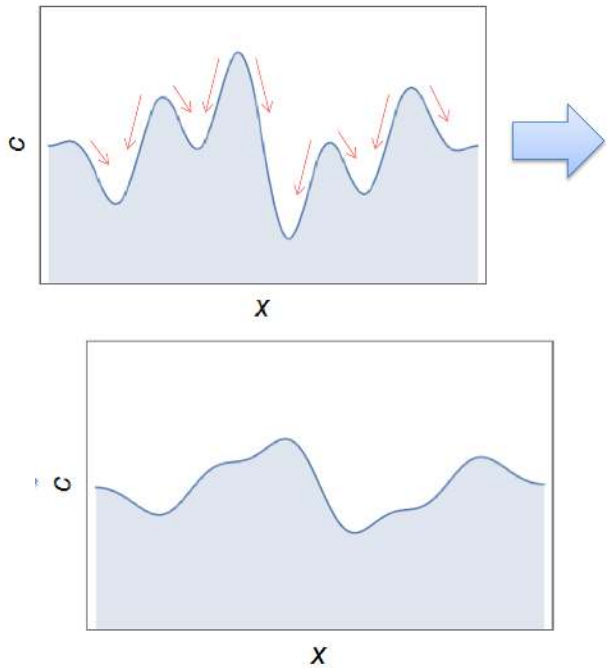


Figure 13.4.1: Schematic illustration of diffusion dynamics.

To develop a model of diffusion using the transport equation, we need to think about the direction of the movement in this process. In diffusion, particles tend to move randomly at microscopic scales, but macroscopically, they tend to "flow" from high to low, bringing down the humps while filling up the dips (Figure 13.4.1, left), eventually resulting in a completely leveled surface. Therefore, a natural choice of the direction of particles' movement is the *opposite* of the gradient of the distribution of the particles ($-\alpha\nabla c$, where α is a positive constant).

The next question you may want to ask is: Which formula should you use, Equation 13.4.1 or Equation 13.4.2? To make a decision, you should think about whether $-\alpha\nabla c$ only gives the velocity of movement or it also determines the actual magnitude of the flux. If the former is the case, all the particles at the same location will move toward that direction, so the actual flux should be given by $-c(\alpha\nabla c)$. But this means that the speed of diffusion will be faster if the whole system is "elevated" by adding 10 to c everywhere. In other words, the speed of diffusion will depend on the elevation of the terrain! This doesn't sound quite right, because the diffusion is a smoothing process of the surface, which should depend only on the shape of the surface and should not be affected by how high or low the overall surface is elevated. Therefore, in this case, Equation 13.4.1 is the right choice; we should consider $-\alpha\nabla c$ the actual flux caused by the diffusion. This assumption is called *Fick's first law of diffusion*, as it was proposed by Adolf Fick in the 19th century.

Plugging $J = -\alpha\nabla c$ into Equation 13.4.1, we obtain

$$\frac{\partial c}{\partial t} = -\nabla\cdot(-\alpha\nabla c) + s \tag{13.4.3}$$

$$= \nabla\cdot(\alpha\nabla c) + s. \tag{13.4.4}$$

If α is a homogeneous constant that doesn't depend on spatial locations, then we can take it out of the parentheses:

$$\frac{\partial c}{\partial t} = \alpha\nabla^2 c + s \tag{13.4.5}$$

Now we see the Laplacian coming out! This is called the *diffusion equation*, one of the most fundamental PDEs that has been applied to many spatio-temporal dynamics in physics, biology, ecology, engineering, social science, marketing science, and many other disciplines. This equation is also called the *heat equation* or *Fick's second law of diffusion*. α is called the *diffusion constant*, which determines how fast the diffusion takes place.

? Exercise 13.4.1

We could still consider an alternative smoothing model in which the flux is given by $-c(\alpha\nabla c)$, which makes the following model equation:

$$\frac{\partial c}{\partial t} = \alpha\nabla\cdot(c\nabla c) + s \tag{13.4.6}$$

Explain what kind of behavior this equation is modeling. Discuss the difference between this model and the diffusion equation (Equation 13.4.5).

? Exercise 13.4.2: circular motion

Develop a PDE model that describes a circular motion of particles around a certain point in space.

? Exercise 13.4.3

Develop a PDE model that describes the attraction of particles to each other.

We can develop more complex continuous field models by combining spatial movement and local dynamics together, and also by including more than one state variable. Let's try developing a PDE-based model of interactions between two state variables: population distribution (p for "people") and economic activity (m for "money"). Their local (non-spatial) dynamics are assumed as follows:

1. The population will never increase or decrease within the time scale being considered in this model.
2. The economy is activated by having more people in a region.
3. The economy diminishes without people.

In the meantime, their spatial dynamics (movements) are assumed as follows:

4. Both the population and the economic activity diffuse gradually.
5. People are attracted to the regions where economy is more active.

Our job is to translate these assumptions into mathematical equations by filling in the boxes below.

$\frac{\partial p}{\partial t} = \square$
$\frac{\partial m}{\partial t} = \square$

First, give it a try by yourself! Then go on to the following.

Were you able to develop your own model? Let's go over all the assumptions to see how we can write each of them in mathematical form.

Assumption 1 says there is no change in the population by default. So, this can be written simply as

$$\frac{\partial m}{\partial t} = \alpha p, \tag{13.4.7}$$

where α is a positive constant that determines the rate of production of the economy per capita.

Assumption 3 tells that the economy should naturally decay if there is no influence from people. The simplest model of such decay would be that of exponential decay, which can be represented by including a linear term of m with a negative coefficient in the equation for m, so that:

$$\frac{\partial m}{\partial t} = \alpha p - \beta m. \tag{13.4.8}$$

Again, β is a positive constant that determines the decay rate of the economy. This equation correctly shows exponential decay if $p = 0$, which agrees with the assumption. So far, all the assumptions implemented are about local, non-spatial dynamics. Therefore we didn't see any spatial derivatives.

Now, Assumption 4 says we should let people and money diffuse over space. This is about spatial movement. It can be modeled using the diffusion equation we discussed above (Equation 13.4.5). There is no need to add source/sink terms, so we just add Laplacian terms to both equations:

$$\frac{\partial p}{\partial t} = D_p \nabla^2 p \tag{13.4.9}$$

$$\frac{\partial m}{\partial t} = D_m \nabla^2 m + \alpha p - \beta m \tag{13.4.10}$$

Here, D_p and D_m are the positive diffusion constants of people and money, respectively.

Finally, Assumption 5 says people can sense the "smell" of money and move toward areas where there is more money. This is where we can use the transport equation. In this case, we can use Equation 13.4.2, because all the people at a certain location would be sensing the same "smell" and thus be moving toward the same direction on average. We can represent this movement in the following transport term

$$-\nabla \cdot (p\gamma \nabla m), \tag{13.4.11}$$

where the gradient of m is used to obtain the average velocity of people's movement (with yet another positive constant γ). Adding this term to the equation for p representspeople's movement toward money.

So, the completed model equations look like this:

$$\frac{\partial p}{\partial t} = D_p \nabla^2 p - \gamma \nabla \cdot (p \nabla m) \tag{13.4.12}$$

$$\frac{\partial m}{\partial t} = D_m \nabla^2 m + \alpha m - \beta m \tag{13.4.13}$$

How does this model compare to yours?

Interestingly, a mathematical model that was essentially identical to our equations above was proposed nearly half a century ago by two physicists/applied mathematicians, Evelyn Keller and Lee Segel, to describe a completely different biological phenomenon [48, 49]. The *Keller-Segel model* was developed to describe *chemotaxis*—movement of organisms toward (or away from) certain chemical signals—of a cellular slime mold species *Dictyostelium discoideum*. These slime mold cells normally behave as individual amoebae and feed on bacteria, but when the food supply becomes low, they first spread over the space and then aggregate to form "slugs" for long-distance migration (Figure 13.4.2). Keller and Segel developed a mathematical model of this aggregation process, where chemotaxis plays a critical role in cellular self-organization.



Figure 13.4.2: Dictyostelium showing aggregation behavior. Image from Wikimedia Commons ("Dictyostelium Aggregation" by Bruno in Columbus – Own work. Licensed under Public domain via Wikimedia Commons – http://commons.wikimedia.org/wiki/File:Dictyostelium_Aggregation.JPG)

Keller and Segel made the following assumptions based on what was already known about these interesting cellular organisms:

- Cells are initially distributed over a space more or less homogeneously (with some random fluctuations).
- Cells show chemotaxis toward a chemical signal called *cAMP* (*cyclic adenosine monophosphate*).
- Cells produce cAMP molecules.
- Both cells and cAMP molecules diffuse over the space.
- Cells neither die nor divide.

Did you notice the similarity between these assumptions and those we used for the population-economy model? Indeed, they are identical, ifyoureplace"cells"with"people" and "cAMP" with "money." We could use the word "moneytaxis" for people's migration toward economically active areas!

The actual Keller-Segel equations look like this:

$$\frac{\partial a}{\partial t} = \mu \nabla^2 \alpha - \chi \nabla \cdot (a \nabla c) \tag{13.4.14}$$

$$\frac{\partial c}{\partial t} D \nabla^2 c + fa - kc \tag{13.4.15}$$

In fact, these equations are simplified ones given in [49], as the original equations were rather complicated with more biological details. a and c are the state variables for cell density and cAMP concentration, respectively. μ is the parameter for cell mobility, χ is the parameter for cellular chemotaxis, D is the diffusion constant of cAMP, f is the rate of cAMP secretion by the cells, and k is the rate of cAMP decay. Compare these equations with Eqs. 13.4.12 and 13.4.13. They are the same! It is intriguing to see that completely different phenomena at vastly distant spatio-temporal scales could be modeled in an identical mathematical formulation.

It is known that, for certain parameter settings (which will be discussed in the next chapter), this model shows *spontaneous pattern formation* from almost homogeneous initial conditions A sample simulation result is shown in Fig. 13.4.3, where we see spots of

aggregated cell clusters forming spontaneously. You will learn how to conduct simulations of continuous field models in the next section.

In Fig. 13.4.3, we can also observe that there is a characteristic distance between nearby spots, which is determined by model parameters (especially diffusion constants). The same observation applies to the formation of cities and towns at geographical scales. When you see a map, you will probably notice that there is a typical distance between major cities, which was probably determined by the human mobility centuries ago, among other factors.

? Exercise 13.4.4

Consider introducing to the Keller-Segel model a new variable b that represents the concentration of a toxic waste chemical. Make the following assumptions:

- cAMP gradually turns into the waste chemical (in addition to natural decay).
- The waste chemical diffuses over space.
- The waste chemical never decays naturally.



Figure **13.4.3**: Simulation of the Keller-Segel model(= the population-economy model). Cell densities are plotted in grayscale (darker = greater). Time flows from left to right.

- Cells tend to avoid regions with high concentrations of the waste chemical.
- Revise the Keller-Segel equations to develop your new model.

? Exercise 13.4.5

Develop a PDE-based model that describes the dynamics of a predator-prey ecosystem with the following assumptions:

- The prey population grows following the logistic growth model.
- The predator population tends to decay naturally without prey.
- Predation (i.e., encounter of prey and predator populations) increases the predator population while it decreases the prey population.
- The predator population tends to move toward areas with high prey density.
- The prey population tends to move away from areas with high predator density.
- Both populations diffuse over space.

This page titled 13.4: Modeling Spatial Movement is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.5: Simulation of Continuous Field Models

Simulation of continuous field models written in PDEs is not an easy task, because it easily involves an enormous amount of computation if you want to obtain fairly accurate simulation results, and also because certain kinds of spatial dynamics may sometimes cause highly sensitive behaviors. In fact, most "supercomputers" built for scientific computation[2] are used to solve large-scale PDE problems, e.g., to simulate global climate systems, complex material properties, etc.

Covering all the advanced techniques of numerical simulations of PDEs is way beyond the scope of this introductory textbook. Instead, we will try a much simpler approach. As we already discussed cellular automata (CA), we will convert PDE-based models into CA by discretizing space and time, and then simulate them simply as CA models with continuous state variables.

Discretizing time in a PDE is nothing different from what we discussed in Section 6.3. By replacing the time derivative by its discretized analog, the original PDE

$$\frac{\partial f}{\partial t} = F(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \dots, x, t) \tag{13.5.1}$$

becomes

$$\frac{\Delta f}{\Delta t} = \frac{f(x, t+\Delta t) - f(x,t)}{\Delta t} = F(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \dots, x, t), \tag{13.5.2}$$

$$f(x, t+\Delta t) = f(x,t) + F(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, \dots, x, t)\Delta t \tag{13.5.3}$$

which is now a difference equation over time (note that Δ above is not a Laplacian!).

We are not done yet, because the right hand side may still contain spatial derivatives ($\partial f/\partial x$, $\partial^2 f/\partial x^2$, ...). We need to discretize them over space too, in order to develop a CA version of this PDE. Here is one way to discretize a first-order spatial derivative:

$$\frac{\partial f}{\partial x} \approx \frac{\Delta f}{\Delta x} = \frac{f(x, t+\Delta t) - f(x,t)}{\Delta x} \tag{13.5.4}$$

This is not incorrect mathematically, but there is one practical issue. This discretization makes the space asymmetric along the x-axis, because the spatial derivative at x depends on $f(x+\Delta x, t)$, but not on $f(x-\Delta x, t)$. Unlike time that has a clear asymmetric direction from past to future, the spatial axis should better be modeled symmetrically between the left and the right (or up and down). Therefore, a better way of performing the spatial discretization is as follows:

$$\frac{\partial f}{\partial x} \approx \frac{2\Delta f}{2\Delta x} = \frac{f(x, t+\Delta t) - f(x,t)}{2\Delta x} \tag{13.5.5}$$

This version treats the left and the right symmetrically.

Similarly, we can derive the discretized version of a second-order spatial derivative, as follows:

$$\begin{aligned}
\frac{\partial^2 f}{\partial x^2} &\approx \frac{\frac{\partial f}{\partial x}\big|_{x+\Delta x} - \frac{\partial f}{\partial x}\big|_{x-\Delta x}}{2\Delta x} \\
& \quad f(x+\Delta x+\Delta x, t) \qquad f(x-\Delta x+\Delta x, t) \\
&\approx \frac{\frac{-f(x+\Delta x-\Delta x, t)}{2\Delta x} - \frac{-f(x-\Delta x-\Delta x, t)}{2\Delta x}}{2\Delta x} \\
&= \frac{f(x+2\Delta x, t) + f(x-2\Delta x, t) - 2f(x,t)}{2\Delta x^2} \\
&= \frac{f(x+\Delta x, t) + f(x-\Delta x, t) - 2f(x,t)}{\Delta x^2} \\
&= \frac{f(x+\Delta x, t) + f(x-\Delta x, t) - 2f(x,t)}{\Delta x^2} \text{by replacing } 2\Delta x \to \Delta x
\end{aligned} \tag{13.5.6}$$

Moreover, we can use this result to discretize a Laplacian as follows:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \ldots + \frac{\partial^2 f}{\partial x_n^2}$$

$$\approx \frac{f(x_1 + \Delta x_1, x_2, \ldots, x_n, t) + f(x_1 - \Delta x_1, x_2, \ldots, x_n, t) - 2f(x_1, x_2, \ldots, x_n, t)}{\Delta x_1^2}$$

$$+ \frac{f(\Delta x_1, x_2, \ldots, x_n, t) + f(\Delta x_1, x_2 - \Delta x_2, \ldots, x_n, t) - 2f(x_1, x_2, \ldots, x_n, t)}{\Delta x_2^2}$$

$$+ \ldots$$

$$+ f(x_1, x_2, \ldots x_n + \Delta x_n, t) + f(x_1, x_2, \ldots, x_n - \Delta x_n, t)$$

$$= (f(x_1 + \Delta x, x_2, \ldots, x_n, t) + f(x_1 - \Delta x, x_2, \ldots x_n, t) - 2f(x_1, x_2, \ldots, x_n, t)$$

$$+ f(x_1, x_2, + \Delta x, \ldots, x_n, t) + f(x_1, x_2, - \Delta x, \ldots, x_n, t)$$

$$+ \ldots$$

$$+ f(x_1, x_2, \ldots, x_n + \Delta x, t) + f(x_1, x_2, \ldots, x_n - \Delta x, t)$$

$$- 2nf(x_1, x_2, \ldots, x_n, t)/\Delta x^2 \tag{13.5.7}$$

(by replacing Δxi $\rightarrow \Delta$x for all i)

The numerator of the formula above has an intuitive interpretation: "Just add the values of f in all of your nearest neighbors (e.g., top, bottom, left, and right for a 2-D space) and then subtract your own value $f(x,t)$ as many times as the number of those neighbors." Or, equivalently: "Measure the difference between your neighbor and yourself, and then sum up all those differences." It is interesting to see that a higher-order operator like Laplacians can be approximated by such a simple set of arithmetic operations once the space is discretized.

You can also derive higher-order spatial derivatives in a similar manner, but I don't think we need to cover those cases here. Anyway, now we have a basic set of mathematical tools (Eqs. 13.5.3, 13.5.5, 13.5.6, 13.5.7) to discretize PDE-based models so we can simulate their behavior as CA models.

Let's work on an example. Consider discretizing a simple transport equation without source or sink in a 2-D space, where the velocity of particles is given by a constant vector:

$$\frac{\partial c}{\partial t} = -\nabla \cdot (cw) \tag{13.5.8}$$

$$w = \begin{pmatrix} w_x \\ w_y \end{pmatrix} \tag{13.5.9}$$

We can discretize this equation in both time and space, as follows:

$$c(x, y, t + \delta t) \approx c(x, y, t) - \nabla \cdot (c(x, y, t)w)\delta t \tag{13.5.10}$$

$$= c(x, y, t) - \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}^T (c(x, y, t) \begin{pmatrix} w_x \\ w_y \end{pmatrix})\Delta t \tag{13.5.11}$$

$$= c(x, y, t) - (w_x \frac{\partial c}{\partial x} + w_y \frac{\partial c}{\partial y})\Delta t \tag{13.5.12}$$

$$\approx c(x, y, t) \tag{13.5.13}$$
$$- (w_x \frac{c(x + \Delta h, y, t) - c(x - \Delta h, y, t)}{2\Delta h} + w_y \frac{c(x, y + \Delta h, t) - c(x, y - \Delta h, t)}{2\Delta h})\Delta t$$

$$= c(x, y, t) - (w_x c(x + \Delta h, y, t) - w_x c(x - \Delta h, y, t) + w_y c(x, y + \Delta h, t) \tag{13.5.14}$$
$$- w_y c(x, y - \Delta h, t)) \frac{\Delta t}{2\Delta h}$$

This is now fully discretized for both time and space, and thus it can be used as a state-transition function of CA. We can implement this CA model in Python, using Code 11.5 as a template. Here is an example of implementation:

Code 13.5: transport-ca.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

n = 100 # size of grid: n * n
Dh = 1. / n # spatial resolution, assuming space is [0,1] * [0,1]
Dt = 0.01 # temporal resolution

wx, wy = -0.01, 0.03 # constant velocity of movement

xvalues, yvalues = meshgrid(arange(0, 1, Dh), arange(0, 1, Dh))
```

```
def initialize():
    global config, nextconfig
    # initial configuration
    config = exp(-((xvalues - 0.5)**2 + (yvalues - 0.5)**2) / (0.2**2))
    nextconfig = zeros([n, n])

def observe():
    global config, nextconfig
    ax = gca(projection = '3d')
    ax.cla()
    ax.plot_surface(xvalues, yvalues, config, rstride = 5, cstride = 5)
    ax.grid(False)
    ax.set_zlim(0, 1)
    show() # <- this is needed to reflect changes made to 3-D axes above
```

```
def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            # state-transition function
            nextconfig[x, y] = config[x, y] - ( wx * config[(x+1)%n, y]
                                               - wx * config[(x-1)%n, y]
                                               + wy * config[x, (y+1)%n]
```

```
                                        - wy * config[x, (y-1)%n])\
          * Dt/(2*Dh)

    config, nextconfig = nextconfig, config

import pycxsimulator
pycxsimulator.GUI(stepSize = 50).start(func=[initialize, observe, update])
```

In this example, we simulate the transport equation with $(w_x, w_y) = (0.01, 0.03)$ in a 2-D `[0,1]×[0,1]` space with periodic boundary conditions, starting from an initial configuration that has a peak in the middle of the space:

$$c(x, y, 0) = exp(-\frac{(x-0.5)^2 + (y+0.5)^2}{0.2^2}) \tag{13.5.15}$$

The temporal and spatial resolutions are set to $\Delta t = 0.01$ and $\Delta h = 0.01$, respectively (so that the size of the grid is `100×100`). This is a very slow simulation process, so the `stepSize` is set to 50 to skip unnecessary visualizations (see the last line). The surface plot is used to visualize the configuration in 3-D axes. Note that you need to call the `show` command at the end of the `observe` function, because changes made to 3-D axes are not automatically reflected to the visualization (at least in the current implementation of matplotlib).

The result of this simulation is shown in Fig. 13.5.1 where you can clearly see the peak being transported to a direction given by (w_x, w_y). Moreover, thanks to the interactive nature of pycxsimulator.py, this 3-D visualization is now interactively manipulatable even if you are not running it from Anaconda Spyder, Enthought Canopy, or other interactive environments. You can click and drag on it to rotate the surface to observe its 3-D structure from various angles.

If you don't need a 3-D surface plot, yo ucan use `imshow` instead. This makes the code a bit simpler, and yet the result can be as good as before (Fig. 13.5.2):

Code 13.6: transport-ca-imshow.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of grid: n * n
Dh = 1. / n # spatial resolution, assuming space is [0,1] * [0,1]
Dt = 0.01 # temporal resolution
```

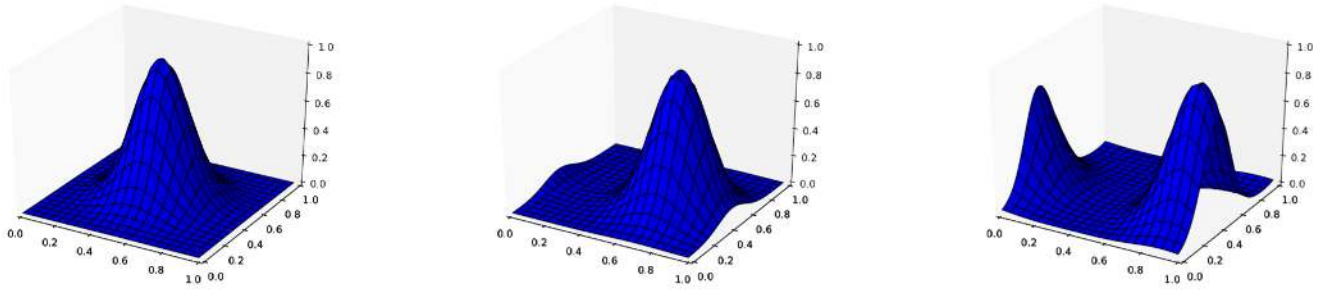


Figure **13.5.1**: *Visual output of Code 13.5. Time flows from left to right.*

```
wx, wy = -0.01, 0.03 # constant velocity of movement

xvalues, yvalues = meshgrid(arange(0, 1, Dh), arange(0, 1, Dh))

def initialize():
    global config, nextconfig
    # initial configuration
    config = exp(-((xvalues - 0.5)**2 + (yvalues - 0.5)**2) / (0.2**2))
    nextconfig = zeros([n, n])
```

```
def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1)

### (the rest is the same as before)
```

One thing I should warn you about is that the simulation method discussed above is still based on the Euler forward method, so it easily accumulates numerical errors. This is the same issue of the stability of solutions and the possibility of "artifacts" that we discussed in Section 6.4, now arising from discretization of both space and time. In the example of the transport equation above, such issues can occur if you choose a value that is too large for Δt or Δh, relative to the spatial/temporal scale of the system's behavior (which is determined by w in this case). For example, if you increase Δt to 0.1, you will get the

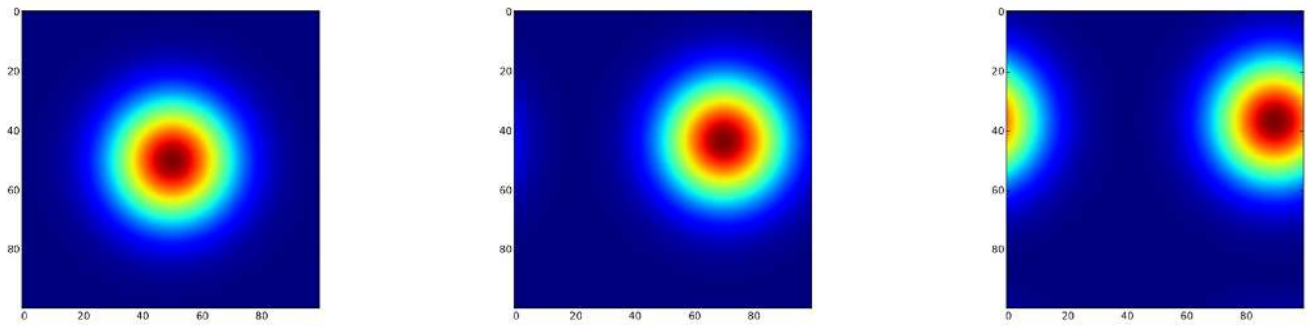


Figure 13.5.2: Visual output of Code 13.6. Time flows from left to right.

result shown in Fig. 13.5.3. This may look cool, but it is actually an invalid result. You can tell it is invalid if you understand the nature of the transport equation; it should only transport the surface and not change its shape. But even if you are unsure if the result you obtained is valid or not, you can try increasing the temporal/spatial resolutions (i.e., making Δt and Δh smaller) to see if the result varies. If the result remains unaffected by this, then you know that the original spatial/temporal resolutions were already sufficient and that the original result was thus valid. But if not, the original result probably contained numerical errors, so you need to keep increasing the resolutions until the result becomes consistent.

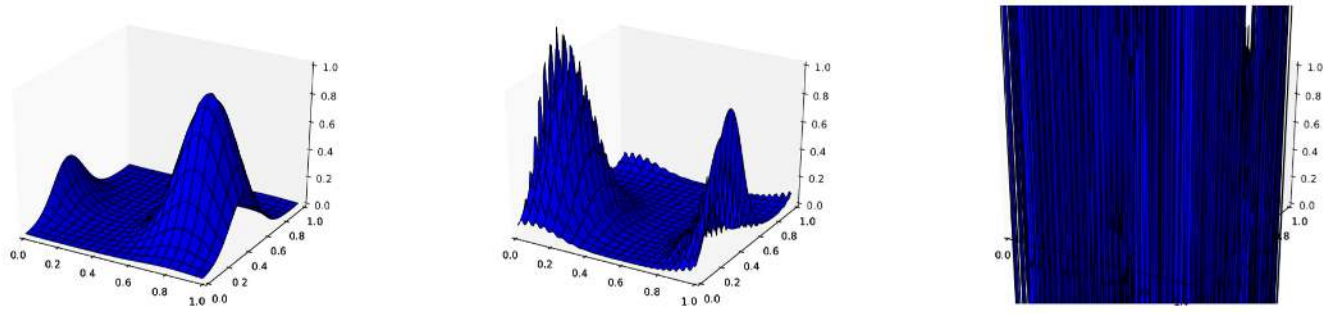


Figure 13.5.3: Accumulation of numerical errors caused by increasing $\Delta t (Dt)$ to 0.1 in Code 13.5. Time flows from left to right.

Before moving on to the next topic, I would like to point out that the discretization/simulation methods discussed above can be extended to PDEs that involve multiple state variables. Here is such an example: interactions between two spatially distributed populations, a escaping from diffusing b:

$$\frac{\partial a}{\partial t} = -\mu_a \nabla \cdot (a(-\nabla b)) \tag{13.5.16}$$

$$\frac{\partial b}{\partial t} = \mu_b \nabla^2 b \tag{13.5.17}$$

Here μ_a and μ_b are the parameters that determine the mobility of the two species. These equations can be expanded and then discretized for a 2-D space, as follows (you should also try doing this all by yourself!):

$$\frac{\partial a}{\partial t} = -\mu_a \left(\begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}^T \begin{pmatrix} -a\frac{\partial b}{\partial x} \\ -a\frac{\partial b}{\partial y} \end{pmatrix} \right. \tag{13.5.18}$$

$$= \mu_a \left(\frac{\partial}{\partial x}\left(a\frac{\partial b}{\partial x}\right) + \frac{\partial}{\partial y}\left(a\frac{\partial b}{\partial y}\right)\right) \tag{13.5.19}$$

$$= \mu_a \left(\frac{\partial a}{\partial x}\frac{\partial b}{\partial x} + a\frac{\partial^2 b}{\partial x^2} + \frac{\partial a}{\partial y}\frac{\partial b}{\partial y} + a\frac{\partial^2 b}{\partial y^2}\right. \tag{13.5.20}$$

$$= \mu_a \left(\frac{\partial a}{\partial x}\frac{\partial b}{\partial x} + \frac{\partial a}{\partial y}\frac{\partial b}{\partial y} a\nabla^2 b\right) \tag{13.5.21}$$

$$label(13.49)$$

$$\begin{gathered} a(x,y,t+\Delta t) \approx a(x,y,t) + \mu_a \\ \left(\frac{a(x+\Delta h,y,t) - a(x-\Delta h,y,t)}{2\,\Delta\, h} \frac{b(x+\Delta h,y,t) - b(x-\Delta h,y,t)}{2\,\Delta\, h} \right. \\ + \frac{a(x,y+\Delta h,t) - a(x,y-\Delta h,t)}{2\,\Delta\, h} \frac{b(x,y+\Delta h,t) - b(x,y-\Delta h,t)}{2\,\Delta\, h} + a(x,y,t) \\ \left. \times \frac{b(x+\Delta h,y,t) + b(x-\Delta h,y,t) + b(x,y+\Delta h,t) + b(x,y-\Delta h,t) - 4b(x,y,t)}{\Delta h^2} \right) \\ \Delta t \end{gathered} \tag{13.5.22}$$

$$\begin{gathered} a'c \approx a_c \\ + \mu_a \left(\frac{a_R - a_L}{2\,\Delta\, h} \frac{b_R - b_L}{2\,\Delta\, h} + \frac{a_V - a_D}{2\,\Delta\, h} \frac{b_V - b_D}{2\,\Delta\, h} + a_C \frac{b_R + b_L + b_U + b_D - 4b_C}{\Delta h^2} \right) \Delta\, t \end{gathered} \tag{13.5.23}$$

Note that I used simpler notations in the last equation, where subscripts (C, R, L, U, D) represent states of the central cell as well as its four neighbors, and $a'C$ is the next value of a_C. In the meantime, the discretized equation for b is simply given by

$$b'_C \approx b_C + \mu_b \frac{b_R + b_L + b_U + b_D - 4b_C}{\Delta h^2} \Delta t. \tag{13.5.24}$$

Below is a sample code for simulating this PDE model starting with an initial configuration with two peaks, one for a and the other for b:

Code 13.7: transport-ca-escaping.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of grid: n * n
Dh = 1. / n # spatial resolution, assuming space is [0,1] * [0,1]
Dt = 0.01 # temporal resolution

mu_a = 0.001 # mobility of species a
mu_b = 0.001 # mobility of species b

xvalues, yvalues = meshgrid(arange(0, 1, Dh), arange(0, 1, Dh))
```

```
def initialize():
    global a, b, nexta, nextb
    # initial configuration
    a = exp(-((xvalues - 0.45)**2 + (yvalues - 0.45)**2) / (0.3**2))
    b = exp(-((xvalues - 0.55)**2 + (yvalues - 0.55)**2) / (0.1**2))
    nexta = zeros([n, n])
    nextb = zeros([n, n])

def observe():
    global a, b, nexta, nextb
    subplot(1, 2, 1)
    cla()
    imshow(a, vmin = 0, vmax = 1)
    title('a')
```

```
    subplot(1, 2, 2)
    cla()
    imshow(b, vmin = 0, vmax = 1)
    title('b')

def update():
```

```
    global a, b, nexta, nextb
    for x in xrange(n):
        for y in xrange(n):
            # state-transition function
            aC, aR, aL, aU, aD = a[x,y], a[(x+1)%n,y], a[(x-1)%n,y], \
                                 a[x,(y+1)%n], a[x,(y-1)%n]
            bC, bR, bL, bU, bD = b[x,y], b[(x+1)%n,y], b[(x-1)%n,y], \
                                 b[x,(y+1)%n], b[x,(y-1)%n]
            bLapNum = bR + bL + bU + bD - 4 * bC
            nexta[x,y] = aC + mu_a * ((aR-aL)*(bR-bL)+(aU-aD)*(bU-bD)
                                      +4*aC*bLapNum) * Dt/(4*Dh**2)
            nextb[x,y] = bC + mu_b * bLapNum * Dt/(Dh**2)

    a, nexta = nexta, a
    b, nextb = nextb, b

import pycxsimulator
pycxsimulator.GUI(stepSize = 50).start(func=[initialize, observe, update])
```

Note that the `observe` function now uses `subplot` to show two scalar fields for a and b. The result is shown in Fig. 13.5.4.

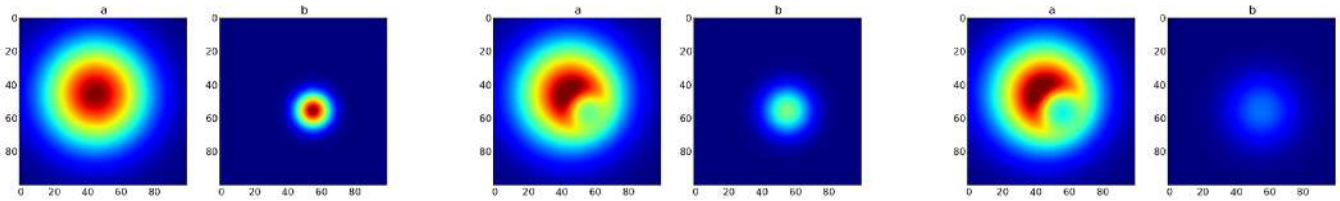


Figure **13.5.4**: Visual output of Code 13.7. Time flows from left to right.

? Exercise 13.5.1

Modify Code 13.7 so that both species diffuse and try to escape from each other. Run simulations to see how the system behaves.

[2]If you are not clear on what I am talking about here, see `http://top500.org/` .

This page titled 13.5: Simulation of Continuous Field Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

13.6: Reaction-Diffusion Systems

Finally, I would like to introduce *reaction-diffusion systems*, a particular class of continuous field models that have been studied extensively. They are continuous field models whose equations are made of only *reaction terms* and *diffusion terms*, as shown below:

$$\frac{\partial f_1}{\partial t} = R_1(f_1, f_2, \ldots, f_n) + D_1 \nabla^2 f_1 \tag{13.6.1}$$

$$\frac{\partial f_2}{\partial t} = R_2(f_1, f_2, \ldots, f_n) + D_2 \nabla^2 f_2 \tag{13.6.2}$$

$$\vdots$$

$$\frac{\partial f_n}{\partial t} = R_n(f_1, f_2, \ldots, f_n) + D_n \nabla^2 f_n \tag{13.6.3}$$

Reaction terms ($R_i(\ldots)$) describe only local dynamics, without any spatial derivatives involved. Diffusion terms ($D_i \nabla^2 f_i$) are strictly limited to the Laplacian of the state variable itself. Therefore, any equations that involve non-diffusive spatial movement (e.g., chemotaxis) are not reaction-diffusion systems.

There are several reasons why reaction-diffusion systems have been a popular choice among mathematical modelers of spatio-temporal phenomena. First, their clear separation between non-spatial and spatial dynamics makes the modeling and simulation tasks really easy. Second, limiting the spatial movement to only diffusion makes it quite straightforward to expand any existing non-spatial dynamical models into spatially distributed ones. Third, the particular structure of reaction-diffusion equations provides an easy shortcut in the stability analysis (to be discussed in the next chapter). And finally, despite the simplicity of their mathematical form, reaction-diffusion systems can show strikingly rich, complex spatio-temporal dynamics. Because of these properties, reaction diffusion systems have been used extensively for modeling self-organization of spatial patterns. There are even specialized software applications available exactly to simulate reaction-diffusion systems[3].

? Exercise 13.6.1

Extend the following non-spatial models into spatially distributed ones as reaction-diffusion systems by adding diffusion terms. Then simulate their behaviors in Python.

- Motion of a pendulum (Eq.(6.2.1)): This creates a spatial model of locally coupled nonlinear oscillators.
- Susceptible-Infected-Recovered (SIR) model (Exercise 7.1.3): This creates a spatial model of epidemiological dynamics.

In what follows, we will review a few well-known reaction-diffusion systems to get a glimpse of the rich, diverse world of their dynamics.

Turing Pattern Formation

As mentioned at the very beginning of this chapter, Alan Turing's PDE models were among the first reaction-diffusion systems developed in the early 1950s [44]. A simple linear version of Turing's equations is as follows:

$$\frac{\partial u}{\partial t} = a(u - h) + b(v - k) + D_u \Delta^2 u \tag{13.6.4}$$

$$\frac{\partial u}{\partial t} = c(u - h) + d(v - k) + D_u \Delta^2 u \tag{13.6.5}$$

The state variables u and v represent concentrations of two chemical species. a, b, c, and d are parameters that determine the behavior of the reaction terms, while h and k are constants. Finally, D_u and D_v are diffusion constants.

If the diffusion terms are ignored, it is easy to show that this system has only one equilibrium point, $(u_eq, v_eq) = (h, k)$. This equilibrium point can be stable for many parameter values for a, b, c, and d. What was most surprising in Turing's findings is that, even for such stable equilibrium points, introducing spatial dimensions and diffusion terms to the equations may destabilize the

equilibrium, and thus the system may spontaneously self-organize into a non-homogeneous pattern. This is called *diffusion-induced instability* or *Turing instability*. A sample simulation result is shown in Fig. 13.6.1.

The idea of diffusion-induced instability is quite counter-intuitive. Diffusion is usually considered a random force that destroys any structure into a homogenized mess, yet in this particular model, diffusion is the key to self-organization! What is going on? The trick is that this system has two different diffusion coefficients, D_u and D_v, and their difference plays a key role in determining the stability of the system's state. This will be discussed in more detail in the next chapter.

There is one thing that needs particular attention when you are about to simulate Turing's reaction-diffusion equations. The Turing pattern formation requires small random perturbations (noise) to be present in the initial configuration of the system; otherwise there would be no way for the dynamics to break spatial symmetry to create non-homogeneous patterns. In the meantime, such initial perturbations should be small enough so that they won't immediately cause numerical instabilities in the simulation. Here is a sample code for simulating Turing pattern formation with a suggested level of initial perturbations, using the parameter settings shown in Fig. 13.6.1:



Figure **13.6.1**: Simulation of the Turing pattern formation model with $(a, b, c, d) = (1, -1, 2, -1.5)$ and $(D_u, D_v) = (10^{-}4, 6 \times 10^{-}4)$. Densities of u are plotted in grayscale (darker = greater). Time flows from left to right.

Code 13.8: turing-pattern-PDE.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of grid: n * n
Dh = 1. / n # spatial resolution, assuming space is [0,1] * [0,1]
Dt = 0.02 # temporal resolution

a, b, c, d, h, k = 1., -1., 2., -1.5, 1., 1. # parameter values

Du = 0.0001 # diffusion constant of u
Dv = 0.0006 # diffusion constant of v
```

```
def initialize():
    global u, v, nextu, nextv
    u = zeros([n, n])
    v = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
            u[x, y] = 1. + uniform(-0.03, 0.03) # small noise is added
            v[x, y] = 1. + uniform(-0.03, 0.03) # small noise is added
    nextu = zeros([n, n])
```

```
    nextv = zeros([n, n])

def observe():
    global u, v, nextu, nextv
    subplot(1, 2, 1)
    cla()
    imshow(u, vmin = 0, vmax = 2, cmap = cm.binary)
    title('u')
    subplot(1, 2, 2)
    cla()
    imshow(v, vmin = 0, vmax = 2, cmap = cm.binary)
    title('v')
```

```
def update():
    global u, v, nextu, nextv
    for x in xrange(n):
        for y in xrange(n):
            # state-transition function
            uC, uR, uL, uU, uD = u[x,y], u[(x+1)%n,y], u[(x-1)%n,y], \
                                 u[x,(y+1)%n], u[x,(y-1)%n]
            vC, vR, vL, vU, vD = v[x,y], v[(x+1)%n,y], v[(x-1)%n,y], \
                                 v[x,(y+1)%n], v[x,(y-1)%n]
            uLap = (uR + uL + uU + uD - 4 * uC) / (Dh**2)
            vLap = (vR + vL + vU + vD - 4 * vC) / (Dh**2)
            nextu[x,y] = uC + (a*(uC-h) + b*(vC-k) + Du * uLap) * Dt
            nextv[x,y] = vC + (c*(uC-h) + d*(vC-k) + Dv * vLap) * Dt
```

```
    u, nextu = nextu, u
    v, nextv = nextv, v

import pycxsimulator
pycxsimulator.GUI(stepSize = 50).start(func=[initialize, observe, update])
```

This simulation starts from an initial configuration $(u(x,y), v(x,y)) \approx (1,1) = (h,k)$, which is the system's homogeneous equilibrium state that would be stable without diffusion terms. Run the simulation to see how patterns spontaneously self-organize!

? Exercise 13.6.2

Conduct simulations of the Turing pattern formation with several different parameter settings, and discuss how the parameter variations (especially for the diffusion constants) affect the resulting dynamics.

? Exercise 13.6.3

Discretize the Keller-Segel slime mold aggregation model (Eqs. (13.4.13) and (13.4.14)) (although this model is not a reaction-diffusion system, this is the perfect time for you to work on this exercise because you can utilize Code 13.8). Implement its simulation code in Python, and conduct simulations with $\mu = 10^{-4}$, $D = 10^{-4}$, $f = 1$, and $k = 1$, while varying χ as a control parameter ranging from 0 to 10^{-3}. Use $a = 1$ and $c = 0$ as initial conditions everywhere, with small random perturbations added to them.

Belousov-Zhabotinsky Reaction

The *Belousov-Zhabotinsky reaction*, or *BZ reaction* for short, is a family of oscillatory chemical reactions first discovered by Russian chemist Boris Belousov in the 1950s and then later analyzed by Russian-American chemist Anatol Zhabotinsky in the 1960s. One of the common variations of this reaction is essentially an oxidation of malonic acid ($CH_2(COOH)_2$) by an acidified bromate solution, yet this process shows nonlinear oscillatory behaviorf or a substantial length of time before eventually reaching chemical equilibrium. The actual chemical mechanism is quite complex, involving about 30 different chemicals. Moreover, if this chemical solution is put into a shallow petri dish, the chemical oscillation starts in different phases at different locations. Interplay between the reaction and the diffusion of the chemicals over the space will result in the self-organization of dynamic traveling waves (Figure 13.6.2), just like those seen in the excitable media CA model in Section 11.5.

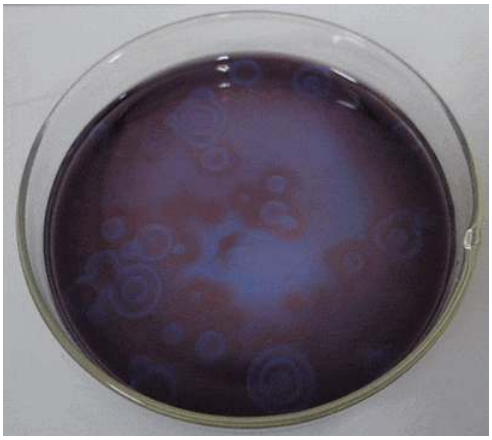


Figure 13.6.2: Belousov-Zhabotinsky reaction taking place in a petri dish. (Chemlignan). https://chemlegin.wordpress.com/2016/04/

A simplified mathematical model called the *"Oregonator"* was among the first to describe the dynamics of the BZ reaction in a simple form [50]. It was originally proposed as a non-spatial model with three state variables, but the model was later simplified to have just two variables and then extended to spatial domains [51]. Here are the simplified "Oregonator" equations:

$$\epsilon \frac{\partial u}{\partial t} = u(1-u) - \frac{u-q}{u+q} f v + D_u \Delta^2 u \tag{13.6.6}$$

$$\frac{\partial v}{\partial t} = u - v + D_u \Delta^2 v \tag{13.6.7}$$

Here, u and v represent the concentrations of two chemical species. If you carefully examine the reaction terms of these equations, you will noticethat the presenceof chemical u has a positive effect on both u and v, while the presence of chemical v has a negative effect on both. Therefore, these chemicals are called the "activator" and the "inhibitor," respectively. Similar interactions between the activator and the inhibitor were also seen in the Turing pattern formation, but the BZ reaction system shows nonlinear chemical oscillation. This causes the formation of traveling waves. Sometimes those waves can form spirals if spatial symmetry is broken by stochastic factors. A sample simulation result is shown in Fig. 13.6.3.

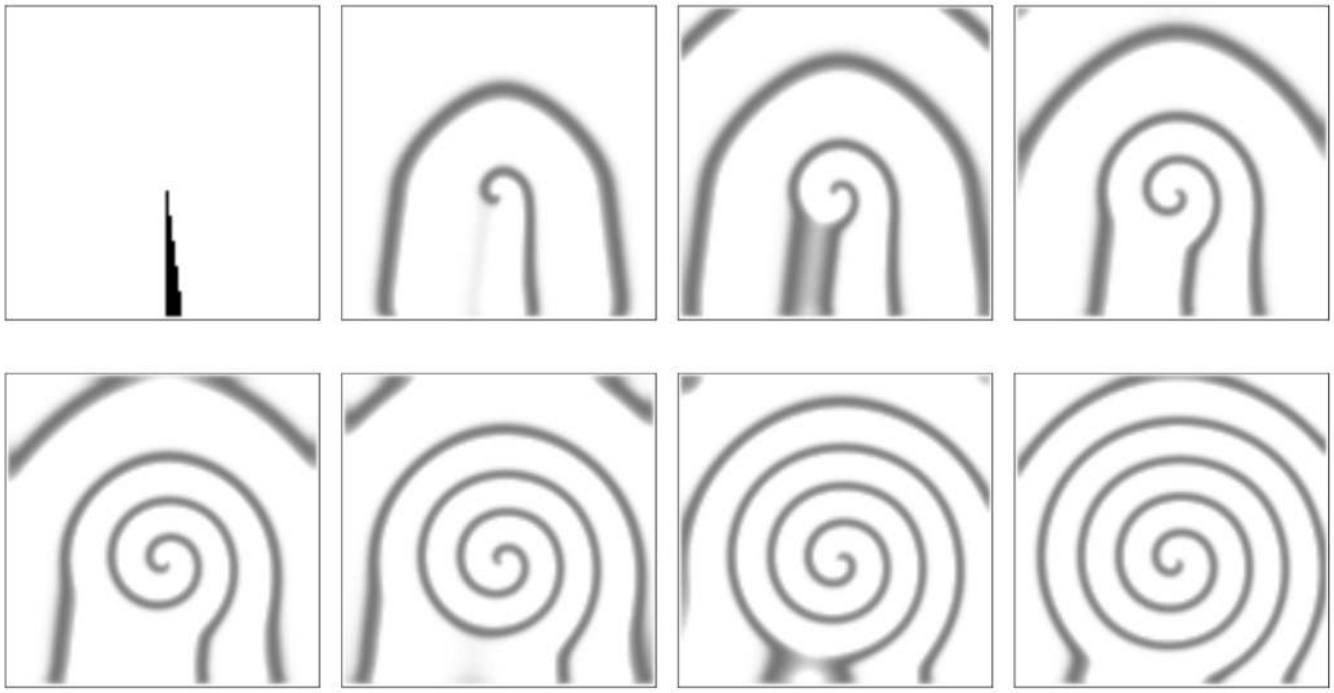


Figure 13.6.3: Simulation of the "Oregonator" model of the BZ reaction with $(\epsilon, q, f) = (0.2, 10^{-3}, 1.0)$ and $D_u = D_v = 10^{-5}$, starting with a tongue-like initial configuration. The concentration of chemical u is plotted in grayscale (darker = greater). Time flows from left to right (the first row followed by the second row).

? Exercise 13.6.5

Implement a simulator code of the "Oregonator" model of the BZ reaction in Python. Then conduct simulations with several different parameter settings, and discuss what kind of conditions would be needed to produce traveling waves.

Gray-Scott Pattern Formation

The final example is the *Gray-Scott model*, another very well-known reaction-diffusion system studied and popularized by John Pearson in the 1990s [52], based on a chemical reaction model developed by Peter Gray and Steve Scott in the 1980s [53, 54, 55]. The model equations are as follows:

$$\frac{\partial u}{\partial t} = F(1-u) - uv^2 + D_u \Delta^2 u \tag{13.6.8}$$

$$\frac{\partial v}{\partial t} = -(F+k)v + uv^2 + D_u \Delta^2 v \tag{13.6.9}$$

The reaction terms of this model assumes the following *autocatalytic reaction* (i.e., chemical reaction for which the reactant itself serves as a catalyst):

$$u + 2v \to 3v \tag{13.6.10}$$

This reaction takes one molecule of u and turns it into one molecule of v, with help of two other molecules of v (hence, autocatalysis). This is represented by the second term in each equation. In the meantime, u is continuously replenished from the external source up to 1 (the first term of the first equation) at feed rate F, while v is continuously removed from the system at a rate slightly faster than u 's replenishment ($F + k$seen in the first term of the second equation). F and k are the key parameters of this model.

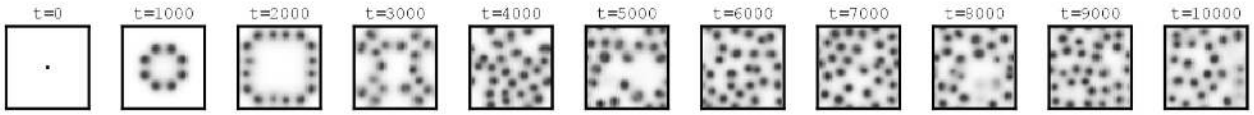
It is easy to show that, if the diffusion terms are ignored, this system always has an equilibrium point at $(u_{eq}, v_{eq}) = (1, 0)$ (which is stable for any positive F and k). Surprisingly, however, this model may show very exotic, biological-looking dynamics if certain spatial patterns are placed into the above equilibrium. Its behaviors are astonishingly rich, even including growth, division, and death of "cells" if the parameter values and initial conditions are appropriately chosen. See Fig. 13.6.4 to see only a few samples of its wondrous dynamics!

? Exercise 13.6.5

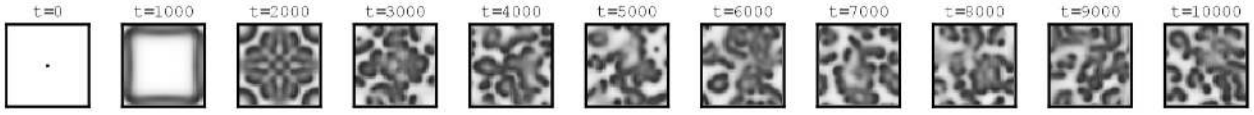
Implement a simulator code of the Gray-Scott model in Python. Then conduct simulations with several different parameter settings and discuss how the parameters affect the resulting patterns.

***Figure* 13.6.4**: *(Next page) Samples of patterns generated by the Gray-Scott model with* $D_u = 2 \times 10^{-5}$ *and* $D_v = 10^{-5}$. *The concentration of chemical* u *is plotted in grayscale (brighter = greater, only in this figure). Time flows from left to right. The parameter values of* F *and* k *are shown above each simulation result. The initial conditions are the homogeneous equilibrium* $(u, v) = (1, 0)$ *everywhere in the space, except at the center where the local state is reversed such that* $(u, v) = (0, 1)$.

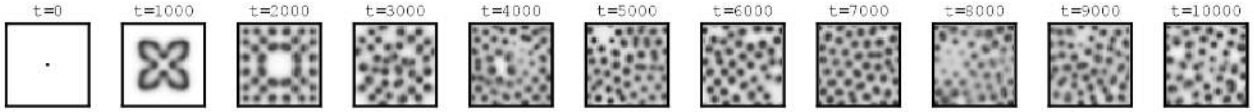
F=0.015, k=0.055



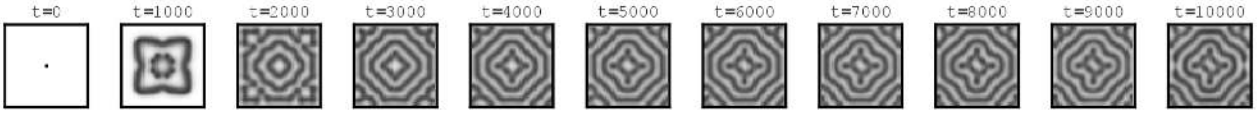
F=0.02, k=0.05



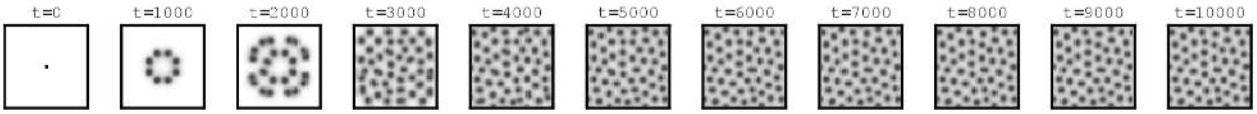
F=0.02, k=0.055



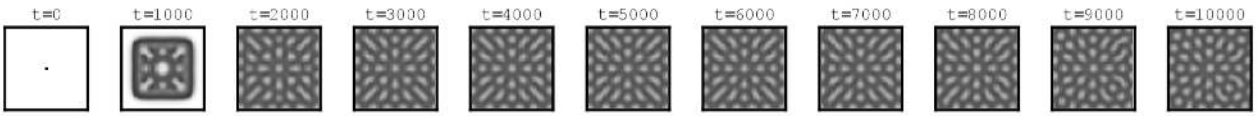
F=0.025, k=0.055



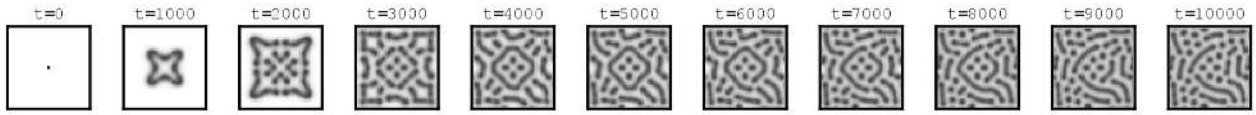
F=0.025, k=0.06



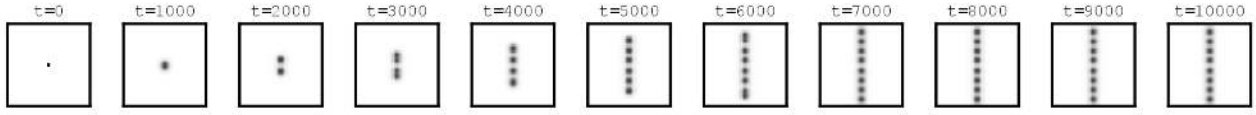
F=0.03, k=0.055



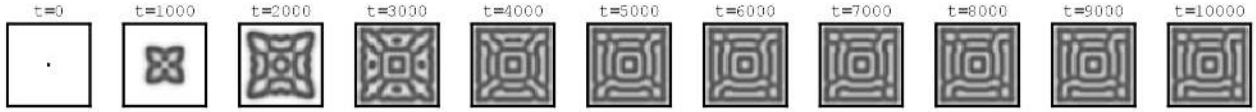
F=0.03, k=0.06



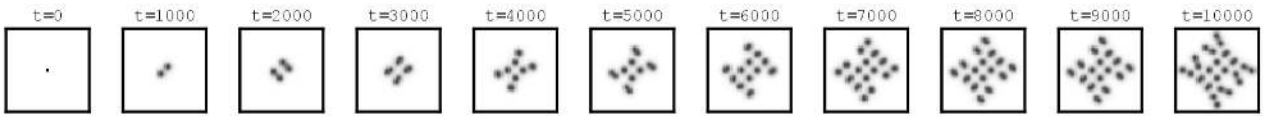
F=0.035, k=0.065



F=0.04, k=0.06



F=0.04, k=0.065



[3]For example, check out Ready (https://code.google.com/p/reaction-diffusion/).

This page titled 13.6: Reaction-Diffusion Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

14: Continuous Field Models II - Analysis

14.1: Finding Equilibrium States

14.2: Variable Rescaling

14.3: Linear Stability Analysis of Continuous Field Models

14.4: Linear Stability Analysis of Reaction-Diffusion Systems

Index

This page titled 14: Continuous Field Models II - Analysis is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

14.1: Finding Equilibrium States

One nice thing about PDE-based continuous field models is that, unlike CA models, everything is still written in smooth differential equations so we may be able to conduct systematic mathematical analysis to investigate their dynamics (especially their stability or instability) using the same techniques as those we learned for non-spatial dynamical systems in Chapter 7.

The first step is, as always, to find the system's equilibrium states. Note that this is no longer about equilibrium "points," because the system's state now has spatial extensions. In this case, the equilibrium state of an autonomous continuous field model

$$\frac{\partial f}{\partial t} = F(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2} ...) \tag{14.1.1}$$

is given as a static spatial function $f_{eq}(x)$, which satisfies

$$0 = F(f_{eq}, \frac{\partial f_{eq}}{\partial x}, \frac{\partial^2 f}{\partial x^2} ...). \tag{14.1.2}$$

A Simple Sinusoidal Source/Sink Model

For example, let's obtain the equilibrium state of a diffusion equation in a 1-D space with a simple sinusoidal source/sink term:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + \sin x \qquad (-\pi \leq x \leq \pi) \tag{14.1.3}$$

The source/sink term $\sin x$ means that the "stuff" is being produced where $0 < x \leq \pi$, while it is being drained where $-\pi \leq x < 0$. Mathematically speaking, this is still a nonautonomous system because the independent variable x appears explicitly on the right hand side. But this non-autonomy can be easily eliminated by replacing x with a new state variable y that satisfies

$$\frac{\partial y}{\partial t} = 0, \tag{14.1.4}$$

$$y(x, 0) = x. \tag{14.1.5}$$

In the following, we will continue to use x instead of y, just to make the discussion easier and more intuitive.

To find an equilibrium state of this system, we need to solve the following:

$$0 = D\nabla^2 c_{eq} + \sin x \tag{14.1.6}$$

$$= D\frac{d^2 c_{eq}}{dx^2} + \sin x \tag{14.1.7}$$

This is a simple ordinary differential equation, because there are no time or additional spatial dimensions in it. You can easily solve it by hand to obtain the solution

$$c_{eq}(x) = \frac{\sin x}{D} + C_1 x + C_2 \tag{14.1.8}$$

where C_1 and C_2 are the constants of integration. Any state that satisfies this formula remains unchanged over time. Figure 14.1.1 shows such an example with $D = C_1 = C_2 = 1$.

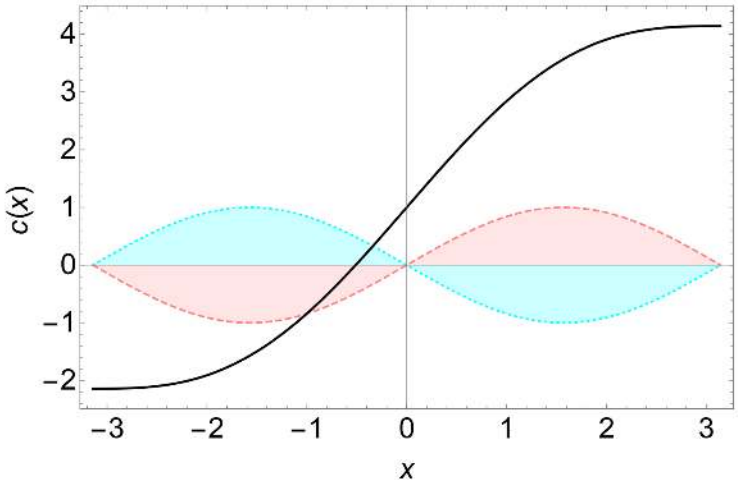


Figure 14.1.1: *Example of equilibrium states of the PDE model Equation 14.1.3 with $D = C_1 = C_2 = 1$. The black solid curve shows $c_{eq}(x)$, while the cyan (dotted) and pink (dashed) curves represent the increase/decrease caused by the diffusion term and the source/sink term, respectively. Those two terms are balanced perfectly in this equilibrium state.*

? Exercise 14.1.1

Obtain the equilibrium states of the following continuous field model in a 1-D space:

$$\partial c/\partial t = D\nabla^2 c + 1 - x^2 \tag{14.1.9}$$

As we see above, equilibrium states of a continuous field model can be spatially heterogeneous. But it is often the case that researchers are more interested in *homogeneous equilibrium states*, i.e., spatially "flat" states that can remain stationary over time. This is because, by studying the stability of homogeneous equilibrium states, one may be able to understand whether a spatially distributed system can remain homogeneous or selforganize to form non-homogeneous patterns spontaneously.

Calculating homogeneous equilibrium states is much easier than calculating general equilibrium states. You just need to substitute the system state functions with constants, which will make all the derivatives (both temporal and spatial ones) become zero. For example, consider obtaining homogeneous equilibrium states of the following Turing pattern formation model:

$$\frac{\partial u}{\partial t} = a(u-h) + b(v-k) + D_u\nabla^2 u \tag{14.1.10}$$

$$\frac{\partial v}{\partial t} = c(u-h) + d(v-k) + D_v\nabla^2 v \tag{14.1.11}$$

The only thing you need to do is to replace the spatio-temporal functions $u(x,t)$ and $v(x,t)$ with the constants u_eq and v_eq, respectively:

$$\frac{\partial u_{eq}}{\partial t} = a(u_{eq}-h) + b(v_{eq}-k) + D_u\nabla^2 u_{eq} \tag{14.1.12}$$

$$\frac{\partial v_{eq}}{\partial t} = c(u_{eq}-h) + d(v_{eq}-k) + D_v\nabla^2 v_{eq} \tag{14.1.13}$$

Note that,since u_{eq} and v_{eq} no longer depend on either time or space, the temporal derivatives on the left hand side and the Laplacians on the right hand side both go away. Then we obtain the following:

$$0 = a(u_{eq}-h) + b(v_{eq}-k) \tag{14.1.14}$$

$$0 = c(u_{eq}-h) + d(v_{eq}-k) \tag{14.1.15}$$

By solving these equations, we get $(u_{eq}, v_{eq}) = (h, k)$, as we expected.

Note that we can now represent this equilibrium state as a "point" in a two-dimensional (u, v) vector space. This is another reason why homogeneous equilibrium states are worth considering; they provide a simpler, low-dimensional reference point to help us

understand the dynamics of otherwise complex spatial phenomena. Therefore, we will also focus on the analysis of homogeneous equilibrium states for the remainder of this chapter.

? Exercise 14.1.2

Obtain homogeneous equilibrium states of the following "Oregonator" model:

$$\epsilon\frac{\partial u}{\partial t} = u(1-u) - \frac{u-q}{u+q}fv + D_u\nabla^2 u \tag{14.1.16}$$

$$\frac{\partial v}{\partial t} = u - v + D_v\nabla^2 v \tag{14.1.17}$$

? Exercise 14.1.3

Obtain homogeneous equilibrium states of the following Keller-Segel model:

$$\frac{\partial a}{\partial t} = \mu\nabla^2 a - \chi\nabla\cdot(a\nabla c) \tag{14.1.18}$$

$$\frac{\partial c}{\partial t} = D\nabla^2 c + fa - kc \tag{14.1.19}$$

This page titled 14.1: Finding Equilibrium States is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

14.2: Variable Rescaling

Variable rescaling of continuous field models comes with yet another bonus variable, i.e., space, which you can rescale to potentially eliminate more parameters from the model. In a 2-D or higher dimensional space, you can, theoretically, have two or more spatial variables to rescale independently. But the space is usually assumed to be isotropic (i.e., there is no difference among the directions of space) in most spatial models, so it may not be practically meaningful to use different rescaling factors for different spatial variables.

Anyway, here is an example. Let's try to simplify the following spatial predator-prey model, formulated as a reaction-diffusion system in a two-dimensional space:

$$\begin{aligned}\frac{\partial r}{\partial t} &= ar - brf + D_r \nabla^2 r \\ &= ar - brf + D_r \left(\frac{\partial^2 r}{\partial x^2} + \frac{\partial^2 r}{\partial y^2} \right) \end{aligned} \tag{14.2.1}$$

$$\begin{aligned}\frac{\partial f}{\partial t} &= -cf + drf + D_f \nabla^2 f \\ &= -cf + drf + D_f \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) \end{aligned} \tag{14.2.2}$$

Here we use r for prey (rabbits) and f for predators (foxes), since x and y are already taken as the spatial coordinates. We can apply the following three rescaling rules to state variables r and f, as well as time t and space x/y:

Multiple \label

With these replacements, the model equations (Equation 14.2.1 and 14.2.2) can be rewritten as follows:

$$\frac{\partial(\alpha r')}{\partial \gamma t'} = a(\alpha r') - b(\alpha r')(\beta f') + D_r \left(\frac{\partial^2(\alpha r')}{\partial(\delta x')^2} + \frac{\partial^2(\alpha r')}{\partial(\delta y')^2} \right) \tag{14.2.3}$$

$$\frac{\partial(\beta f')}{\partial(\gamma t')} = -c(\beta f') + d(\alpha r')(\beta f') + D_f \left(\frac{\partial^2(\beta f')}{\partial(\delta x')^2} + \frac{\partial^2(\alpha r')}{\partial(\delta y')}^2 \right) \tag{14.2.4}$$

We can collect parameters and rescaling factors together, as follows:

$$\frac{\partial r'}{\partial t'} = a\gamma r' - b\beta\gamma r' f' + \frac{D_r \gamma}{\delta^2} \left(\frac{\partial^2 r'}{\partial x'^2} + \frac{\partial^2 r'}{\partial y'^2} \right) \tag{14.2.5}$$

$$\frac{\partial f'}{\partial t'} = -c\gamma f' + d\alpha\gamma r' f' + \frac{D_f \gamma}{\delta^2} \left(\frac{\partial^2 f'}{\partial x'^2} + \frac{\partial^2 f'}{\partial y'^2} \right) \tag{14.2.6}$$

Then we can apply, e.g., the following rescaling choices

$$(\alpha, \beta, \gamma, \delta) = \left(\frac{a}{d}, \frac{a}{b}, \frac{1}{a}, \sqrt{\frac{D_r}{a}} \right) \tag{14.2.7}$$

to simplify the model equations into

$$\frac{\partial r'}{\partial t'} = r' - r' f' + \nabla^2 r', \tag{14.2.8}$$

$$\frac{\partial f'}{\partial t'} = -e\gamma f' + r' f' + D_{ratio} \nabla^2 f' \tag{14.2.9}$$

with $e = c/a$ and $D_{ratio} = D_f / D_r$. The original model had six parameters (a, b, c, d, D_r, and D_f), but we were able to reduce them into just two parameters (e and D_{ratio}) with a little additional help from spatial rescaling factor δ. This rescaling result also tells us some important information about what matters in this system: It is the ratio between the growth rate of the prey (a) and the decay rate of the predators (c) (i.e., $e = c/a$), as well as the ratio between their diffusion speeds ($D_{ratio} = D_f / D_r$), which essentially determines the dynamics of this system. Both of these new parameters make a lot of sense from an ecological viewpoint too.

Exercise 14.2.1: Rescaling the Keller-Segel model

Simplify the following Keller-Segel model by variable rescaling:

Multiple \label

This page titled 14.2: Variable Rescaling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

14.3: Linear Stability Analysis of Continuous Field Models

We can apply the *linear stability analysis* to continuous field models. This allows us to analytically obtain the conditions for which a homogeneous equilibrium state of a spatial system loses its stability and thereby the system spontaneously forms non-homogeneous spatial patterns. Note again that the homogeneous equilibrium state discussed here is no longer a single point, but it is a straight line (or a flat plane) that covers the entire spatial domain.

Consider the dynamics of a nonlinear continuous field model

$$\frac{\partial f}{\partial t} = F\left(f, \frac{\partial f}{\partial x}, \frac{\partial^2 f}{\partial x^2}, ...\right) \tag{14.3.1}$$

around its homogeneous equilibrium state f_{eq}, which satisfies

$$0 = F(f_{eq}, 0, ...). \tag{14.3.2}$$

The basic approach of linear stability analysis is exactly the same as before. Namely, we will represent the system's state as a sum of the equilibrium state and a small perturbation, and then we will determine whether this small perturbation added to the equilibrium will grow or shrink over time. Using Δf to represent the small perturbation, we apply the following replacement

$$f(x,t) \Rightarrow f_{eq} + \nabla f(x,t) \tag{14.3.3}$$

to Equation 14.3.1, to obtain the following new continuous field model:

$$\frac{\partial f_{eq} + \nabla f}{\partial t} = F(F_{eq} + \nabla f, \frac{\partial (f_{eq}) + \nabla f}{\partial x}, \frac{\partial^2 (f_{eq} + \nabla f)}{\partial x^2}, \cdots) \tag{14.3.4}$$

$$\frac{\partial \nabla f}{\partial t} = F(f_{eq} + \nabla f, \frac{\partial \nabla f}{\partial x}, \frac{\partial^2 \nabla f}{\partial x^2}, ...) \tag{14.3.5}$$

Now, look at the equation above. The key difference between this equation and the previous examples of the non-spatial models (e.g.,Equation (7.5.4)) is that the right hand side of Equation 14.3.5 contains spatial derivatives. Without them, F would be just a nonlinear scalar or vector function of f, so we could use its Jacobian matrix to obtain a linear approximation of it. But we can't do so because of those ∂'s! We need something different to eliminate those nuisances.

In fact, we saw a similar situation before. When we discussed how to obtain analytical solutions for linear dynamical systems, the nuisances were the matrices that existed in the equations. We "destroyed" those matrices by using their eigenvectors, i.e., vectors that can turn the matrix into a scalar eigenvalue when applied to it. Can we do something similar to destroy those annoying spatial derivatives?

The answer is, *yes we can*. While the obstacle we want to remove is no longer a simple matrix but a linear differential operator, we can still use the same approach. Instead of using eigenvectors, we will use so-called *eigenfunctions*. An eigenfunction of a *linear operator* L is a function that satisfies

$$Lf = \lambda f, \tag{14.3.6}$$

where λ is (again) called an *eigenvalue* that corresponds to the eigenfunction f. Look at the similarity between the definition above and the definition of eigenvectors (Equation 5.6.5)!

This similarity is no surprise, because, after all, the linear operators and eigenfunctions are straightforward generalizations of matrices and eigenvectors. They can be obtained by increasing the dimensions of matrices/vectors (= number of rows/columns) to infinity. Figure 14.3.1 gives a visual illustration of how these mathematical concepts are related to each other, where the second-order spatial derivative of a spatial function $f(x)$ in a $[0, 1]$ 1-D space is shown as an example.

When the space is discretized into n compartments, the function $f(x)$ is also defined on a discrete spatial grid made of n cells, and the calculation of its second-order spatial derivative (or, to be more precise, its discrete equivalent) can be achieved by adding its two nearest neighbors' values and then subtracting the cell's own value twice, as shown in Equation (13.5.11). This operation can be represented as a matrix with three diagonal lines of non-zero elements (Figure 14.3.1, left and center). Because it is a square matrix, we can calculate its eigenvalues and eigenvectors. And if the number of compartments n goes to infinity, we eventually move into the realm of continuous field models, where what used to be a matrix is now represented by a linear operator ($\partial 2/\partial x 2$,

i.e., ∇^2 in 1-D space), while the eigenvector is now made of infinitely many numbers, which entitles it to a new name, "eigenfunction."

In fact, this is just one instance of how mathematical concepts for matrices and vectors can be generalized to linear operators and functions in a continuous domain. Nearly all the concepts developed in linear algebra can be seamlessly extended continuous linear operators and functions, but we won't go into details about this in this textbook.

Most linear operators we see in PDE-based continuous field models are the secondorder (and sometimes first-order) differential operators. So here are their eigenfunctions (which are nothing more than general solutions of simple linear differential equations):

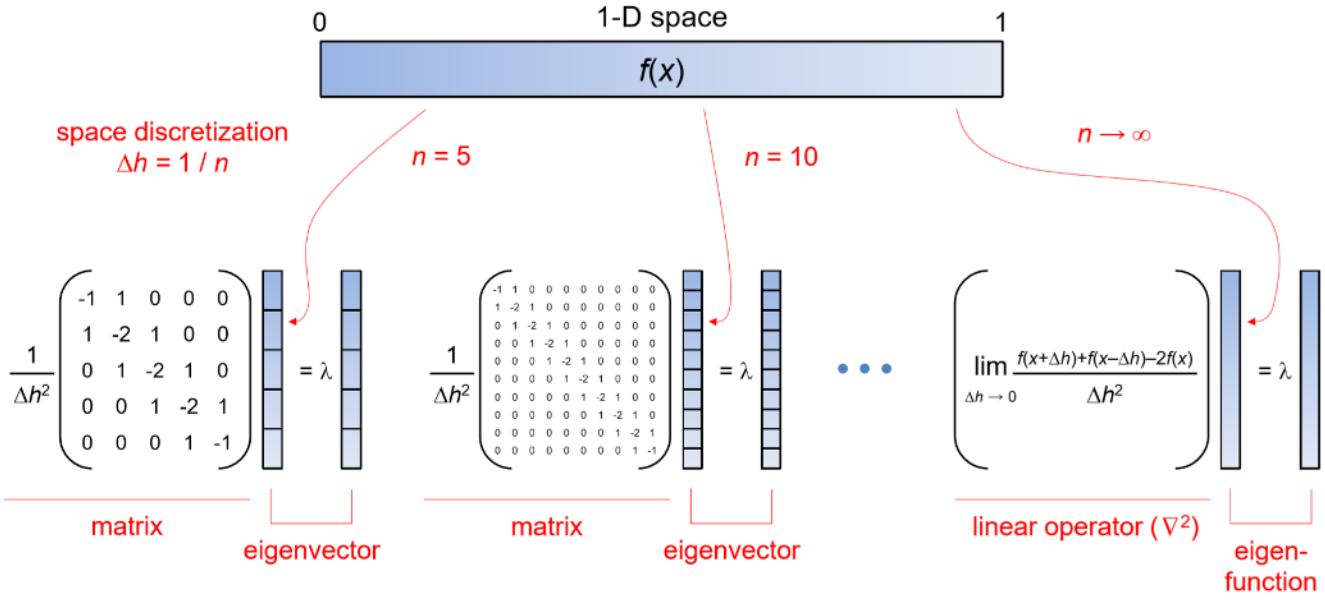


Figure **14.3.1**: Relationship between matrices/eigenvectors and linear operators/eigenfunctions.

- For $L = \dfrac{\partial}{\partial x}$:

$$f(x) = Ce^{\lambda x} \tag{14.3.7}$$

- For $L = \dfrac{\partial^2}{\partial x^2}$:

$$f(x) = C_1 e^{\sqrt{\lambda} x} + C_2 e^{-\sqrt{\lambda} x} \tag{14.3.8}$$

Here, λ is the eigenvalue and C, C_1, and C_2 are the constants of integration. You can confirm that these eigenfunctions satisfy Equation14.3.6 by applying L to it. So, if we use such an eigenfunction of the spatial derivative remaining in Equation 14.3.5 as a small perturbation Δf, the equation could become very simple and amenable to mathematical analysis.

There is one potential problem, though. The eigenfunctions given above are all exponential functions of x. This means that, for x with large magnitudes, these eigenfunctions could explode literally exponentially! This is definitely not a good property for a perturbation that is supposed to be "small." What should we do? There are at least two solutions for this problem. One way is to limit the scope of the analysis to a finite domain of x (and λ, too) so that the eigenfunction remains finite without explosion. The other way is to further investigate those eigenfunctions to see if they can take a form that doesn't show exponential divergence. Is it possible?

For Equation 14.3.7, a purely imaginary λ could make $f(x)$ non-divergent for $x \to \pm\infty$, but then $f(x)$ itself would also show complex values. This wouldn't be suitable as a candidate of perturbations to be added to real-valued system states. But for Equation 14.3.8, there are such eigenfunctions that don't explode exponentially and yet remain real. Try $\lambda < 0$ (i.e, $\sqrt{\lambda} = ai$) with complex conjugates C_1 and C_2 (i.e., $C_1 = c + bi$, $C_2 = c - bi$), and you will obtain

$$f(x) = (c+bi)e^{iax} + (c-bi)e^{-iax} \tag{14.3.9}$$
$$= c(e^{iax} + e^{-iax}) + bi(e^{iax} - e^{iax}) \tag{14.3.10}$$
$$= c(\cos ax + i\sin ax + \cos(-ax) + i\sin(-ax)) + bi(\cos ax + i\sin ax - \cos(-ax) - i\sin(-ax)) \tag{14.3.11}$$
$$= 2c\cos ax - 2b\sin ax \tag{14.3.12}$$
$$= A(\sin\phi)\cos ax - \cos\phi\sin ax \tag{14.3.13}$$
$$= A\sin(\phi - ax), \tag{14.3.14}$$

where $\varphi = \arctan(c/b)$ and $A = 2c/\sin\varphi = 2b/\cos\varphi$. This is just a normal, real-valued sine wave that will remain within the range $[-A, A]$ for any x! We can definitely use such sine wave-shaped perturbations for Δf to eliminate the second-order spatial derivatives.

Now that we have a basic set of tools for our analysis, we should do the same trick as we did before: Represent the initial condition of the system as a linear combination of eigen-(vector or function), and then study the dynamics of each eigen-(vector or function) component separately. The sine waves derived above are particularly suitable for this purpose, because, as some of you might know, the waves with different frequencies (a in the above) are independent from each other, so they constitute a perfect set of bases to represent any initial condition.

Let's have a walk-through of a particular example to see how the whole process of linear stability analysis works on a continuous field model. Consider our favorite Keller-Segel model:

$$\frac{\partial a}{\partial t} = \mu\nabla^2 a - \chi\nabla\cdot(a\nabla c) \tag{14.3.15}$$
$$\frac{\partial c}{\partial t} = D\nabla^2 c + fa - kc \tag{14.3.16}$$

The first thing we need to do is to find the model's homogeneous equilibrium state which we will study. As you may have done this in Exercise 14.3, any a_{eq} and c_{eq} that satisfy

$$fa_{eq} = kc_{eq} \tag{14.3.17}$$

can be a homogeneous equilibrium state of this model. Here we denote the equilibrium state as

$$(a_{eq}, c_{eq}) = (a_{eq}, \frac{f}{k}a_{eq}). \tag{14.3.18}$$

Then we introduce small perturbations into this homogeneous equilibrium state, as follows:

$$\begin{pmatrix} a(x,t) \\ c(x,t) \end{pmatrix} \Rightarrow \begin{pmatrix} a_{eq} \\ \frac{f}{k}a_{eq} \end{pmatrix} + \begin{pmatrix} \Delta a(x,t) \\ \Delta x(x,t) \end{pmatrix}. \tag{14.3.19}$$

Here, we assume that the space is just one-dimensional for simplicity (therefore no y's above). By applying these variable replacements to the Keller-Segel model, we obtain

$$\frac{\partial \Delta a}{\partial t} = \mu\frac{\partial^2}{\partial x^2}(a_{eq} + \Delta a) - \chi\frac{\partial}{\partial x}((a_{eq} + \Delta a) + \frac{\partial}{\partial x}(\frac{f}{x}a_{eq} + \Delta c)) \tag{14.3.20}$$
$$= \mu\frac{\partial^2 \Delta a}{\partial x^2} - \chi\frac{\partial}{\partial x}(a_{eq}\frac{\partial \Delta c}{\partial x} + \Delta a\frac{\partial \Delta c}{\partial x}) \tag{14.3.21}$$
$$= \mu\frac{\partial^2 \Delta a}{\partial x^2} - \chi a_{eq}\frac{\partial^2 \Delta c}{\partial x^2} - \chi\frac{\partial}{\partial x}(\Delta a\frac{\partial \Delta c}{\partial x}) \tag{14.3.22}$$
$$= \mu\frac{\partial^2 \Delta a}{\partial x^2} - \chi a_{eq}\frac{\partial^2 \Delta c}{\partial x^2} - \chi\frac{\partial \Delta a}{\partial x}\frac{\partial \Delta c}{\partial x} - \chi\Delta a\frac{\partial^2 \Delta c}{\partial x^2}, \tag{14.3.23}$$
$$\frac{\partial \Delta c}{\partial t} = D\frac{\partial^2}{\partial x^2}(\frac{f}{k}a_{eq} + \Delta c) + f(a_{eq} + \Delta a) - k(\frac{f}{k}a_{eq} + \Delta c) \tag{14.3.24}$$
$$= D\frac{\partial^2 \Delta c}{\partial x^2} + f\Delta a - k\Delta c. \tag{14.3.25}$$

In the equations above, both the second-order and first-order spatial derivatives remain. We can't find an eigenfunction that eliminates both simultaneously, so let's adopt sine waves, i.e., the eigenfunction for the second-order spatial derivatives that appear more often in the equations above, and see how the product of two first-order spatial derivatives in Equation 14.3.23 responds to it. Hence, we will assume

$$\begin{pmatrix}\Delta a(x,t)\\ \Delta c(x,t)\end{pmatrix} = \sin(\omega x) + \phi + \begin{pmatrix}\Delta a(t)\\ \Delta c(t)\end{pmatrix}, \tag{14.3.26}$$

where ω and φ are parameters that determine the spatial frequency and phase offset of the perturbation, respectively. $\omega/2\pi$ will give a spatial frequency (= how many waves there are per unit of length), which is called a wave number in physics. The phase offset φ doesn't really make any difference in this analysis, but we include it anyway for the sake of generality. Note that, by adopting a particular shape of the perturbation above, we have decoupled spatial structure and temporal dynamics in Equation 14.3.26 [1]. Now the only dynamical variables are $\Delta a(t)$ and $\Delta c(t)$, which are the amplitudes of the sine waveshaped perturbation added to the homogeneous equilibrium state.

By plugging Equation 14.3.26 into Eqs. 14.3.23 and 14.3.25, we obtain the following:

$$\begin{aligned}&\sin(\omega x+\phi)\frac{\partial \Delta a}{\partial t}\\ &= -\mu\omega^2\sin(\omega x+\phi)\Delta a + \chi a_{eq}\omega^2\sin(\omega x+\phi)\Delta x - \chi\omega^2\cos^2(\omega x+\phi)\Delta a\Delta c\\ &\quad + \chi\omega^2\sin(\omega x+\phi)\Delta a\Delta c\end{aligned} \tag{14.3.27}$$

$$sin(\omega x+\phi)\frac{\partial \Delta c}{\partial t} = -D\omega^2\sin(\omega x+\phi)\Delta c + f\sin(\omega x+\phi)\Delta a - k\sin(\omega x+\phi)\Delta c \tag{14.3.28}$$

Here, we see the product of two amplitudes ($\Delta a\ \Delta c$) in the last two terms of Equation 14.3.27, which is "infinitely smaller than infinitely small," so we can safely ignore them to linearize the equations. Note that one of them is actually the remnant of the product of the two first-order spatial derivatives which we had no clue as to how to deal with. We should be glad to see it exiting from the stage!

After ignoring those terms, every single term in the equations equally contains $\sin(\omega x+\varphi)$, so we can divide the entire equations by \(\sin{(ωx + φ)} to obtain the following linear ordinary differential equations:

$$\frac{d\Delta a}{\partial t} = -\mu\omega^2 + \chi a_{eq}\omega^2\Delta c \tag{14.3.29}$$

$$\frac{d\Delta c}{\partial t} = -D\omega^2\Delta c + f\Delta a - k\Delta c \tag{14.3.30}$$

Or, using a linear algebra notation:

$$\frac{d}{dt}\begin{pmatrix}\Delta a\\ \Delta c\end{pmatrix} = \begin{pmatrix}-\mu\omega^2 & \chi a_{eq}\omega^2\\ f & -D\omega^2-k\end{pmatrix}\begin{pmatrix}\Delta a\\ \Delta c\end{pmatrix} \tag{14.3.31}$$

We are finally able to convert the spatial dynamics of the original Keller-Segel model (only around its homogeneous equilibrium state) into a very simple, non-spatial linear dynamical system. What we have done is to constrain the shape of the small perturbations (i.e., deviations from the homogeneous equilibrium state) to a certain eigenfunction to eliminate spatial effects, and then ignore any higher-order terms to linearize the dynamics. Each point in this new $(\Delta a, \Delta c)$ phase space still represents a certain spatial configuration of the original model, as illustrated in Fig. 14.3.2. Studying the stability of the origin $(\Delta a, \Delta c) = (0,0)$ tells us if the Keller-Segel model can remain homogeneous or if it undergoes a spontaneous pattern formation.

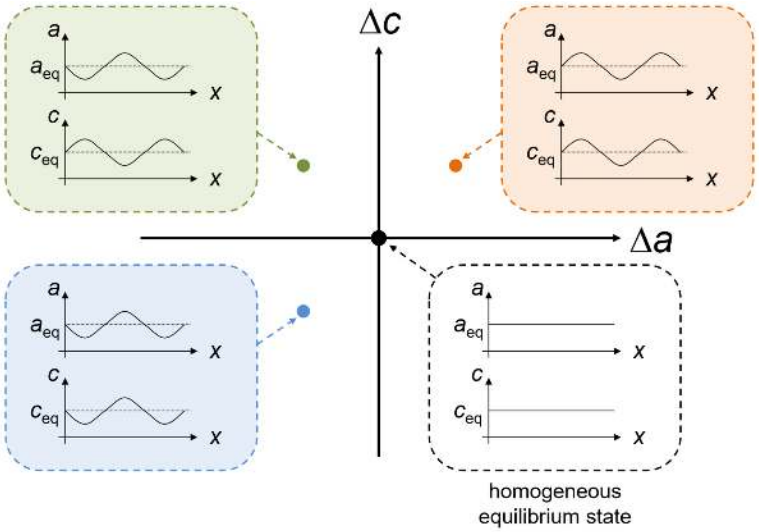


Figure **14.3.2**: Visual illustration of how to interpret each point in the $(\Delta a, \Delta c)$ phase space. Each call-out shows the actual state of the system that corresponds to each point in the phase space.

The only thing we need to do is to calculate the eigenvalues of the matrix in Equation14.3.31 and check the signs of their real parts. This may be cumbersome, but let's get it done. Here is the calculation process, where λ is the eigenvalue of the matrix:

$$\begin{vmatrix} -\mu\omega^2 - \lambda & \chi a_{eq}\omega^2 \\ f & -D\omega^2 - k - \lambda \end{vmatrix} = 0 \tag{14.3.32}$$

$$(-\mu\omega^2 - \lambda)(-D\omega^2 - k - \lambda) - \chi a_{eq}\omega^2 f = 0 \tag{14.3.33}$$

$$\lambda^2 + (\mu\omega^2 + D\omega^2 + k)\lambda + \mu\omega^2(D\omega^2 + k) - \chi a_{eq}\omega^2 f = 0 \tag{14.3.34}$$

$$\lambda = \frac{1}{2}(-(\mu\omega^2 + D\omega^2 + k) \pm \sqrt{(\mu\omega^2 + D\omega^2 + k)^2 - 4(\mu\omega(D\omega^2 + k) - \chi a_{eq}\omega^2 f)}) \tag{14.3.35}$$

Now, the question is whether either eigenvalue's real part could be positive. Since all the parameters are non-negative in this model, the term before "$\pm$" can't be positive by itself. This means that the inside the radical must be positive and sufficiently large in order to make the real part of the eigenvalue positive. Therefore, the condition for a positive real part to arise is as follows:

$$(\mu\omega^2 + D\omega^2 + k) < \sqrt{(\mu\omega^2 + D\omega^2 + k)^2 - 4(\mu\omega^2(D\omega^2 + k) - \chi a_{eq}\omega^2 f)} \tag{14.3.36}$$

If this is the case, the first eigenvalue (with the "+" operator in the parentheses) is real and positive, indicating that the homogeneous equilibrium state is unstable. Let's simplify the inequality above to get a more human-readable result:

$$(\mu\omega^2 + D\omega^2 + k)^2 < (\mu\omega^2 + D\omega^2 + k)^2 - 4(\mu\omega^2 9D\omega^2 + k) - \chi a_{eq}\omega^2 f) \tag{14.3.37}$$

$$0 < -4(\mu\omega^2(D\omega^2) + k - \chi a_{eq}\omega^2 f) \tag{14.3.38}$$

$$\chi a_{eq}\omega^2 f > \mu\omega(D\omega^2 + k) \tag{14.3.39}$$

$$\chi a_{eq} f > \mu(D\omega^2 + k) \tag{14.3.40}$$

At last, we have obtained an elegant inequality that ties all the model parameters together in a very concise mathematical expression. If this inequality is true, the homogeneous equilibrium state of the Keller-Segel model is unstable, so it is expected that the system will show a spontaneous pattern formation. One of the benefits of this kind of mathematical analysis is that we can learn a lot about each model parameter's effect on pattern formation all at once. From inequality 14.3.40, for example, we can make the following predictions about the aggregation process of slime mold cells (and people too, if we consider this a model of population-economy interaction):

- χ, a_eq, and f on the left hand side indicate that the aggregation of cells (or the concentration of population in major cities) is more likely to occur if
  - the cells' chemotaxis (or people's "moneytaxis") is stronger (χ),

– there are more cells (or people) in the system (a_eq), and/or

– the cells (or people) produce cAMP molecules (or economic values) at a faster pace (f).

- μ, D, and k on the right hand side indicate that the aggregation of cells (or the concentration of population in major cities) is more likely to be suppressed if

  – the cells and cAMP molecules (or people and economic values) diffuse faster (μ and D), and/or

  – the cAMP molecules (or economic values) decay more quickly (k).

It is quite intriguing that such an abstract mathematical model can provide such a detailed se of insights into the problem of urbanization (shift of population and economy from rural to urban areas), one of the critical socio-economical issues our modern society is facing today. Isn't it?

Moreover, solving inequality 14.3.40 in terms of ω^2 gives us the critical condition between homogenization and aggregation:

$$\frac{\chi a_{eq} f - \mu k}{\mu D} > \omega^2 \tag{14.3.41}$$

Note that ω can be any real number but ω^2 has to be non-negative, so aggregation occurs if and only if

$$\chi a_{eq} f > \mu k. \tag{14.3.42}$$

And if it does, the spatial frequencies of perturbations that are going to grow will be given by

$$\frac{\omega}{2\pi} < \frac{1}{2\pi}\sqrt{\frac{\chi a_{eq} f - \mu k}{\mu D}}. \tag{14.3.43}$$

The wave length is the inverse of the spatial frequency, so we can estimate the length of the growing perturbations as follows:

$$l = \frac{2\pi}{\omega} > 2\pi\sqrt{\frac{\mu D}{\chi a_{eq} f - \mu k}} = l_c \tag{14.3.44}$$

This result means that spatial perturbations whose spatial length scales are greater than l_c are going to grow and become visible, while perturbations with length scales smaller than l_c are going to shrink and disappear. This critical length scale tells us the characteristic distance between aggregated points (or cities) spontaneously forming at the beginning of the process.

We can confirm these analytical results with numerical simulations. Figure 14.6.3 shows simulation results with $\mu = 10^{-4}$, $D = 10^{-4}$, $f = 1$, $k = 1$, and $a_eq = 1$, while χ is varied as a control parameter. With these parameter values, inequality 14.3.42 predicts that the critical value of χ above which aggregation occurs will be

$$\chi_c - \frac{\mu k}{a_{eq} f} = 10^{-4}. \tag{14.3.45}$$

which is confirmed in the simulation results perfectly.

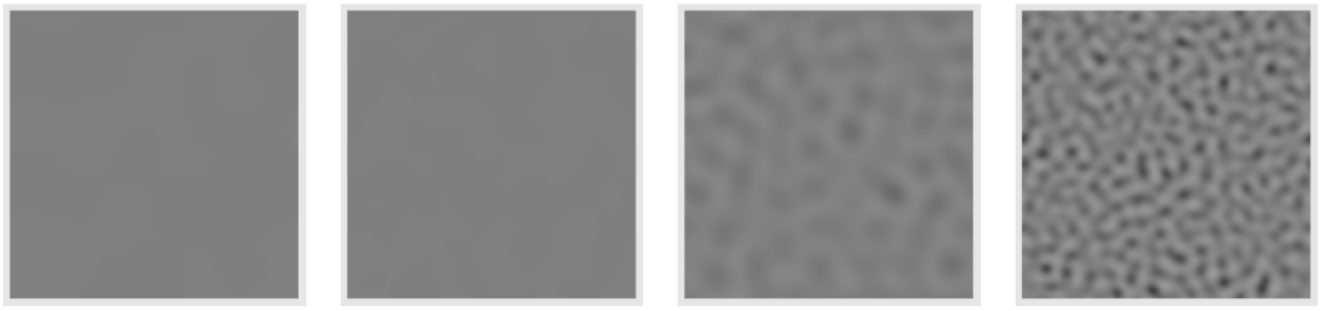


Figure 14.3.3: Numerical simulation results of the Keller-Segel model with $\mu = 10^{-4}$, $D = 10^{-4}$, $f = 1$, $k = 1$, and $a_{eq} = 1$. Cell densities are plotted in grayscale (darker = greater). The value of χ is given below each result.

This concludes a guided tour of the linear stability analysis of continuous field models. It may have looked rather complicated, but the key ideas are simple and almost identical to those of linear stability analysis of non-spatial models. Here is a summary of the procedure:

Linear stability analysis of continuous-field models

1. Find a homogeneous equilibrium state of the system you are interested in.
2. Represent the state of the system as a sum of the homogeneous equilibrium state and a small perturbation function.
3. Represent the small perturbation function as a product of a dynamic amplitude and a static shape of the perturbation, which is chosen to be an eigenfunction of the spatial linear operator remaining in the equation (most likely just sine waves for ∇^2).
4. Eliminate the spatial linear operator and ignore higher-order terms of small perturbations to simplify the equation into a linear non-spatial form.
5. Calculate the eigenvalues of the resulting coefficient matrix.
6. If the real part of the dominant eigenvalue is:
    - Greater than 0⇒The homogeneous equilibrium state is unstable.
    - Less than 0⇒The homogeneous equilibrium state is stable
    - . • Equal to 0 ⇒ The homogeneous equilibrium state *may* be neutral (Lyapunov stable).
7. In addition, if there are complex conjugate eigenvalues involved, oscillatory dynamics are going on around the homogeneous equilibrium state.

Finally, we should also note that this eigenfunction-based linear stability analysis of continuous-field models works only if the spatial dynamics are represented by a local linear differential operator (e.g., $\partial f/\partial x$, Laplacian, etc.). Unfortunately, the same approach wouldn't be able to handle global or nonlinear operators, which often arise in mathematical models of real-world phenomena. But this is beyond the scope of this textbook.

Exercise 14.3.5

Add terms for population growth and decay to the first equation of the Keller-Segel model. Obtain a homogeneous equilibrium state of the revised model, and then conduct a linear stability analysis. Find out the condition for which spontaneous pattern formation occurs. Interpret the result and discuss its implications.

[1]In mathematical terms, this is an example of *separation of variables*—breaking down the original equations into a set of simpler components each of which has fewer independent variables than the original ones. PDEs for which separation of variables is possible are called *separable PDEs*. Our original Keller-Segel model equations are *not* separable PDEs, but we are trying to separate variables anyway by focusing on the dynamics around the model's homogeneous equilibrium state and using linear approximation.

This page titled 14.3: Linear Stability Analysis of Continuous Field Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

14.4: Linear Stability Analysis of Reaction-Diffusion Systems

You may have found that the linear stability analysis of continuous field models isn't as easy as that of non-spatial models. For the latter, we have a very convenient tool called the Jacobian matrices, and the stability analysis is just calculating a Jacobian matrix and then investigating its eigenvalues. Everything is so mechanistic and automatic, compared to what we went through in the previous section. You may wonder, aren't there any easier shortcuts in analyzing the stability of continuous field models?

Well,if you feel that way, you will become a big fan of the *reaction-diffusion systems* we discussed in Section 13.6. Their linear stability analysis is *much easier*, because of the clear separation of local reaction dynamics and spatial diffusion dynamics. To be more specific, you can bring the Jacobian matrix back to the analysis! Here is how and why it works.

Consider conducting a linear stability analysis to the following standard reaction-diffusion system:

$$\frac{\partial f_1}{\partial t} = R_1(f_1, f_2, \cdots, f_n) + D_1 \nabla^2 f_1 \tag{14.4.1}$$

$$\frac{\partial f_2}{\partial t} = R_2(f_1, f_2, \cdots, f_n) + D_2 \nabla^2 f_2 \tag{14.4.2}$$

$$\vdots$$

$$\frac{\partial f_n}{\partial t} = R_n(f_1, f_2, \cdots, f_n) + D_n \nabla^2 f_n \tag{14.4.3}$$

The homogeneous equilibrium state of this system, $(f_{1eq}, f_{2eq}, \ldots, f_{neq})$, is a solution of the following equations:

$$0 = R_1(f_{1eq}, f_{2eq}, \cdots . f_{neq}) \tag{14.4.4}$$

$$0 = R_2(f_{1eq}, f_{2eq}, \cdots . f_{neq}) \tag{14.4.5}$$

$$\vdots$$

$$0 = R_n(f_{1eq}, f_{2eq}, \cdots . f_{neq}) \tag{14.4.6}$$

To conduct a linear stability analysis, we replace the original state variables as follows:

$$f_i(x,t) \Rightarrow f_{ieq} + \Delta f_i(x,t) = f_{ieq} + \sin(\omega x + \phi)\Delta f_1(t) \text{for all i} \tag{14.4.7}$$

This replacement turns the dynamical equations into the following form:

$$S\frac{\partial \Delta f_1}{\partial t} = R_1(f_{1eq} + S\Delta f_1 . f_{2eq} + S\Delta f_2, \cdots f_{neq} + S\Delta f_n) - D_1\omega^2 S\Delta f_1 \tag{14.4.8}$$

$$S\frac{\partial \Delta f_2}{\partial t} = R_2(f_{1eq} + S\Delta f_1 . f_{2eq} + S\Delta f_2, \cdots f_{neq} + S\Delta f_n) - D_2\omega^2 S\Delta f_2 \tag{14.4.9}$$

$$\vdots$$

$$S\frac{\partial \Delta f_n}{\partial t} = R_n(f_{1eq} + S\Delta f_1, f_{2eq} + S\Delta f_2, \cdots f_{neq} + S\Delta f_n) - D_n\omega^2 S\Delta f_n \tag{14.4.10}$$

Here I used $S = sin(\omega x + \varphi)$ only in the expressions above to shorten them. These equations can be summarized in a single vector form about Δf,

$$\sin(\omega x + \phi)\frac{\partial \Delta f}{\partial t} = R(f_{eq} + \sin(\omega x + \phi)\Delta f) - D\omega^2 \sin \omega x + \phi)\Delta f, \tag{14.4.11}$$

where R is a vector function that represents all the reaction terms, and D is a diagonal matrix whose diagonal components are D_i for the i-th position. Now that all the diffusion terms have been simplified, if we can also linearize the reaction terms, we can complete the linearization task. And this is where the Jacobian matrix is brought back into the spotlight. The reaction terms are all local without any spatial operators involved, and therefore, from the discussion in Section 5.7, we know that the vector function $R(f_{eq} + \sin(\omega x + \varphi)\ \Delta f)$ can be linearly approximated as follows:

$$R(f_{eq} + \sin(\omega x + \phi)\Delta f) \approx R(f_{eq}) + \begin{pmatrix} \frac{\partial R_1}{\partial f_1} & \frac{\partial R_1}{\partial f_2} & \dots & \frac{\partial R_1}{\partial f_n} \\ \frac{\partial R_2}{\partial f_1} & \frac{\partial R_2}{\partial f_2} & \dots & \frac{\partial R_2}{\partial f_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial f_1} & \frac{\partial R_n}{\partial f_2} & \dots & \frac{\partial R_n}{\partial f_n} \end{pmatrix} |_{f=f_{eq}} \sin(\omega x + \phi)\Delta f \tag{14.4.12}$$

$$= \sin(\omega x + \phi) \begin{pmatrix} \frac{\partial R_1}{\partial f_1} & \frac{\partial R_1}{\partial f_2} & \dots & \frac{\partial R_1}{\partial f_n} \\ \frac{\partial R_2}{\partial f_1} & \frac{\partial R_2}{\partial f_2} & \dots & \frac{\partial R_2}{\partial f_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial f_1} & \frac{\partial R_n}{\partial f_2} & \dots & \frac{\partial R_n}{\partial f_n} \end{pmatrix} |_{f=f_{eq}} \Delta f \tag{14.4.13}$$

Note that we can eliminate $R(f_eq)$ because of Eqs. 14.4.4, 14.4.5 14.4.6. By plugging this result into Equation 14.4.11, we obtain

$$\sin(\omega x + \phi)\frac{\partial \Delta f}{\partial t} = \sin(\omega x + \phi) J|_{f=f_{eq}} \Delta f = D\omega^2 \sin(\omega x + \phi)\Delta f, \tag{14.4.14}$$

$$\frac{\partial \Delta f}{\partial t} - (J - D\omega^2)|_{f=f_{eq}} \Delta f, \tag{14.4.15}$$

where J is the Jacobian matrix of the reaction terms (R). Very simple! Now we just need to calculate the eigenvalues of this coefficient matrix to study the stability of the system.

> The stability of a reaction-diffusion system at its homogeneous equilibrium state f_{eq} can be studied by calculating the eigenvalues of
>
> $$(J - D\omega^2)|_{f=f_{eq}}, \tag{14.4.16}$$
>
> where J is the Jacobian matrix of the reaction terms, D is the diagonal matrix made of diffusion constants, and w is a parameter that determines the spatial frequency of perturbations.

This shortcut in linear stability analysis is made possible thanks to the clear separation of reaction and diffusion terms in reaction-diffusion systems. I hope you now understand part of the reasons why so many researchers are fond of this modeling framework.

Let's apply this new knowledge to some example. Here is the Turing model we discussed before:

$$\frac{\partial u}{\partial t} = a(u - h) + b(v - k) + D_u \nabla^2 u \tag{14.4.17}$$

$$\frac{\partial v}{\partial t} = c(v - h) + d(v - k) + D_v \nabla^{2v} \tag{14.4.18}$$

Using Equation 14.4.16, we can immediately calculate its coefficient matrix:

$$(\begin{pmatrix} a & \\ c & d \end{pmatrix} - \begin{pmatrix} D_u & 0 \\ 0 & D_v \end{pmatrix} \omega^2)|_{(u,v)=(h,k)} = \begin{pmatrix} a - D_u\omega^2 & b \\ c & d - D_v\omega^2 \end{pmatrix} \tag{14.4.19}$$

From the discussion in Section 7.4, we already know that, in order for this matrix to show stability, its determinant must be positive and its trace must be negative (see Fig. 7.4.1). Therefore, the condition for the homogeneous equilibrium state of this system to be stable is that both of the following two inequalities must be true for all real values of ω:

$$0 < (a - D_u\omega^2)(d - D_v\omega^2) - bc \tag{14.4.20}$$

$$0 > a - D_u\omega^2 + d - D_v\omega^2 \tag{14.4.21}$$

These inequalities can be rewritten using $det(A)$ and $Tr(A)$ of $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, as follows:

$$aD_v\omega^2 + dD_u\omega^2 - D_uD_v\omega^4 < \det(A) \tag{14.4.22}$$

$$D_u\omega^2 + D_v\omega^2 > Tr(A) \tag{14.4.23}$$

Now, imagine that the original non-spatial model without diffusion terms was already stable, i.e., $det(A) > 0$and $Tr(A) < 0$. Is there any possibility that the introduction of diffusion to the model could destabilize the system by itself? The second inequality is always true for negative \(Tr(A)), because its left hand side can't be negative. But the first inequality can be violated, if

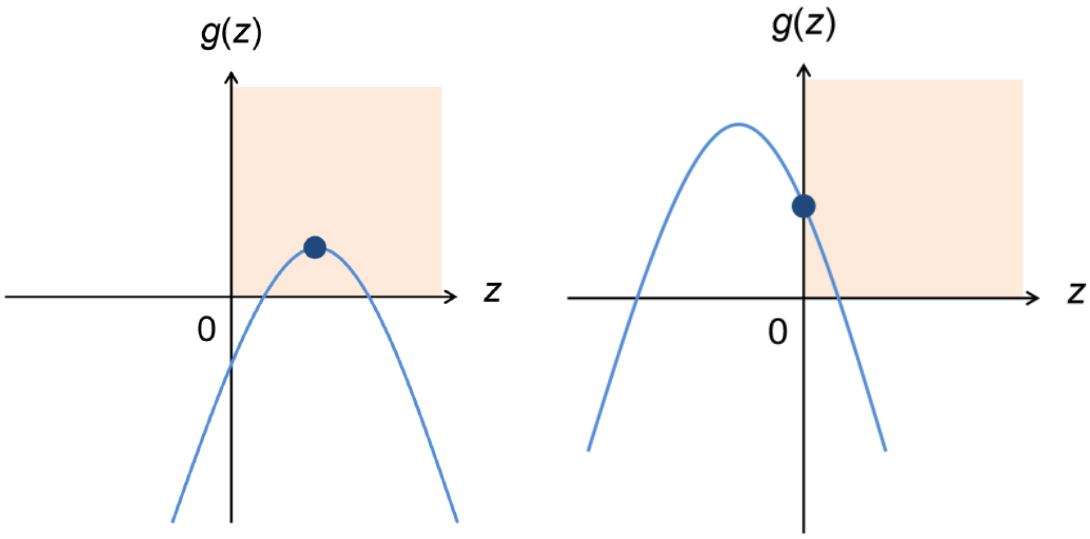
$$g(z) = -D_uD_vz^2 + (aD_v + dD_u)z - \det(A) \qquad (with\ z = \omega^2) \tag{14.4.24}$$

can take a positive value for some $z > 0$. $g(z)$ can be rewritten as

$$g(z) = -D_uD_v(z - \frac{aD_v + dD_u}{2D_{uD_v}})^2 + \frac{(aD_v + dD_u)^2}{4D_uD_v} - \det(A). \tag{14.4.25}$$

There are two potential scenarios in which this polynomial can be positive for some $z > 0$, as shown in Fig. 14.4.1.

***Figure* 14.4.1**: Two possible scenarios in which $g(z)$ can take a positive value for some $z > 0$. Left: When the peak exists on the positive side of z. Right: When the peak exists on the negative side of z.



If the peak exists on the positive side of $z(aD_v + dD_u > 0$; Fig. 14.4.1 left), the only condition is that the peak should stick out above the z-axis, i.e.

$$\frac{(aD_v + dD_u)^2}{4D_uD_v} - \det(A) > 0. \tag{14.4.26}$$

Or, if the peak exists on the negative side $z(aD_v + dD_u < 0$; Fig. 14.4.1 right), the condition is that the intercept of $g(z)$ should be positive, i.e.,

$$g(0) = -\det(A) > 0, \tag{14.4.27}$$

but this can't be true if the original non-spatial model is stable. Therefore, the only possibility for diffusion to destabilize the otherwise stable system is the first case, whose condition can be simplified to

$$aD_v + dD_u > 2\sqrt{D_uD_v \det(A)}. \tag{14.4.28}$$

Let's test how well this prediction applies to actual dynamics of Turing models. In the previous chapter, we used $(a, b, c, d) = (1, -1, 2, -1.5)$ and $(D_u, D_v) = (10^{-4}, 6 \times 10^{-4})$ to generate the simulation result shown in Fig. 13.17. With these parameter settings, det(A) = −1.5−(−2) = 0.5 > 0 and Tr(A) = −0.5 < 0, so the system would be stable if there were no diffusion terms. However,

$$aD_v + dD_u = 6 \times 10^{-4} - 1.5 \times 106{-}4 = 4.5 \times 10^{-4}, \tag{14.4.29}$$

$$2\sqrt{D_u D_v \det(A)} = 2\sqrt{10^{-4} \times 6 \times 10^{-4} \times 0.5} = 2 \times 10^{-4}\sqrt{3} \approx 3.464 \times 10^{-4} \tag{14.4.30}$$

therefore inequality 14.4.28 holds. This indicates that the homogeneous equilibrium state must be unstable and non-homogeneous spatial patterns should arise which you can actually see in Fig.13.17. As briefly mentioned in Section 13.6, this is called the *diffusion induced instability*. It is quite a counter-intuitive phenomenon, because diffusion is usually considered a process where a non-homogeneous structure is being destroyed by random motion. But here, the system is stable at its homogeneous state w*ithout diffusion*, but it can spontaneously create non-homogeneous structures *with diffusion*. This is a really nice example of how mind-boggling the behavior of complex systems can be sometimes.

? Exercise 14.4.1

Conduct a linear stability analysis of the spatially extended predator-prey model, around its non-zero homogeneous equilibrium state, and discuss the results:

$$\frac{\partial r}{\partial t} = ar - brf + D_r \nabla^2 r \tag{14.4.31}$$

$$\frac{\partial f}{\partial t} = -cf - drf + D_f \nabla^2 f \tag{14.4.32}$$

Assume that all model parameters are positive.

? Exercise 14.4.2

Conduct a linear stability analysis of the Gray-Scott model, around its homogeneous equilibrium state $(u_{eq}, v_{eq}) = (1, 0)$, and discuss the results:

$$\frac{\partial u}{\partial t} = F(1-u) - uv^2 + D_u \nabla^2 u \tag{14.4.33}$$

\[\frac{\partial{v}}{\partial{t}} =-(F+k) v +uv^{2} +D_{v} \nabla^{2}v \label{(14.113)}]

Again, assume that all model parameters are positive.

There are a few more useful predictions we can make about spontaneous pattern formation in reaction-diffusion systems. Let's continue to use the Turing model discussed above as an example. We can calculate the actual eigenvalues of the coefficient matrix, as follows:

$$\begin{vmatrix} 1 - 10 - 4\omega^2 - \lambda & -1 \\ 2 & -1.5 - 6 \times 10 - 4\omega^2 - \lambda \end{vmatrix} = 0 \tag{14.4.34}$$

$$(1 - 10 - 4\omega^2 - \lambda)(-1.5 - 6 \times 10 - 4\omega^2 - \lambda) - (-2) = 0 \tag{14.4.35}$$

$$\lambda^2 + (0.5 + 7 \times 10 - 4\omega^2)\lambda + (1 - 10^{-4}\omega^2)(-1.5 - 6 \times 10^{-4}\omega^2) + 2 = 0 \tag{14.4.36}$$

$$\lambda = \frac{1}{2}(0.5 + 7 \times 10^{-4}\omega^2) \pm \sqrt{(0.5 + 7 \times 10^{-4}\omega^2)2 - 4(1 - 10^{-4}\omega^2)(-1.5 - 6 \times 10^{-4}\omega^2) - 8)} \tag{14.4.37}$$

$$= \frac{1}{2} - (0.5 + 7 \times 10^{-4}\omega^2) \pm \sqrt{2.5 \times 10^{-7} w^4 + 2.5 \times 10^{-3} w^2 - 1.75)} \tag{14.4.38}$$

Out of these two eigenvalues, the one that could have a positive real part is the one with the "+" sign (let's call it \(λ_{+}).

Here, what we are going to do is to calculate the value of ω that attains the largest real part of λ_+. This is a meaningful question, because the largest real part of eigenvalues corresponds to the dominant eigenfunction ($\sin(\omega x + \varphi)$) that grows fastest, which should be the most visible spatial pattern arising in the system's state. If we find out such a value of ω, then $2\pi/\omega$ gives us the length scale of the dominant eigenfunction.

We can get an answer to this question by analyzing where the extremum of λ_+ occurs. To make analysis simpler, we let $z = \omega^2$ again and use z as an independent variable, as follows:

$$\frac{d\lambda_+}{dz} = \frac{1}{2}(-7\times10^{-4} + \frac{5\times10^{-7}z + 2.5\times10^{-3}}{2\sqrt{2.5\times10^{-7} + 2.5\times10^{-3}z - 1.75}}) = 0 \tag{14.4.39}$$

$$7\times10^{-4}(2\sqrt{2.5\times10^{-7}z^2 2.5\times10^{-3}z - 1.75}) = 5\times10^{-7}z + 2.5\times10^{-3} \tag{14.4.40}$$

$$1.96\times10^{-6}(2.5\times10^{-7}z^2 + 2.5\times10^{-3}z - 1.75)$$

$$= 2.5\times10^{-13}z^2 + 2.5\times10^{-9}z + 6.25\times10^{-6} \tag{14.4.41}$$

$$(\ldots blah\ blah\ blah\ldots) \tag{14.4.42}$$

$$2.4\times10^{-13}z^2 + 2.4\times10^{-9}z - 9.68\times10^{-6} = 0 \tag{14.4.43}$$

$$z = 3082.9, -13082.9 \tag{14.4.44}$$

Phew. Exhausted. Anyway, since $z = \omega^2$, the value of ω that corresponds to the dominant eigenfunction is

$$\omega = \sqrt{3082.9} = 55.5239. \tag{14.4.45}$$

Now we can calculate the length scale of the corresponding dominant eigenfunction, which is

$$= \frac{2\pi}{\omega} \approx 0.113162. \tag{14.4.46}$$

This number gives the characteristic distance between the ridges(or between the valleys) in the dominant eigenfunction, which is measured in unit length. 0.11 means that there are about nine ridges and valleys in one unit of length. In fact, the simulation shown in Fig. 13.17 was conducted in a $[0,1]\times[0,1]$ unit square domain, so go ahead and count how many waves are lined up along its x- or y-axis in its final configuration.

Have you finished counting them? Yes, the simulation result indeed showed about nine waves across each axis! This is an example of how we can predict not only the stability of the homogeneous equilibrium state, but also the characteristic length scale of the spontaneously forming patterns if the equilibrium state turns out to be unstable.

? Exercise 14.4.3

Estimate the length scale of patterns that form in the above Turing model with $(a,b,c,d) = (0.5,-1,0.75,-1)$ and $(D_u, D_v) = (10^{-4}, 10^{-3})$. Then confirm your prediction with numerical simulations.

? Exercise 14.4.4

If both diffusion constants are multiplied by the same factor ψ,how does that affect the length scale of the patterns?

Okay, let's make just one more prediction, and we will be done. Here we predict the critical ratio of the two diffusion constants, at which the system stands right at the threshold between homogenization and pattern formation. Using a new parameter $\rho = D_v/D_u$, the condition for instability (inequality 14.4.28) can be further simplified as follows:

$$a\rho D_u + dD_u > 2\sqrt{\rho D_u^2 \det(A)} \tag{14.4.47}$$

$$a\rho + d > 2\sqrt{\rho \det(A)} \tag{14.4.48}$$

For the parameter values we used above, this inequality is solved as follows:

$$\rho - 1.5 > 2\sqrt{0.5\rho} \tag{14.4.49}$$

$$\rho^2 - 5\rho + 2.25 > 0 (\text{with } \rho - 1.5 > 0) \tag{14.4.50}$$

$$\rho > 4.5 \tag{14.4.51}$$

This means that the diffusion of v must be at least 4.5 times faster than u in order to cause the diffusion instability. In other words, u acts more locally, while the effects of v reach over longer spatial ranges. If you look back at the original coefficient matrix $\begin{pmatrix} 1 & -1 \\ 2 & -1.5 \end{pmatrix}$, you will realize that u tends to increase both u and v, while v tends to suppress u and v. Therefore, this represents typical "short-range activation and long-range inhibition" dynamics that we discussed in Section 11.5, which is essential in many pattern formation processes.

Figure 14.4.2 shows the numerical simulation results with the ratio of the diffusion constants systematically varied. Indeed, a sharp transition of the results across $\rho = 4.5$ is actually observed! This kind of transition of a reaction-diffusion system's behavior between homogenization and pattern formation is called a *Turing bifurcation*, which Turing himself showed in his monumental paper in the 1950s [44].

? Exercise 14.4.5

Below is a variant of the Turing pattern formation model:

$$\frac{\partial u}{\partial t} = u(v-1) - \alpha + D_u \nabla^2 u \tag{14.4.52}$$

$$\frac{\partial v}{\partial t} = \beta - uv + D_v \nabla^2 v \tag{14.4.53}$$

Here α and β are positive parameters. Let $(\alpha, \beta) = (12, 16)$ throughout this exercise. Do the following:

1. Find its homogeneous equilibrium state.
2. Examine the stability of the homogeneous equilibrium state without diffusion terms.
3. With $(D_u, D_v) = (10^{-4}, 10^{-3})$, conduct a linear stability analysis of this model around the homogeneous equilibrium state to determine whether nonhomogeneous patterns form spontaneously. If they do, estimate the length scale of the patterns.
4. Determine the critical ratio of the two diffusion constants.
5. Confirm your predictions with numerical simulations.



Figure 14.4.2: Numerical simulation results of the Turing pattern formation model with $(a, b, c, d) = (1, -1, 2, -1.5)$, $D_u = 10^{-4}$, and $D_v = \rho D\_u$. Densities of u are plotted in grayscale (darker = greater). The value of ρ is given below each result.

This page titled 14.4: Linear Stability Analysis of Reaction-Diffusion Systems is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

15: Basics of Networks

15.1: Network Models

15.2: Terminologies of Graph Theory

15.3: Constructing Network Models with NetworkX

15.4: Visualizing Networks with NetworkX

15.5: Importing/Exporting Network Data

15.6: Generating Random Graphs

This page titled 15: Basics of Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.1: Network Models

We are now moving into one of the most recent developments of complex systems science: *networks*. Stimulated by two seminal papers on *small-world* and *scale-free networks* published in the late 1990s [56, 57], the science of complex networks, or *network science* for short, has been rapidly growing and producing novel perspectives, research questions, and analytical tools to study various kinds of systems in a number of disciplines, including biology, ecology, sociology, economics, political science, management science, engineering, medicine, and more [23, 24, 25].

The historical roots of network science can be sought in several disciplines. One is obviously discrete mathematics, especially *graph theory*, where mathematicians study various properties of abstract structures called *graphs* made of *nodes* (a.k.a. vertices—plural of *vertex*) and *edges* (a.k.a. links, ties). Another *theoretical* root is statistical *physics*, where properties of collective systems made of a large number of entities (such as phase transitions) are studied using analytical means. A more applied root of network science is in the social sciences, especially *social network analysis* [58, 59, 60]. Yet another application-oriented root would be in dynamical systems, especially *Boolean networks* discussed in theoretical and systems biology [22, 61] and artificial neural networks discussed in computer science [20, 21]. In all of those investigations, the research foci were put on the connections and interactions among the components of a system, not just on each individual component.

Network models are different from other more traditional dynamical models in some fundamental aspects. First, the components of the system may not be connected uniformly and regularly, unlike cells in cellular automata that form regular homogeneous grids. This means that, in a single network, some components may be very well connected while others may not. Such non-homogeneous connectivity makes it more difficult to analyze the system's properties mathematically (e.g., mean-field approximation may not apply to networks so easily). In the meantime, it also gives the model greater power to represent connections among system components more closely with reality. You can represent any *network topology* (i.e., shape of a network) by explicitly specifying in detail which components are connected to which other components, and how. This makes network modeling necessarily *data-intensive*. No matter whether the network is generated using some mathematical algorithm or reconstructed from real-world data, the created network model will contain a good amount of detailed information about how exactly the components are connected. We need to learn how to build, manage, and manipulate these pieces of information in an efficient way.

Second, the number of components may dynamically increase or decrease over time in certain dynamical network models. Such growth (or decay) of the system's topology is a common assumption typically made in generative network models that explain self organizing processes of particular network topologies. Note, however, that such a dynamic change of the number of components in a system realizes a *huge* leap from the other more conventional dynamical systems models, including all the models we have discussed in the earlier chapters. This is because, when we consider states of the system components, having one more (or less) component means that the system's phase space acquires one more (or less) dimensions! From a traditional dynamical systems point of view, it sounds almost illegal to change the dimensions of a system's phase space over time, yet things like that do happen in many real-world complex systems. Networkmodels allow us to naturally describe such crazy processes.

This page titled 15.1: Network Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.2: Terminologies of Graph Theory

Before moving on to actual dynamical network modeling, we need to cover some basics of graph theory, especially the definitions of technical terms used in this field. Let's begin with something we have already discussed above:

> A *network* (or *graph*) consists of a set of *nodes* (or *vertices*, *actors*) and a set of *edges* (or *links*, *ties*) that connect those nodes.

As indicated above, different disciplines use different terminologies to talk about networks; mathematicians use "graph/vertex/edge," physicists use "network/node/edge," computer scientists use "network/node/link," social scientists use "network/actor/tie," etc. This is a typical problem when you work in an interdisciplinary research area like network science. We just need to get used to it. In this textbook, I mostly use "network/node/edge" or "network/node/link," but I may sometimes use other terms interchangeably as well.

By the way, some people ask what are the differences between a "network" and a "graph." I would say that a "network" implies it is a model of something real, while a "graph" emphasizes more on the aspects as an abstract mathematical object (which can also be used as a model of something real, of course). But this distinction isn't so essential either.

To represent a network, we need to specify its nodes and edges. One useful term is *neighbor*, defined as follows:

> Node j is called a neighbor of node i if (and only if) node i is connected to node j.

The idea of neighbors is particularly helpful when we attempt to relate network models with more classical models such as CA (where neighborhood structures were assumed regular and homogeneous). Using this term, we can say that the goal of network representation is to represent neighborhood relationships among nodes. There are many different ways of representing networks, but the following two are the most common:

> **Adjacency matrix** A matrix with rows and columns labeled by nodes, whose i-th row, j-th column component a_{ij} is 1 if node i is a neighbor of node j, or 0 otherwise.
>
> **Adjacency list** A list of lists of nodes whose i-th component is the list of node i's neighbors.

Figure15.1 shows an example of the adjacency matrix/list. As you can see, the adjacency list offers a more compact, memory-efficient representation, especially if the network is sparse (i.e., if the network density is low—which is often the case for most real-world networks). In the meantime, the adjacency matrix also has some benefits, such as its feasibility for mathematical analysis and easiness of having access to its specific components.

Here are some more basic terminologies:

> **Degree** The number of edges connected to a node. Node i's degree is often written as $deg(i)$.

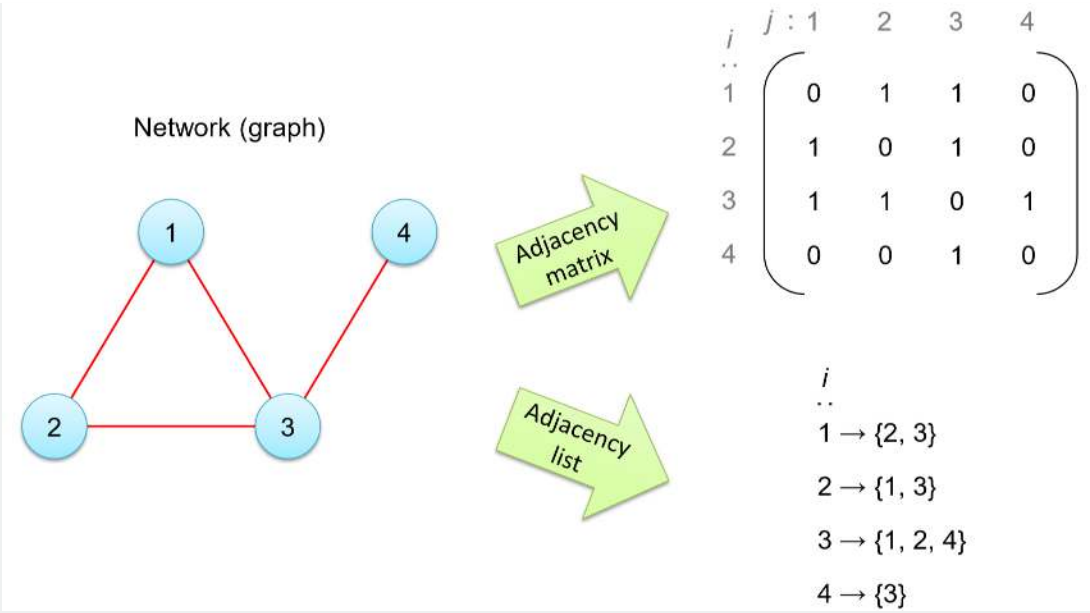
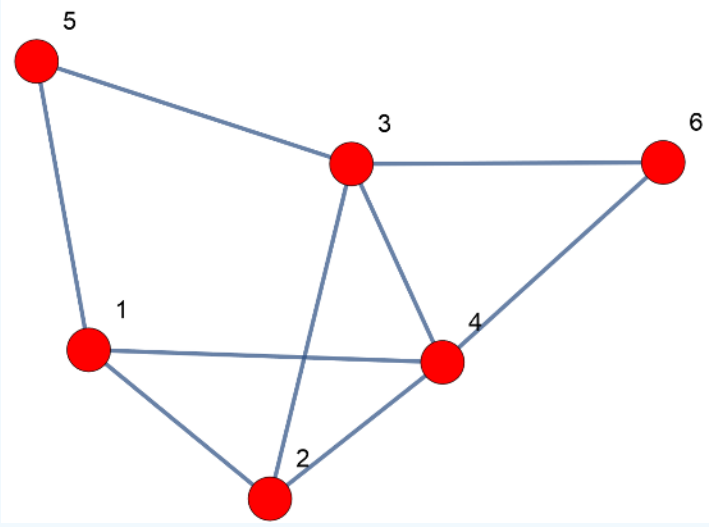


Figure **15.2.1**: Examples of an adjacent matrix and an adjacent list.

- **Walk** A list of edges that are sequentially connected to form a continuous route on a network. In particular:
- **Trail** A walk that doesn't go through any edge more than once.
- **Path** A walk that doesn't go through any node (and therefore any edge, too) more than once.
- **Cycle** A walk that starts and ends at the same node without going through any node more than once on its way.
- **Subgraph** Part of the graph.
- **Connected graph** A graph in which a path exists between any pair of nodes.
- **Connected component** A subgraph of a graph that is connected within itself but not connected to the rest of the graph.

? Exercise 15.2.1

See the following network and answer the following questions:



1. Represent the network in (a) an adjacency matrix, and (b) an adjacency list.
2. Determine the degree for each node.
3. Classify the following walks as trail, path, cycle, or other.
    - 6 → 3 → 2 → 4 → 2 → 1

- $1 \to 4 \to 6 \to 3 \to 2$
- $5 \to 1 \to 2 \to 3 \to 5$

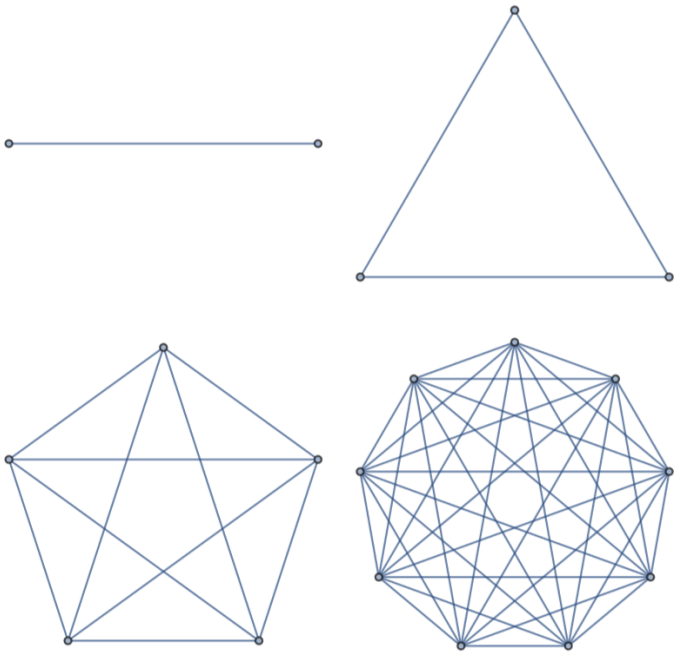
4. Identify all fully connected three-node subgraphs (i.e., triangles).

5. Remove nodes 3 and 4 (and all edges connected to them). Then identify the connected components in the resulting graph.

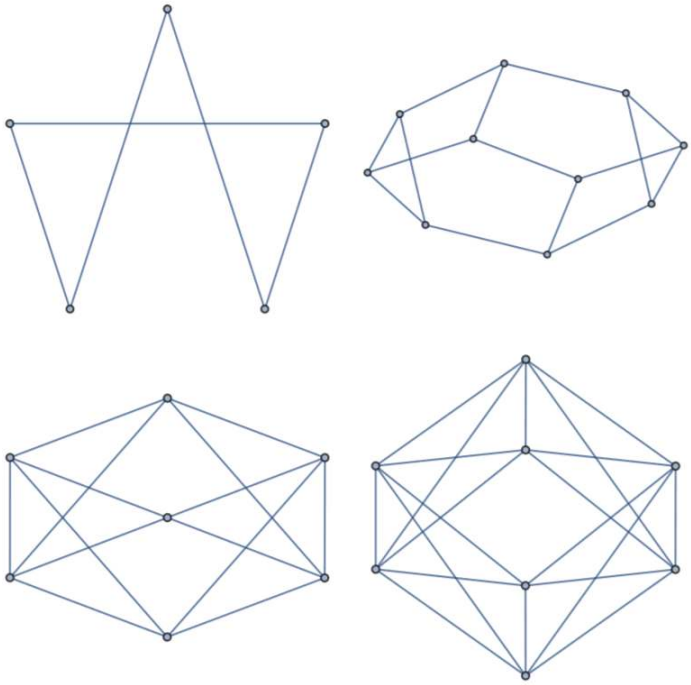
Some graphs with characteristic topological properties are given their own unique names, as follows

- **Complete graph** A graph in which any pair of nodes are connected (Fig. 15.2.2A).
- **Regular graph** A graph in which all nodes have the same degree(Fig.15.2.2B).Every complete graph is regular.
- **Bipartite (n -partite) graph** A graph whose nodes can be divided into two (or n) groups so that no edge connects nodes within each group (Fig. 15.2.2C).
- **Tree graph** A graph in which there is no cycle (Fig. 15.2.2D). A graph made of multiple trees is called a forest graph. Every tree or forest graph is bipartite.
- **Planar graph** A graph that can be graphically drawn in a two-dimensional plane with no edge crossings (Fig. 15.2.2E). Every tree or forest graph is planar.

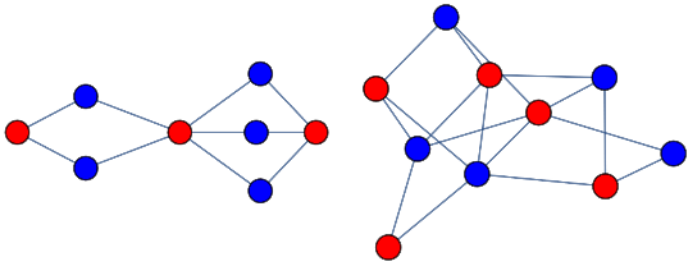
A



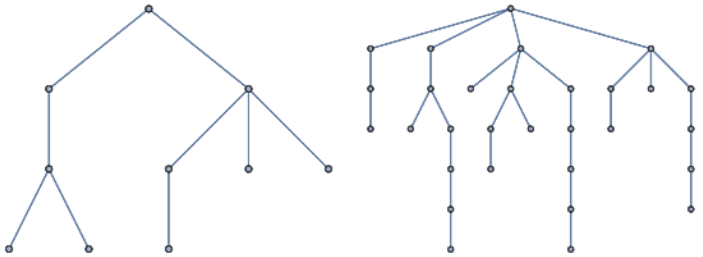
B



C



D



E

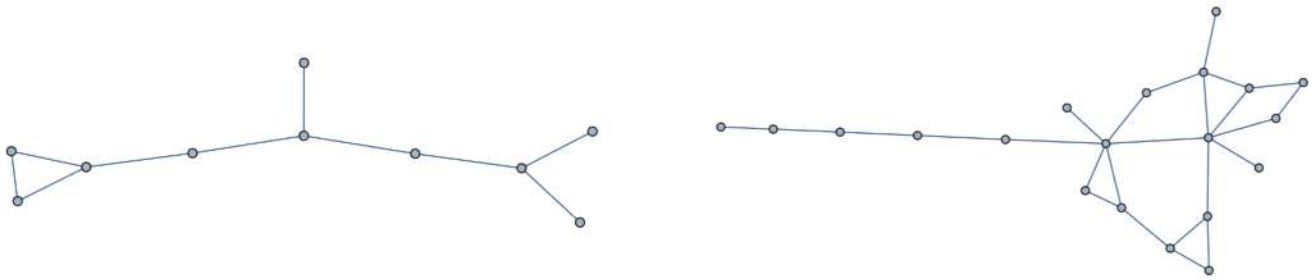


Figure 15.2.2: Examples of graphs with specific names. A: Complete graphs. B:Regular graphs. C: Bipartite graphs (colors show groups). D: Tree graphs. E: Planar graphs.

? Exercise 15.2.2

Calculate the following:

- The number of edges in a complete graph made of n nodes
- The number of edges in a regular graph made of n nodes each of which has degree k

? Exercise 15.2.3

Determine whether the following graphs are planar or not:

- **A complete graph made of four nodes**
- **A complete graph made of five nodes**
- **A bipartite graph made of a two-node group and a three-node group in which all possible inter-group edges are present**

- **A bipartite graph made of two three-node groups in which all possible intergroup edges are present**

? Exercise 15.2.4

Every connected tree graph made of n nodes has exactly $n-1$edges. Explain why.

There are also classifications of graphs according to the types of their edges:

- **Undirected edge** A symmetric connection between nodes. If node i is connected to node j by an undirected edge, then node j also recognizes node i as its neighbor. A graph made of undirected edges is called an undirected graph. The adjacency matrix of an undirected graph is always symmetric.
- **Directed edge** An asymmetric connection from one node to another. Even if node i is connected to node j by a directed edge, the connection isn't necessarily reciprocated from node j to node i. A graph made of directed edges is called a directed graph. The adjacency matrix of a directed graph is generally asymmetric.
- **Unweighted edge** An edge without any weight value associated to it. There are only two possibilities between a pair of nodes in a network with unweighted edges; whether there is an edge between them or not. The adjacency matrix of such a network is made of only 0's and 1's.
- **Weighted edge** An edge with a weight value associated to it. A weight is usually given by a non-negative real number, which may represent a connection strength or distance between nodes, depending on the nature of the system being modeled. The definition of the adjacency matrix can be extended to contain those edge weight values for networks with weighted edges. The sum of the weights of edges connected to a node is often called the node strength, which corresponds to a node degree for unweighted graphs.
- **Multiple edges** Edges that share the same origin and destination. Such multiple edges connect two nodes more than once.
- **Self-loop** An edge that originates and ends at the same node.
- **Simple graph** A graph that doesn't contain directed, weighted, or multiple edges, or self-loops. Traditional graph theory mostly focuses on simple graphs.
- **Multigraph** A graph that may contain multiple edges. Many mathematicians also allow multigraphs to contain self-loops. Multigraphs can be undirected or directed.
- **Hyperedge** A generalized concept of an edge that can connect any number of nodes at once, not just two. A graph made of hyperedges is called a hypergraph (not covered in this textbook).

According to these taxonomies, all the examples shown in Fig. 15.2.2 are simple graphs. But many real-world networks can and should be modeled using directed, weighted, and/or multiple edges

? Exercise 15.2.5

Discuss which type of graph should be used to model each of the following networks. Make sure to consider (a) edge directedness, (b) presence of edge weights, (c) possibility of multiple edges/self-loops, (d) possibility of connections among three or more nodes, and so on.

- Family trees/genealogies
- Food webs among species
- Roads and intersections
- Web pages and links among them
- Friendships among people
- Marriages/sexual relationships
- College applications submitted by high school students
- Email exchanges among coworkers
- Social media (Facebook, Twitter, Instagram, etc.)

This page titled 15.2: Terminologies of Graph Theory is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.3: Constructing Network Models with NetworkX

Now that we have finished the above crash course on graph theoretic terminologies, it is time to begin with computational modeling of networks. As briefly previewed in Sections 5.4 and 12.2, there is a wonderful Python module called *NetworkX* [27] for network modeling and analysis. It is a free network analysis toolkit widely used by network researchers. If you are using Anaconda, NetworkX is already installed. If you are using Enthought Canopy, you can still easily install NetworkX by using its Package Manager. You can find the documentation for NetworkX online at `http://networkx.github.io` .

Perhaps it is worth mentioning why we are choosing NetworkX over other options. First, its functions are all written in Python, and their source codes are all available on the website above, so you can check the details of their algorithms and even modify the codes if you want. Second, NetworkX uses Python's plain "dictionaries" to store information about networks, which are fully accessible and flexible to store additional information you want to add. This property is particularly important when we implement simulation models of dynamical networks. Specialized data structures implemented in other network analysis tools may not have this flexibility.

So, let's first talk about the data structure used in NetworkX. There are the following four different data types (called "classes") that NetworkX offers:
Graph For undirected simple graphs (self-loops are allowed)
DiGraph For directed simple graphs (self-loops are allowed)
MultiGraph For undirected multigraphs (self-loops and multiple edges are allowed)
MultiDiGraph For directed multigraphs (self-loops and multiple edges are allowed)

You can choose one of these four data types that suits your modeling purposes. In this textbook, we will use `Graph` and `DiGraph` mostly.

You can construct a graph of your own manually. Here is an example:

Code 15.1: networkx-test.py

```
import networkx as nx

# creating a new empty Graph object
g = nx.Graph()

# adding a node named 'John'
g.add_node('John')

# adding a bunch of nodes at once
g.add_nodes_from(['Josh', 'Jane', 'Jess', 'Jack'])

# adding an edge between 'John' and 'Jane'
g.add_edge('John', 'Jane')
```

```
# adding a bunch of edges at once
g.add_edges_from([('Jess', 'Josh'), ('John', 'Jack'), ('Jack', 'Jane')])

# adding more edges
# undefined nodes will be created automatically
g.add_edges_from([('Jess', 'Jill'), ('Jill', 'Jeff'), ('Jeff', 'Jane')])

# removing the edge between 'John' and 'Jane'
g.remove_edge('John', 'Jane')

# removing the node 'John'
# all edges connected to that node will be removed too
g.remove_node('John')
```

Here I used strings for the names of the nodes, but a node's name can be a number, a string, a list, or any "hashable" object in Python. For example, in the phase space visualization code in Section 5.4, we used a tuple for a name of each node.

I believe each command used above to add or remove nodes/edges is quite self-explanatory. If not, you can always look at NetworkX's online documentation to learn more about how each command works. There are also several other ways to manipulate a graph object, which are not detailed here.

Once this code is executed, you can see the results in the Python command line, like this:

Code 15.2:

```
>>> g
<networkx.classes.graph.Graph object at 0x0000000002999860>
>>> g.nodes()
['Jeff', 'Josh', 'Jess', 'Jill', 'Jack', 'Jane']
>>> g.edges()
[('Jeff', 'Jane'), ('Jeff', 'Jill'), ('Josh', 'Jess'), ('Jess', 'Jill'),
('Jack', 'Jane')]
>>>
```

The first output ("< `networkx.classes` . ... >") shows that `g` is a `Graph` object. In order to see the contents of the data in the object, we need to use some commands. `g.nodes()` returns a list of all nodes in the network, while `g.edges()` returns a list of all edges. Compare these results with the commands we used above to check if the node/edge additions/removals were conducted correctly.

Those `nodes()` and `edges()` are functions that read the raw data inside the `Graph` object g and then produce a cleaned-up list of nodes or edges. But NetworkX also allows you to have direct access to the Graph object's internal raw data structure. The

data about nodes are stored in a dictionary called `node` right under `g` :

Code 15.3:

```
>>> g.node
{'Jeff': {}, 'Josh': {}, 'Jess': {}, 'Jill': {}, 'Jack': {}, 'Jane': {}}
```

This is a dictionary whose keys and values are the nodes' names and their properties, respectively. The properties of each node are also stored in a dictionary (initially they are all empty, as shown above), so you can dynamically add or modify any node property as follows:

Code 15.4:

```
>>> g.node['Jeff']['job'] = 'student'
>>> g.node['Jeff']['age'] = 20
>>> g.node['Jeff']
{'job': 'student', 'age': 20}
```

We will use this method to add dynamic states to the nodes in Section 16.2.

Similarly, the data about the edges are also stored in a dictionary called `edge` under g:

Code 15.5:

```
>>> g.edge
{'Jeff': {'Jane': {}, 'Jill': {}}, 'Josh': {'Jess': {}}, 'Jess': {'Jill':
{}, 'Josh': {}}, 'Jill': {'Jess': {}, 'Jeff': {}}, 'Jack': {'Jane': {}},
'Jane': {'Jack': {}, 'Jeff': {}}}
```

This is a little hard to read, so let me insert line breaks at the appropriate places so that the structure of the data is clearer:

Code 15.6:

```
{
    'Jeff': {'Jane': {}, 'Jill': {}},
    'Josh': {'Jess': {}},

    'Jess': {'Jill': {}, 'Josh': {}},
    'Jill': {'Jess': {}, 'Jeff': {}},
    'Jack': {'Jane': {}},
    'Jane': {'Jack': {}, 'Jeff': {}}
}
```

Now its structure is much clearer. `g.edge` is a dictionary that describes edges in the network in the `adjacency list` format. Each entry of `g.edge` is a pair of the node's name and a dictionary that contains its neighbors. For example, the first entry says that Jeff is connected to Jane and Jill. Note that the connections are perfectly symmetric; if Jeff is connected to Jane, then the entry for Jane (in the last line) also contains Jeff as her neighbor. Such symmetry is automatically maintained by NetworkX, because this `Graph` object is for undirected graphs.

Moreover, each neighbor's name works as a key associated with another dictionary in which we can store properties of the connection (initially they are all empty, as shown above). So you can dynamically add or modify any edge property as follows:

Code 15.7:

```
>>> g.edge['Jeff']['Jane']['trust'] = 1.0
>>> g.edge['Josh']['Jess']['love'] = True
>>> g.edge['Jess']['Josh']['love']
True
>>> g.edge
{
    'Jeff': {'Jane': {'trust': 1.0}, 'Jill': {}},
    'Josh': {'Jess': {'love': True}},
    'Jess': {'Jill': {}, 'Josh': {'love': True}},
    'Jill': {'Jess': {}, 'Jeff': {}},
    'Jack': {'Jane': {}},
    'Jane': {'Jack': {}, 'Jeff': {'trust': 1.0}}
}
```

Again, the output is reformatted to enhance the readability. Note that the new edge properties (' `trust` ' between Jeff and Jane; ' `love` ' between Josh and Jess) are correctly inserted into the respective dictionaries, always maintaining symmetry between nodes. I just added `True` ' `love` ' from Josh to Jess, which has been reciprocated from Jess to Josh! What a beautiful world. This is all because we are using a `Graph` object.

The `DiGraph` object behaves differently. It is designed to represent directed graphs, so the edges are all considered one-way. Symmetries are not maintained automatically, as illustrated in the following example:

Code 15.8:

```
>>> g = nx.DiGraph()
>>> g.add_edge('Josh', 'Jess')
>>> g.edge['Josh']['Jess']['love'] = True
>>> g.edge
{'Jess': {}, 'Josh': {'Jess': {'love': True}}}
>>> g.edge['Jess']['Josh']['love']

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    g.edge['Jess']['Josh']['love']
KeyError: 'Josh'
>>>
```

The last error message means that Jess doesn't know a guy named Josh in this case, because the graph is asymmetric. Life is hard.

? Exercise 15.3.1

Represent the network shown in Exercise 15.1 as a `Graph` object of NetworkX.

? Exercise 15.3.2

Represent a small social network around you (say, 10 people) as either `Graph` or `DiGraph` object of NetworkX. Then add properties to nodes and edges, such as:

- Node properties: full name, age, job, address, etc.
- Edge properties: relationship, connection weight, etc.

In the examples above, we manually constructed network models by adding or removing nodes and edges. But NetworkX also has some built-in functions that can generate networks of specific shapes more easily. Here are a few examples:

Code 15.9: networkx-test2.py

```
import networkx as nx


# complete graph made of 5 nodes
g1 = nx.complete_graph(5)

# complete (fully connected) bipartite graph
# made of group of 3 nodes and group of 4 nodes
g2 = nx.complete_bipartite_graph(3, 4)

# Zachary's Karate Club graph
g3 = nx.karate_club_graph()
```

The last example (*Zachary's Karate Club graph*) is a famous classic example of social networks reported by Wayne Zachary in the 1970s [59]. It is a network of friendships among 34 members of a karate club at a U.S. university. The edge lists of these examples are as follows:

Code 15.10:

```
>>> g1.edges()
[(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(3, 4)]
>>> g2.edges()
[(0, 3), (0, 4), (0, 5), (0, 6), (1, 3), (1, 4), (1, 5), (1, 6), (2, 3),
(2, 4), (2, 5), (2, 6)]
>>> g3.edges()
[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 10),
(0, 11), (0, 12), (0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2),
(1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 21), (1, 30), (2, 3),
(2, 32), (2, 7), (2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (3, 7),
(3, 12), (3, 13), (4, 10), (4, 6), (5, 16), (5, 10), (5, 6), (6, 16),
(8, 32), (8, 30), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33),
(15, 32), (15, 33), (18, 32), (18, 33), (19, 33), (20, 32), (20, 33),
(22, 32), (22, 33), (23, 32), (23, 25), (23, 27), (23, 29), (23, 33),
(24, 25), (24, 27), (24, 31), (25, 31), (26, 33), (26, 29), (27, 33),
(28, 33), (28, 31), (29, 32), (29, 33), (30, 33), (30, 32), (31, 33),
(31, 32), (32, 33)]
```

? Exercise 15.3.3

Construct a graph by generating a complete graph made of 10 nodes, and then connect a new additional node to each of the 10 nodes, using one edge each.

? Exercise 15.3.4

Create Zachary's Karate Club graph using NetworkX's built-in function, and inspect its nodes and edges to see if they have any non-topological properties.

This page titled 15.3: Constructing Network Models with NetworkX is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.4: Visualizing Networks with NetworkX

NetworkX also provides functions for visualizing networks. They are not as powerful as other more specialized software[1], but still quite handy and useful, especially for small- to mid-sized network visualization. Those visualization functions depend on the functions defined in matplotlib (pylab), so we need to import it before visualizing networks.

The simplest way is to use NetworkX's `draw` function:

Code 15.11: karate-club-visualization.py

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()
nx.draw(g)
show()
```

The result is shown in Fig. 15.4.1. Red circles represent the nodes, and the black lines connecting them represent the edges. By default, the layout of the nodes and edges is automatically determined by the *Fruchterman-Reingold* force-directed algorithm [62] (called "spring layout" in NetworkX), which conducts a pseudo-physics simulation of the movements of the nodes, assuming that each edge is a spring with a fixed equilibrium distance. This heuristic algorithm tends to bring groups of well-connected nodes closer to each other, making the result of visualization more meaningful and aesthetically more pleasing.

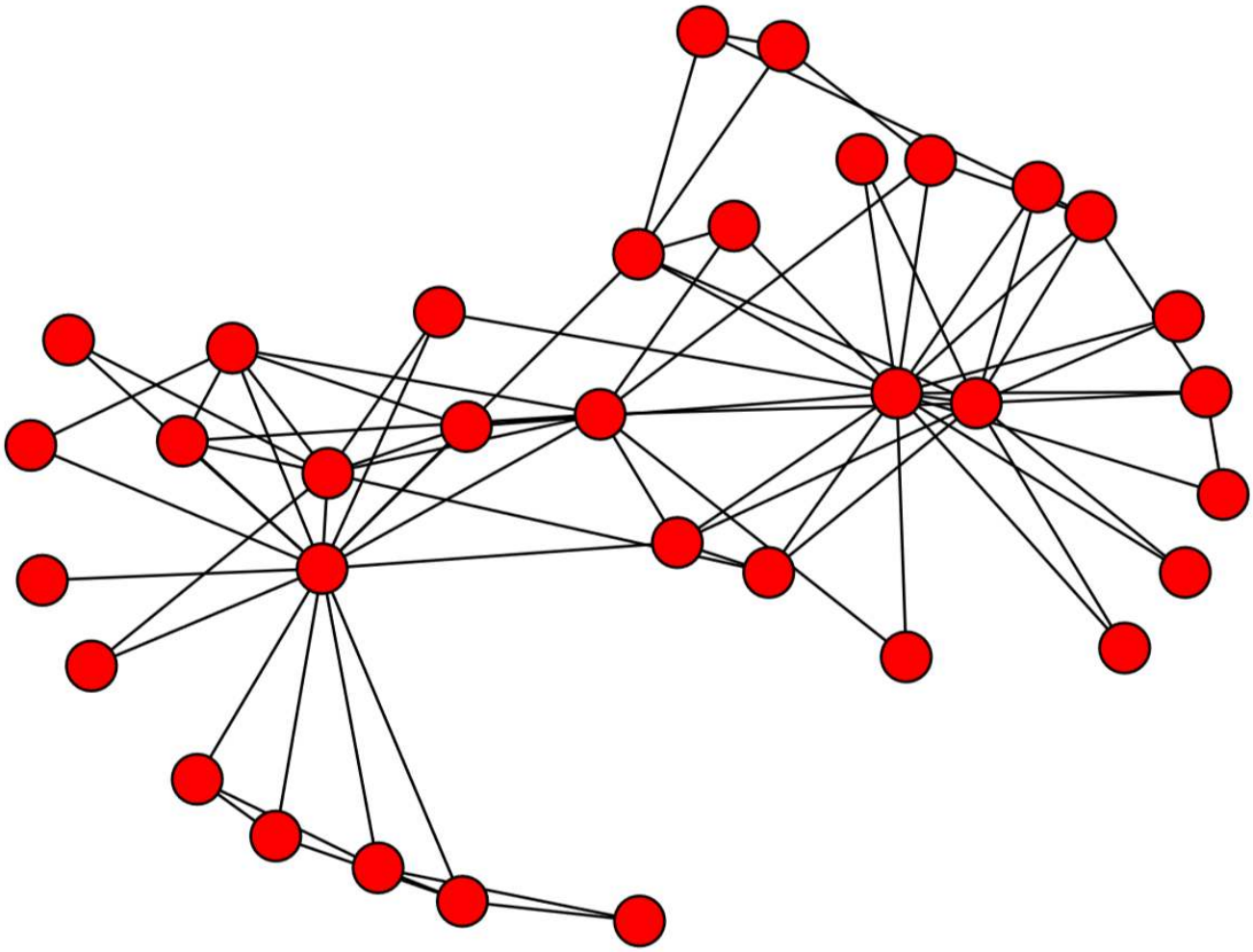


Figure **15.4.1**: Visual output of Code 15.11. Zachary's Karate Club graph is visualized. Your result may not look like this because the spring layout algorithm uses random initial positions by default.

There are several other layout algorithms, as shown in Code 15.12 and Fig.15.5.1. Also, there are many options you can use to customize visualization results (see Code 15.13 and Fig. 15.4.3). Check out NetworkX's online documentation to learn more about what you can do.

Code 15.12: network-layouts.py

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()

subplot(2, 2, 1)
nx.draw_random(g)
title('random layout')

subplot(2, 2, 2)
nx.draw_circular(g)
title('circular layout')

subplot(2, 2, 3)
nx.draw_spectral(g)
title('spectral layout')

subplot(2, 2, 4)
shells = [[0, 1, 2, 32, 33],
          [3, 5, 6, 7, 8, 13, 23, 27, 29, 30, 31],
          [4, 9, 10, 11, 12, 14, 15, 16, 17, 18,
           19, 20, 21, 22, 24, 25, 26, 28]]
nx.draw_shell(g, nlist = shells)
title('shell layout')

show()
```

Code 15.13: network-drawing-options.py

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()
positions = nx.spring_layout(g)

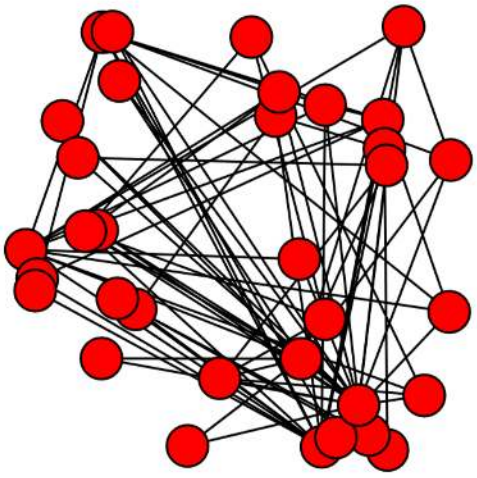
subplot(3, 2, 1)
nx.draw(g, positions, with_labels = True)
title('showing node names')

subplot(3, 2, 2)
nx.draw(g, positions, node_shape = '>')
title('using different node shape')
```

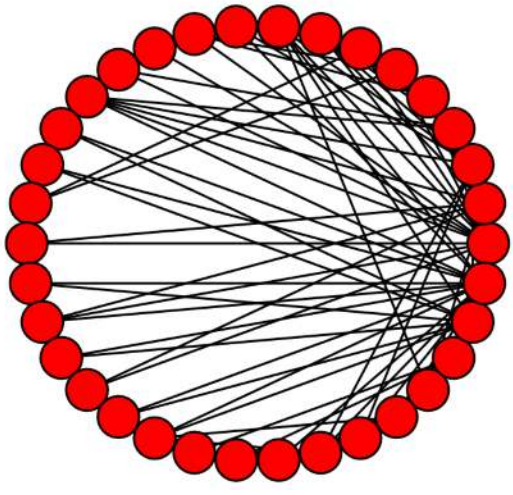
```
subplot(3, 2, 3)
nx.draw(g, positions,
        node_size = [g.degree(i) * 50 for i in g.nodes()])
title('changing node sizes')

subplot(3, 2, 4)
nx.draw(g, positions, edge_color = 'pink',
```

random layout



circular layout



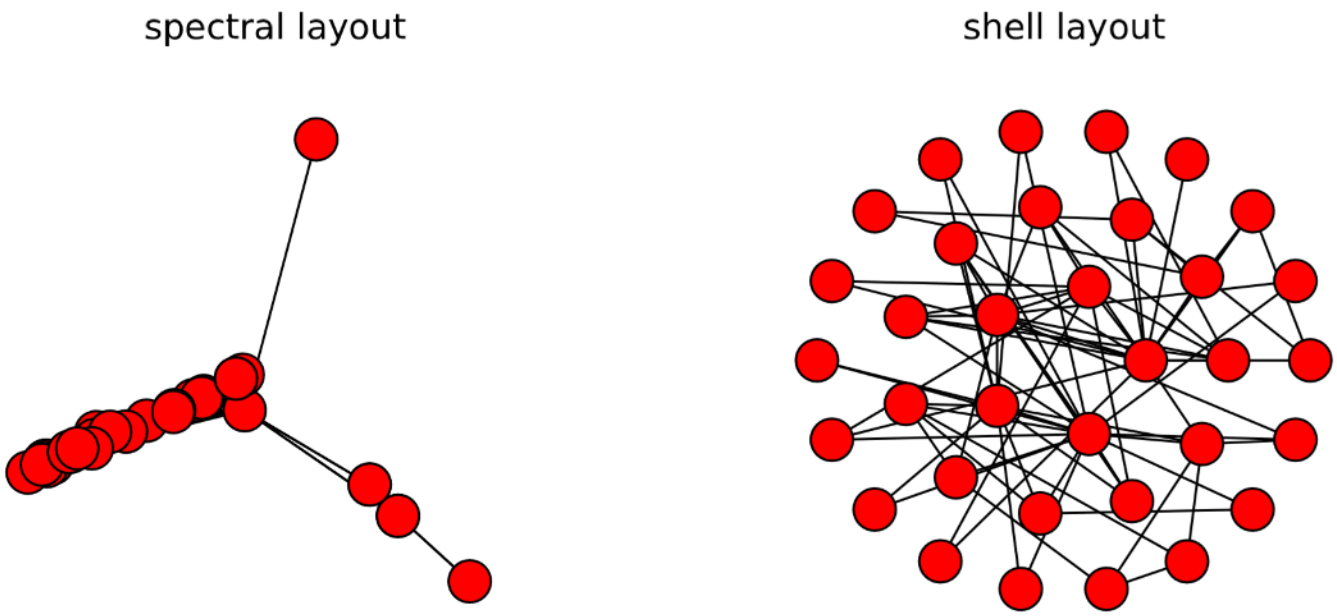


Figure **15.4.2**: Visual output of Code 15.12, showing examples of network layouts available in NetworkX.

```
        node_color = ['yellow' if i < 17 else 'green' for i in g.nodes()])
title('coloring nodes and edges')

subplot(3, 2, 5)
nx.draw_networkx_nodes(g, positions)
title('nodes only')

subplot(3, 2, 6)
nx.draw_networkx_edges(g, positions)
title('edges only')

show()
```

? Exercise 15.4.1

Visualize the following graphs. Look them up in NetworkX's online documentation to learn how to generate them.

- A "hypercube graph" of four dimensions.
- A "ladder graph" of length 5.
- A "barbell graph" made of two 20-node complete graphs that are connected by a single edge.
- A "wheel graph" made of 100 nodes.

? Exercise 15.4.2

Visualize the social network you created in Exercise 15.3.2. Try several options to customize the result to your preference.

[1]For example, Gephi (http://gephi.github.io/) can handle much larger networks, and it has many more node-embedding algorithms than NetworkX.

This page titled 15.4: Visualizing Networks with NetworkX is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.5: Importing/Exporting Network Data

In most network studies, researchers need to model and analyze networks that exist in the real world. To do so, we need to learn how to import (and export) network data from outside Python/NetworkX. Fortunately, NetworkX can read and write network data from/to files in a number of formats[2].

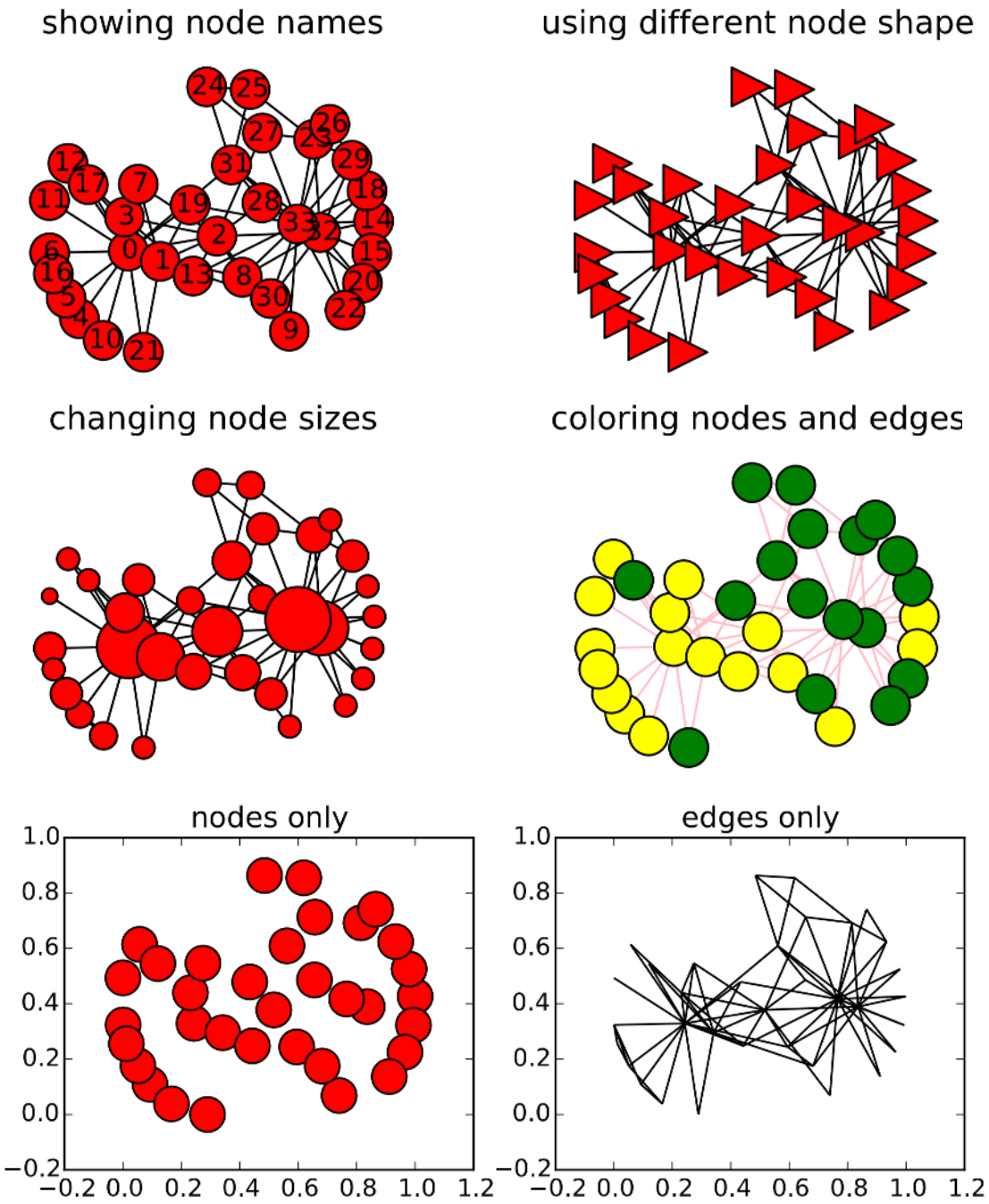


Figure 15.5.1: Visual output of Code 15.13, showing examples of drawing options available in NetworkX. The same node positions are used in all panels. The last two examples show the axes because they are generated using different drawing functions. To suppress the axes, use axis(' `off` ') right after the network drawing.

Let's work on a simple example. You can create your own adjacency list in a regular spreadsheet (such as Microsoft Excel or Google Spreadsheet) and then save it in a "csv" format file (Fig. 15.5.2). In this example, the adjacency list says that John is connected to Jane and Jack; Jess is connected to Josh and Jill; and so on.

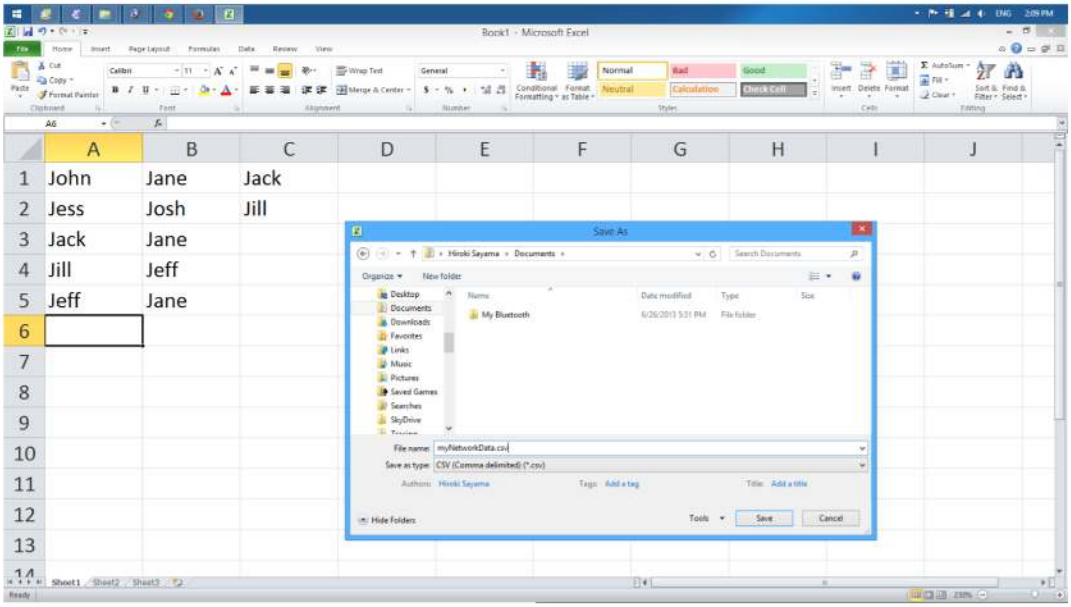


Figure **15.5.2**: Creating an adjacency list in a spreadsheet.

You can read this file by using the `read_adjlist` command, as follows:

Code 15.14: read-adjlist.py

```
from pylab import *
import networkx as nx

g = nx.read_adjlist('myNetworkData.csv', delimiter = ',')
nx.draw(g, with_labels = True)
show()
```

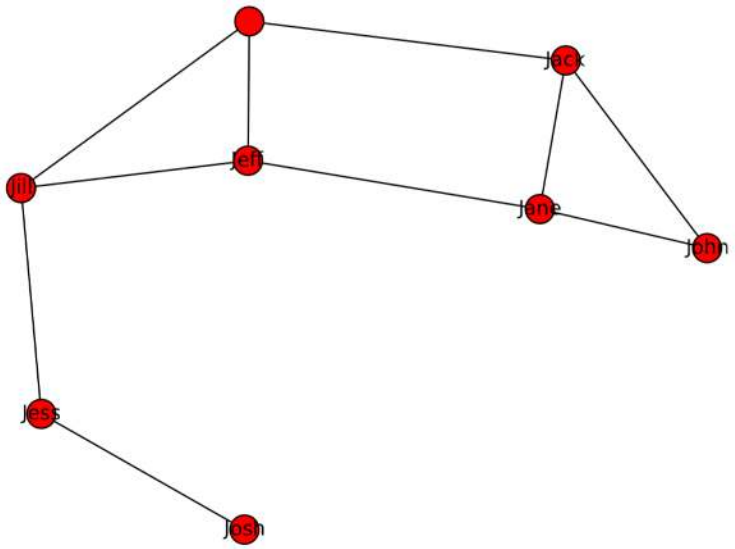


Figure **15.5.3**: Visual output of Code 15.14.

Looks good. No—wait—there is a problem here. For some reason, there is an extra node without a name (at the top of the network in Fig.15.5.3)! What happened? The reason becomes clear if you open the CSV file in a text editor, which reveals that the file actually looks like this:

Code 15.15:

```
John,Jane,Jack
Jess,Josh,Jill
Jack,Jane,
Jill,Jeff,
Jeff,Jane,
```

Note the commas at the end of the third, fourth, and fifth lines. Many spreadsheet applications tend to insert these extra commas in order to make the number of columns equal for all rows. This is why NetworkX thought there would be another node whose name was “” (blank). You can delete those unnecessary commas in the text editor and save the file. Or you could modify the visualization code to remove the nameless node before drawing the network. Either way, the updated result is shown in Fig. 15.5.4.

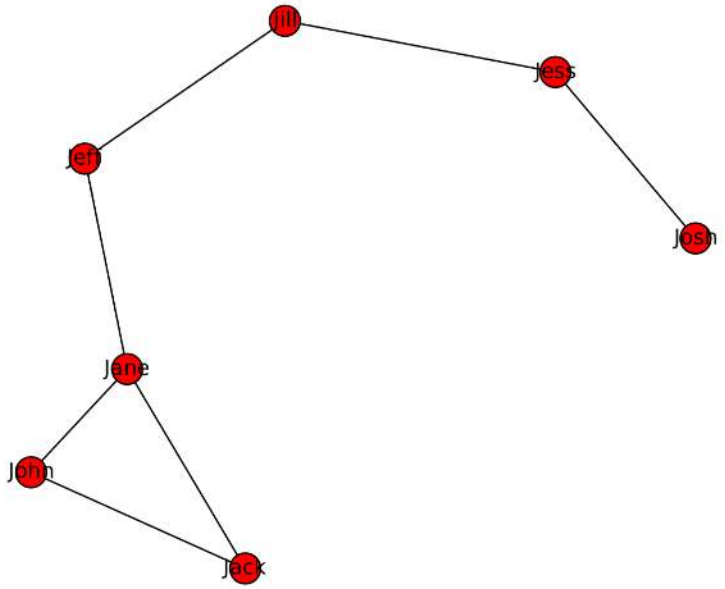


Figure **15.5.4** Visual output of Code 15.14 with a corrected CSV data file.

If you want to import the data as a directed graph, you can use the `create_using` option in the read_adjlist command, as follows:

Code 15.16: read-adjlist-directed.py

```
from pylab import *
import networkx as nx

g = nx.read_adjlist('myNetworkData.csv', delimiter = ',',
                    create_using = nx.DiGraph())
nx.draw(g, with_labels = True)
show()
```

The result is shown in Fig.15.5.5, in which arrowheads are indicated by thick line segments.

Another simple function for data importing is `read_edgelist` . This function reads an edge list, i.e., a list of pairs of nodes, formatted as follows:

Code 15.17:

```
John Jane
John Jack
Jess Josh
Jess Jill
Jack Jane
```

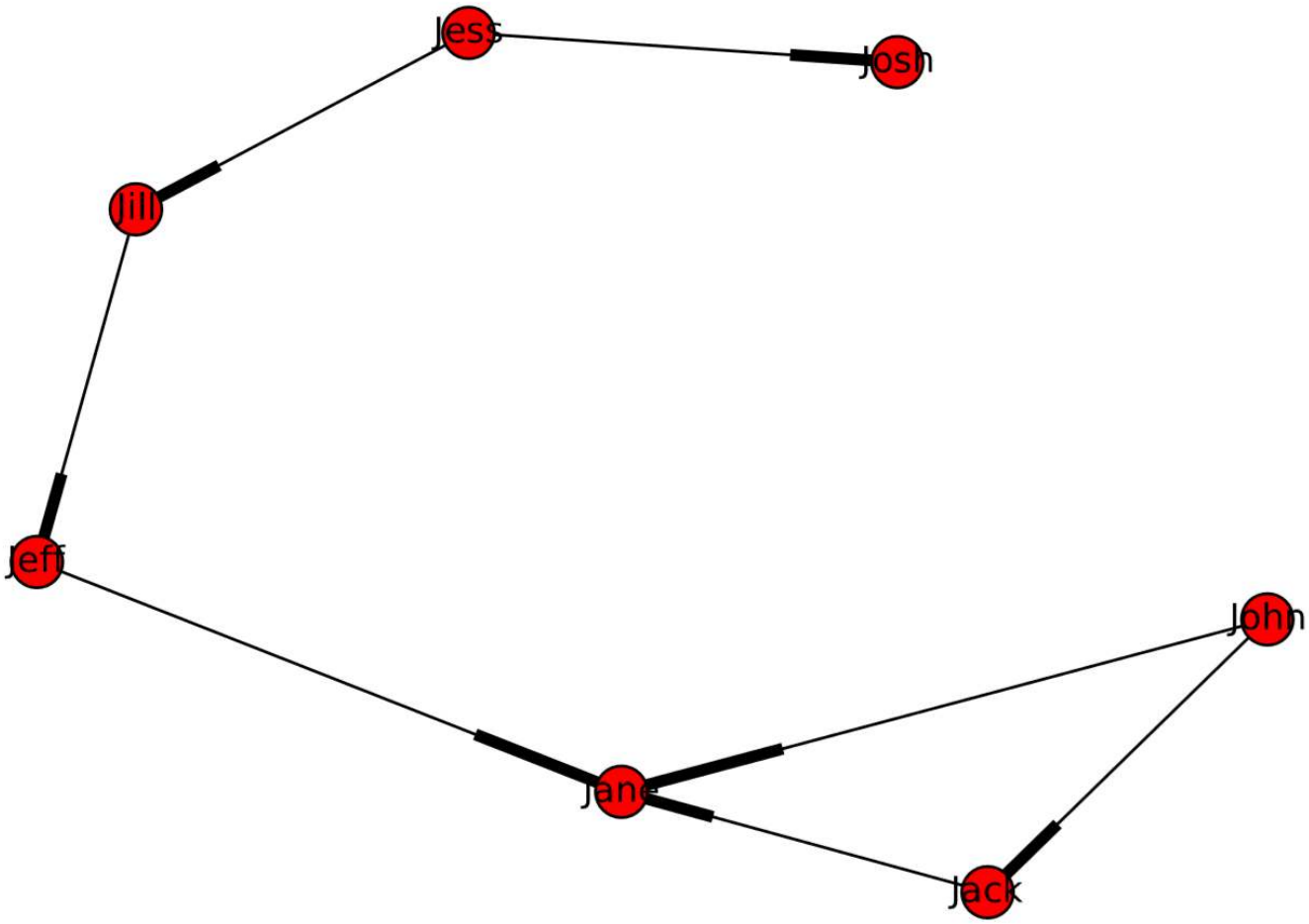


Figure **15.5.5**: Visual output of Code 15.16.

One useful feature of `read_edgelist` is that it can import not only edges but also their properties, such as:

Code 15.18:

```
John Jane {'weight': 0.8}
John Jack {'weight': 0.8, 'trust': 1.5}
Jess Josh {'weight': 0.2}
Jess Jill {'trust': 2.0}
Jack Jane {'weight': 1.0}
```

The third column of this data file is structured as a Python dictionary, which will be imported as the property of the edge.

? Exercise 15.5.1

Create a data file of the social network you created in Exercise 15.3.2 using a spreadsheet application, and then read the file and visualize it using NetworkX. You can use either the adjacency list format or the edge list format.

? Exercise 15.5.2

Import network data of your choice from Mark Newman's Network Data website: `http://www-personal.umich.edu/~mejn/netdata/`. The data on this website are all in GML format, so you need to figure out how to import them into NetworkX. Then visualize the imported network.

Finally, NetworkX also has functions to write network data files from its graph objects, such as `write_adjlist` and `write_edgelist`. Their usage is very straightforward. Here is an example:

Code 15.19:

```
import networkx as nx
g = nx.complete_graph(5)
nx.write_adjlist(g, 'complete-graph.txt')
```

Then a new text file named '`complete-graph.txt`' will appear in the same folder, which looks like this:

Code 15.20:

```
#C:/Users/Hiroki/Desktop/test.py
# GMT Sat Nov 22 20:24:00 2014
# complete_graph(5)
0 1 2 3 4
1 2 3 4
2 3 4
3 4
4
```

Note that the adjacency list is optimized so that there is no redundant information included in this output. For example, node 4 is connected to 0, 1, 2, and 3, but this information is not included in the last line because it was already represented in the preceding lines.

? Exercise 15.5.3

Write the network data of Zachary's Karate Club graph into a file in each of the following formats:

- Adjacency list
- Edge list

[2]For more details, see `https://networkx.github.io/documenta...ence/readwrite.html`.

[3]If you can't read the data file or get the correct result, it may be because your NetworkX did not recognize operating system-specific newline characters in the file (this may occur particularly for Mac users). You can avoid this issue by saving the CSV file in a different mode (e.g., "MS-DOS" mode in Microsoft Excel).

This page titled 15.5: Importing/Exporting Network Data is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

15.6: Generating Random Graphs

So far, we have been creating networks using deterministic methods, e.g., manually adding nodes and edges, importing data files, etc. In the meantime, there are some occasions where you want to have a randomly generated network. Here are some examples of NetworkX's built-in functions that can generate random graph samples:

Code 15.21: random-graphs.py

```
from pylab import *
import networkx as nx

subplot(2, 2, 1)
nx.draw(nx.gnm_random_graph(10, 20))
title('random graph with\n10 nodes, 20 edges')

subplot(2, 2, 2)
nx.draw(nx.gnp_random_graph(20, 0.1))
title('random graph with\n 20 nodes, 10% edge probability')

subplot(2, 2, 3)
nx.draw(nx.random_regular_graph(3, 10))
title('random regular graph with\n10 nodes of degree 3')
```

```
subplot(2, 2, 4)
nx.draw(nx.random_degree_sequence_graph([3,3,3,3,4,4,4,4,5,5]))
title('random graph with\n degree sequence\n[3,3,3,3,4,4,4,4,5,5]')

show()
```

The output is shown in Fig. 15.10. The first example, `gnm_random_graph(n, m)`, simply generates a random graph made of `n` nodes and `m` edges. The second example, `gnp_random_graph(n, p)`, generates a random graph made of `n` nodes, where each node pair is connected to each other with probability `p`. These two types of random graphs are called *Erd˝os-R´enyi random graphs* after two Hungarian mathematicians Paul Erd˝os and Alfr´ed R´enyi who studied random graphs in the 1950s [63]. The third example, `random_regular_graph(k, n)`, generates a *random regular graph* made of `n` nodes that all have the same degree `k`. The fourth example, `random_degree_sequence_graph(list)`, generates a random graph whose nodes have degrees specified in `list`. In this particular example, four nodes have degree 3, four nodes have degree 4, and two nodes have degree 5. Note that this function may not be able to generate a graph within a given number of trials (which is set to 10 by default). You can increase the number of trials by specifying the `tries` option in the function, e.g., `tries = 100`.

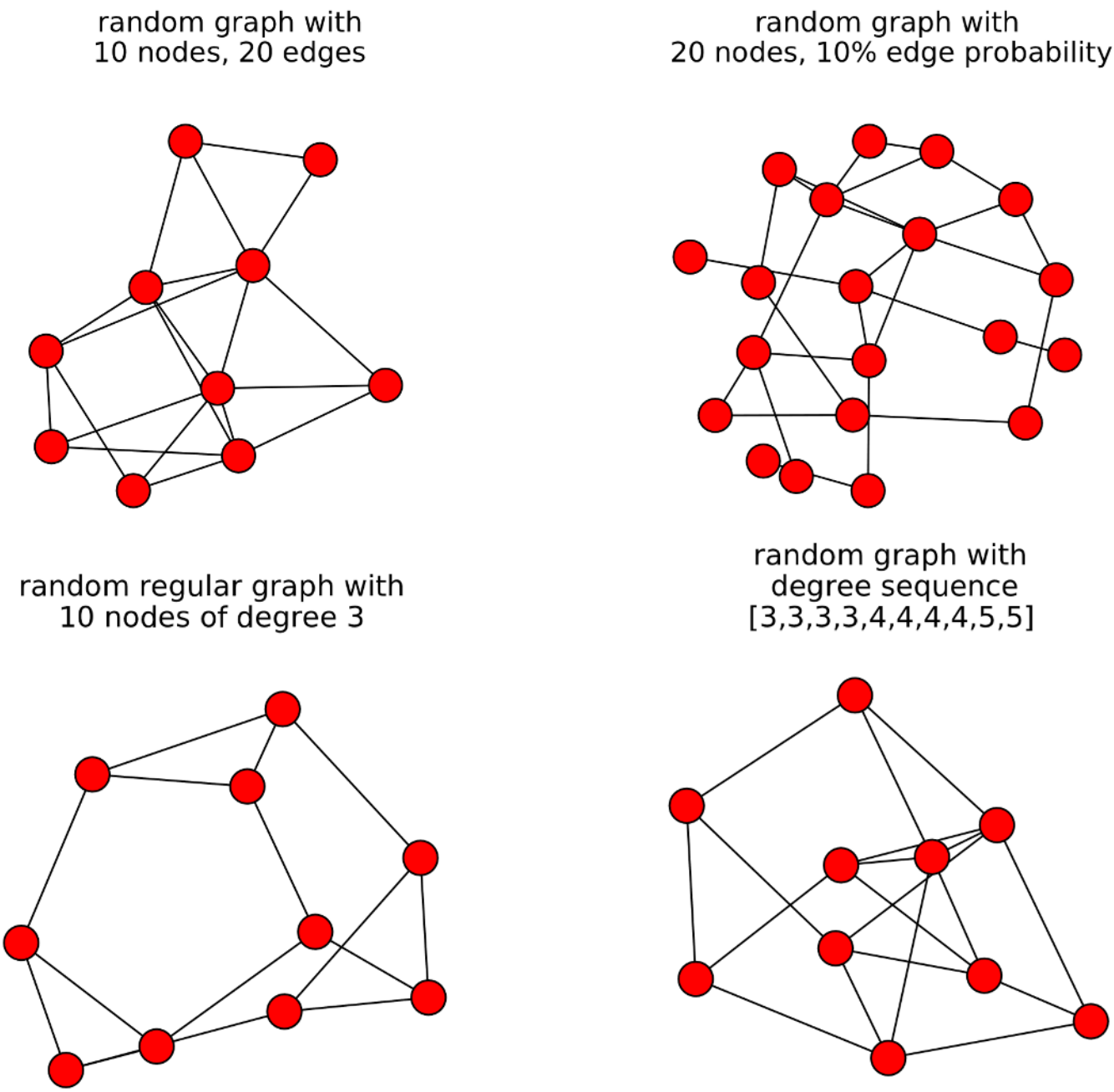


Figure **15.6.1**: Visual output of Code 15.21.

Random graphs can also be generated by randomizing existing graph topologies. Such *network randomization* is particularly useful when you want to compare a certain property of an observed network with that of a randomized "null" model. For example, imagine you found that information spreading was very slow in the network you were studying. Then you could test whether or not the network's specific topology caused this slowness by simulating information spreading on both your network and on its randomized counterparts, and then comparing the speed of information spreading between them. There are several different ways to randomize network topologies. They differ regarding which network properties are to be conserved during randomization. Some examples are shown below (here we assume that `g` is the original network to be randomized):

- `gnm_random_graph(g.number_of_nodes(), g.number_of_edges())` conserves the numbers of nodes and edges, nothing else.
- `random_degree_sequence_graph(g.degree().values())` conserves the sequence of node degrees, in addition to the numbers of nodes and edges. As noted above, this function may not generate a graph within a given number

of trials.

- `expected_degree_graph(g.degree().values())` conserves the number of nodes and the sequence of "expected" node degrees. This function can always generate a graph very quickly using the efficient Miller-Hagberg algorithm[64]. While the generated network's degree sequence doesn't exactly match that of the input (especially if the network is dense), this method is often a practical, reasonable approach for the randomization of large real-world networks.
- `double_edge_swap(g)` conserves the numbers of nodes and edges and the sequence of node degrees. Unlike the previous three functions, this is not a graph-generating function, but it directly modifies the topology of graph `g`. It conducts what is called a *double edge swap*, i.e., randomly selecting two edges `a − b` and `c − d`, removing those edges, and then creating new edges `a−c` and `b−d` (if either pair is already connected, another swap will be attempted). This operation realizes a slight randomization of the topology of `g`, and by repeating it many times, you can gradually randomize the topology from the original g to a completely randomized network. `double_edge_swap(g, k)` conducts `k` double edge swaps. There is also `connected_double_edge_swap` available, which guarantees that the resulting graph remains connected.

? Exercise 15.6.1

RandomizethetopologyofZachary'sKarateClubgraphbyusing each of the following functions and visualize the results:

- `gnm_random_graph`
- `random_degree_sequence_graph`
- `expected_degree_graph`
- `double_edge_swap` (5 times)
- `double_edge_swap` (50 times)

This page titled 15.6: Generating Random Graphs is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

16: Dynamical Networks I - Modeling

16.1: Dynamical Network Models
16.2: Simulating Dynamics on Networks
16.3: Simulating Dynamics of Networks
16.4: Simulating Adaptive Networks

This page titled 16: Dynamical Networks I - Modeling is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

16.1: Dynamical Network Models

There are several different classes of dynamical network models. In this chapter, we will discuss the following three, in this particular order:

- **Models for "dynamics *on* networks"** These models are the most natural extension of traditional dynamical systems models. They consider how the states of components, or nodes, change over time through their interactions with other nodes that are connected to them. The connections are represented by links of a network, where the network topology is fixed throughout time. Cellular automata, Boolean networks, and artificial neural networks (without learning) all belong to this class.
- **Models for "dynamics *of* networks"** These are the models that consider dynamical changes of network topology itself over time, for various purposes: to understand mechanisms that bring particular network topologies, to evaluate robustness and vulnerability of networks, to design procedures for improving certain properties *of* networks, etc. The dynamics of networks are a particularly hot topic in network science nowadays (as of 2015) because of the increasing availability of *temporal network* data [65].
- **Models for "adaptive networks"** I must admit that I am not 100% objective when it comes to this class of models, because I am one of the researchers who have been actively promoting it [66]. Anyway, the adaptive network models are models that describe the co-evolution of dynamics *on* and *of* networks, where node states and network topologies dynamically change adaptively to each other. Adaptive network models try to unify different dynamical network models to provide a generalized modeling framework for complex systems, since many real-world systems show such adaptive network behaviors [67].

This page titled 16.1: Dynamical Network Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

16.2: Simulating Dynamics on Networks

Because NetworkX adopts plain dictionaries as their main data structure, we can easily add states to nodes (and edges) and dynamically update those states iteratively. This is a simulation of dynamics *on* networks. This class of dynamical network models describes dynamic state changes taking place on a static network topology. Many real-world dynamical networks fall into this category, including:

- Regulatory relationships among genes and proteins within a cell, where nodes are genes and/or proteins and the node states are their expression levels.
- Ecological interactions among species in an ecosystem, where nodes are species and the node states are their populations.
- Disease infection on social networks, where nodes are individuals and the node states are their epidemiological states (e.g., susceptible, infected, recovered, immunized, etc.).
- Information/culture propagation on organizational/social networks, where nodes are individuals or communities and the node states are their informational/cultural states.

The implementation of simulation models for dynamics on networks is strikingly similar to that of CA. You may find it even easier on networks, because of the straightforward definition of "neighbors" on networks. Here, we will work on a simple local *majority rule* on a social network, with the following assumptions:

- Nodes represent individuals, and edges represent their symmetric connections for information sharing
- Each individual takes either 0 or 1 as his or her state.
- Each individual changes his or her state to a majority choice within his or her local neighborhood (i.e., the individual him- or herself and the neighbors connected to him or her). This neighborhood is also called the ego network of the focal individual in social sciences.
- State updating takes place simultaneously on all individuals in the network.
- Individuals' states are initially random.

Let's continue to use pycxsimulator.py and implement the simulator code by defining three essential components—initialization, observation, and updating.

The initialization part is to create a model of social network and then assign random states to all the nodes. Here I propose to perform the simulation on our favorite Karate Club graph. Just like the CA simulation, we need to prepare two network objects, one for the current time step and the other for the next time step, to avoid conflicts during the state updating process. Here is an example of the initialization part:

Code 16.1:

```
def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0
    nextg = g.copy()
```

Here, we pre-calculate node positions using `spring_layout` and store the results under `g` as an attribute `g.pos`. Python is so flexible that you can dynamically add any attribute to an object without declaring it beforehand; we will utilize this trick later in the agent-based modeling as well. `g.pos` will be used in the visualization so that nodes won't jump around every time you draw the network.

The `g.nodes_iter()` command used in the `for` loop above works essentially the same way as `g.nodes()` in this context, but it is much more memory efficient because it doesn't generate an actual fully spelled-out list, but instead it generates a much more compact representation called an *iterator*, an object that can return next values iteratively. There is also `g.edges_iter()` for similar purposes for edges. It is generally a good idea to use `nodes_iter()` and `edges_iter()` in loops, especially if your network is large. Finally, in the last line, the copy command is used to create a duplicate copy of the network.

The observation part is about drawing the network. We have already done this many times, but there is a new challenge here: We need to visualize the node states in addition to the network topology. We can use the `node_color` option for this purpose:

Code 16.2:

```
def observe():

    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)
```

The `vmin/vmax` options are to use a fixed range of state values; otherwise matplotlib would automatically adjust the color mappings, which sometimes causes misleading visualization. Also note that we are using `g.pos` to keep the nodes in pre-calculated positions.

The updating part is pretty similar to what we did for CA. You just need to sweep all the nodes, and for each node, you sweep its neighbors, counting how many 1's you have in the local neighborhood. Here is a sample code:

Code 16.3:

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        count = g.node[i]['state']
        for j in g.neighbors(i):
            count += g.node[j]['state']
        ratio = count / (g.degree(i) + 1.0)
        nextg.node[i]['state'] = 1 if ratio > .5 \
                                 else 0 if ratio < .5 \
                                 else 1 if random() < .5 else 0
    g, nextg = nextg, g
```

I believe this part deserves some in-depth explanation. The first `for` loop for `i` is to sweep the space, i.e., the whole network. For each node, `i`, the variable `count` is first initialized with node `i`'s own state. This is because, unlike in the previous CA implementations, the node `i` itself is not included in its neighbors, so we have to manually count it first. The second `for` loop for `j` is to sweep node `i`'s neighbors. NetworkX's `g.neighbors(i)` function gives a list of `i`'s neighbors, which is a lot simpler than the neighborhood sweep in CA; we don't have to write nested `for` loops for `dx` and `dy`, and we don't have to worry about boundary conditions at all. Neighbors are `neighbors`, period.

Once the local neighborhood sweep is done, the state ratio is calculated by dividing `count` by the number of nodes counted (= the degree of node `i` plus 1 for itself). If the state ratio is above 0.5, the local majority is 1, so the next state of node `i` will be 1. If it is below 0.5, the local majority is 0, so the next state will be 0. Moreover, since this dynamic is taking place on a network, there is a chance for a tie (which never occurs on CA with von Neumann or Moore neighborhoods). In the example above, I just included a tie-breaker coin toss, to be fair. And the last swap of `g` and `nextg` is something we are already familiar with.

The entire simulation code looks like this:

Code 16.4: net-majority.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0
```

```
    nextg = g.copy()

def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)

def update():
    global g, nextg
    for i in g.nodes_iter():
        count = g.node[i]['state']
```

```
        for j in g.neighbors(i):
            count += g.node[j]['state']
        ratio = count / (g.degree(i) + 1.0)
        nextg.node[i]['state'] = 1 if ratio > .5 \
                                 else 0 if ratio < .5 \
                                 else 1 if random() < .5 else 0
```

```
    g, nextg = nextg, g

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

ParseError: invalid XmlAttribute (click for details)

Run this code and enjoy your first dynamical network simulation. You will notice that the network sometimes converges to a homogeneous state, while at other times it remains in a divided condition (Fig. 16.2.1). Can you identify the areas where the boundaries between the different states tend to form?

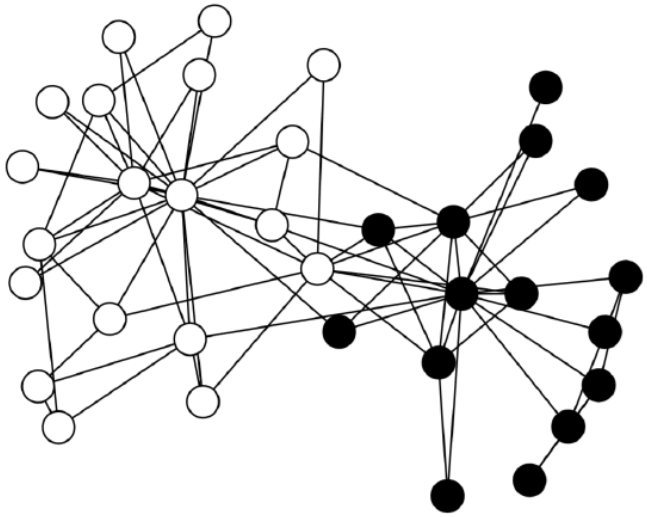


Figure **16.2.1**: Visual output of Code 16.4. In this example, two groups of different states have formed.

? Exercise 16.2.1

Revise the majority rule dynamical network model developed above so that each node stochastically flips its state with some probability. Then simulate the model and see how its behavior is affected.

In what follows, we will see some more examples of two categories of dynamics on networks models: discrete state/time models and continuous state/time models. We will discuss two exemplar models in each category.

Discrete state/time models (1): Voter model

The first example is a revision of the majority rule dynamical network model developed above. A very similar model of abstract opinion dynamics has been studied in statistical physics, which is called the *voter model*. Just like in the previous model, each node ("voter") takes one of the finite discrete states (say, black and white, or red andblue—you could view it as a political opinion), but the updating rule is different. Instead of having all the nodes update their states simultaneously based on local majority choices, the voter model considers only one opinion transfer event at a time between a pair of connected nodes that are randomly chosen from the network (Fig. 16.2.2). In this sense, it is an *asynchronous dynamical* network model.

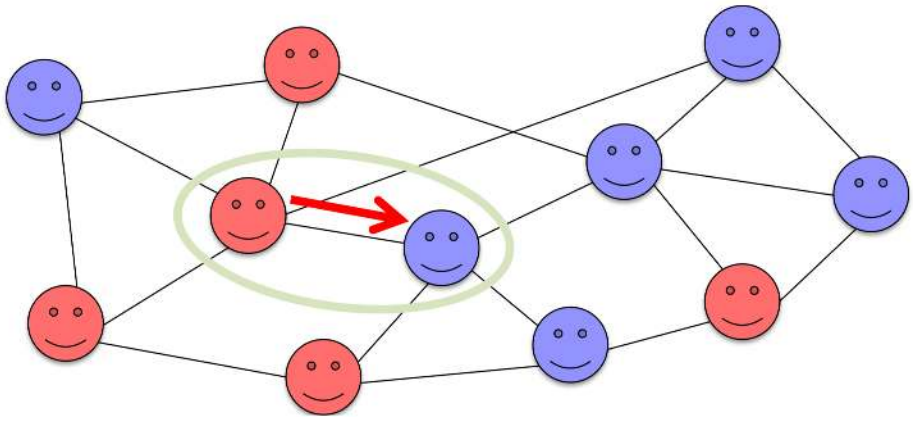


Figure **16.2.2**: Schematic illustration of the voter model. Each time a pair of connected nodes are randomly selected (light green circle), the state of the speaker node (left) is copied to the listener node (right).

There are three minor variations of how those nodes are chosen:

- *Original ("pull") version*: First, a "listener" node is randomly chosen from the network, and then a "speaker" node is randomly chosen from the listener's neighbors.

- *Reversed ("push") version*: First, a "speaker" node is randomly chosen from the network, and then a "listener" node is randomly chosen from the speaker's neighbors.
- *Edge-based (symmetric) version:* First, an edge is randomly chosen from the network, and then the two endpoints (nodes) of the edge are randomly assigned to be a "speaker" and a "listener."

In either case, the "listener" node copies the state of the "speaker" node. This is repeated a number of times until the entire network reaches a consensus with a homogenized state.

If you think through these model assumptions carefully, you may notice that the first two assumptions are not symmetric regarding the probability for a node to be chosen as a "speaker" or a "listener." The probability actually depends on how popular a node is. Specifically, the original version tends to choose higher-degree nodes as a speaker more often, while the reversed version tends to choose higher-degree nodes as a listener more often. This is because of the famous *friendship paradox*, a counter-intuitive fact first reported by sociologist Scott Feld in the 1990s [68], which is that a randomly selected neighbor of a randomly selected node tends to have a larger-than-average degree. Therefore, it is expected that the original version of the voter model would promote homogenization of opinions as it gives more speaking time to the popular nodes, while the reversed version would give an equal chance of speech to everyone so the opinion homogenization would be much slower, and the edge-based version would be somewhere in between. We can check these predictions by computer simulations.

The good news is that the simulation code of the voter model is much simpler than that of the majority rule network model, because of the asynchrony of state updating. We no longer need to use two separate data structures; we can keep modifying just one `Graph` object directly. Here is a sample code for the original "pull" version of the voter model, again on the Karate Club graph:

Code 16.5: voter-model.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx
import random as rd

def initialize():
    global g
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0

def observe():
    global g
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)


def update():
    global g
    listener = rd.choice(g.nodes())
    speaker = rd.choice(g.neighbors(listener))
    g.node[listener]['state'] = g.node[speaker]['state']

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

See how simple the `update` function is! This simulation takes a lot more steps for the system to reach a homogeneous state, because each step involves only two nodes. It is probably a good idea to set the step size to 50 under the "Settings" tab to speed up

the simulations.

? Exercise 16.2.2

Revise the code above so that you can measure how many steps it will take until the system reaches a consensus (i.e., homogenized state). Then run multiple simulations (Monte Carlo simulations) to calculate the average time length needed for consensus formation in the original voter model.

? Exercise 16.2.3

Revise the code further to implement (1) the reversed and (2) the edge-based voter models. Then conduct Monte Carlo simulations to measure the average time length needed for consensus formation in each case. Compare the results among the three versions.

Discrete state/time models (2): Epidemic model

The second example of discrete state/time dynamical network models is the epidemic model on a network. Here we consider the *Susceptible-Infected-Susceptible (SIS) model*, which is even simpler than the SIR model we discussed in Exercise 7.3. In the SIS model, there are only two states: Susceptible and Infected. A susceptible node can get infected from an infected neighbor node with infection probability p_i (per infected neighbor), while an infected node can recover back to a susceptible node (i.e., no immunity acquired) with recovery probability p_r (Fig. 16.2.3). This setting still puts everything in binary states like in the earlier examples, so we can recycle their codes developed above.

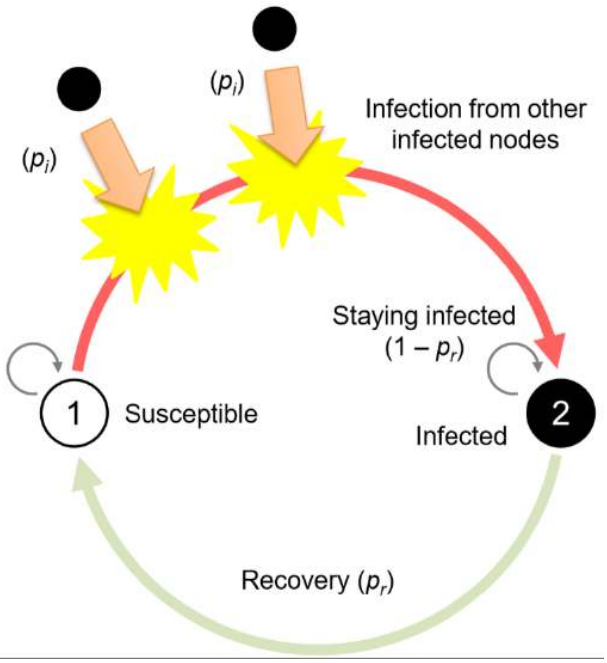


Figure 16.2.3: Schematic illustration of the state-transition rule of the SIS model

Regarding the updating scheme, I think you probably liked the simplicity of the asynchronous updating we used for the voter odel, so let's adopt it for this SIS model too. We will first choose a node randomly, and if it is susceptible, then we will randomly choose one of its neighbors; this is similar to the original "pull" version of the voter model. All you need is to revise just the `updating` function as follows:

Code 16.6: SIS-model.py

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

Again, you should set the step size to 50 to speed up the simulations. With these parameter settings ($p_i = p_r = 0.5$), you will probably find that the disease (state 1 = black nodes) quickly dies off and disappears from the network, even though half of the initial population is infected at the beginning. This means that for these particular parameter values and network topology, the system can successfully suppress a pandemic without taking any special action.

? Exercise 16.2.4

Conduct simulations of the SIS model with either p_i or p_r varied systematically, while the other one is fixed at 0.5. Determine the condition in which a pandemic occurs (i.e., the disease persists for an indefinitely long period of time). Is the transition gradual or sharp? Once you get the results, try varying the other parameter as well.

? Exercise 16.2.5

Generate a much larger random network of your choice and conduct the same SIS model simulation on it. See how the dynamics are affected by the change of network size and/or topology. Will the disease continue to persist on the network?

Continuous state/time models (1): Diffusion model

So far, we haven't seen any equations in this chapter, because all the models discussed above were described in algorithmic rules with stochastic factors involved. But if they are deterministic, dynamical network models are not fundamentally different from other conventional dynamical systems. In fact, any deterministic dynamical model of a network made of n nodes can be described in one of the following standard formulations using an n-dimensional vector state x,

$$x_t = F(x_{t-1}, t) \qquad \text{(for discrete-time models)} \tag{16.2.1}$$

or

$$\frac{dx}{dt} = F(x, t) \qquad \text{(for continuous-time models)} \tag{16.2.2}$$

If function F correctly represents the interrelationships between the different components of x. For example, the local majority rule network model we discussed earlier in this chapter is fully deterministic, so its behavior can be captured entirely in a set of difference equations in the form of Equation 16.2.1. In this and next subsections, we will discuss two illustrative examples of the differential equation version of dynamics on networks models, i.e., the diffusion model and the coupled oscillator model. They are both extensively studied in network science.

Diffusion on a network can be a generalization of spatial diffusion models into non-regular, non-homogeneous spatial topologies. Each node represents a local site where some "stuff" can be accumulated, and each symmetric edge represents a channel through

which the stuff can be transported, one way or the other, driven by the gradient of its concentration. This can be a useful model of the migration of species between geographically semi-isolated habitats, flow of currency between cities across a nation, dissemination of organizational culture within a firm, and so on. The basic assumption is that the flow of the stuff is determined by the difference in its concentration across the edge:

$$\frac{dc_i}{dt} = \alpha \sum_{j \in N_i} (c_i - c_i) \tag{16.2.3}$$

Here c_i is the concentration of the stuff on node i, α is the diffusion constant, and N_i is the set of node i's neighbors. Inside the parentheses $(c_j - c_i)$ represents the difference in the concentration between node j and node i across the edge (i, j). If neighbor j has more stuff than node i, there is an influx from j to i, causing a positive effect on dc_i/dt. Or if neighbor j has less than node i, there is an outflux from i to j, causing a negative effect on dc_i/dt. This makes sense.

Note that the equation above is a linear dynamical system. So, if we represent the entire list of node states by a state vector $\boldsymbol{c} = (c_1 c_2 \ldots c_n)^T$, Equation 16.2.3 can be written as

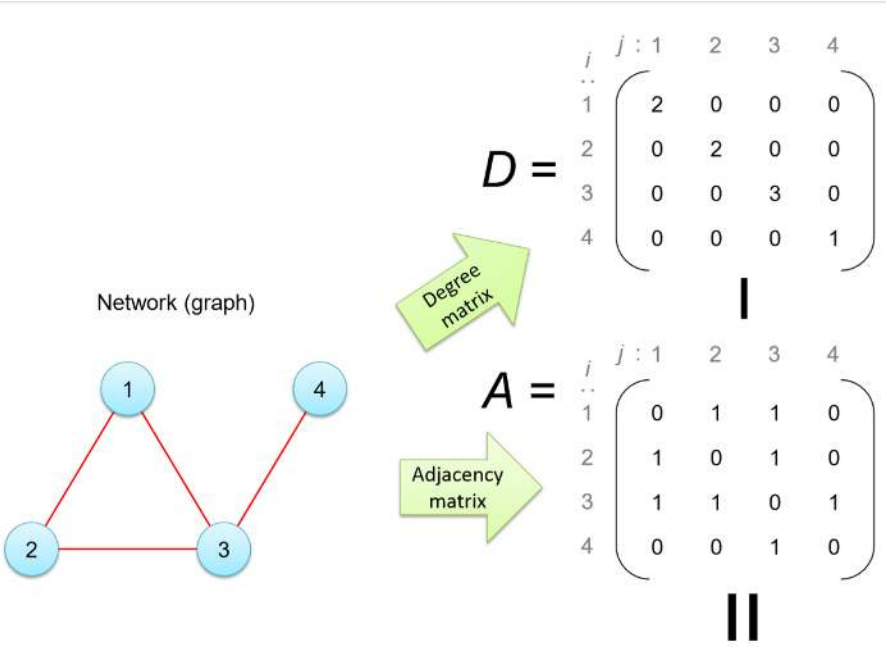
$$\frac{d\boldsymbol{c}}{dt} = -\alpha L \boldsymbol{c}, \tag{16.2.4}$$

where L is what is called a *Laplacian matrix* of the network, which is defined as

$$L = D - A, \tag{16.2.5}$$

where A is the adjacency matrix of the network, and D is the degree matrix of the network, i.e., a matrix whose i-th diagonal component is the degree of node i while all other components are 0. An example of those matrices is shown in Fig. 16.2.4.

Wait a minute. We already heard "Laplacian" many times when we discussed PDEs. The Laplacian operator (∇^2) appeared in spatial diffusion equations, while this new Laplacian matrix thing also appears in a network-based diffusion equation. Are they related? The answer is *yes*. In fact, they are (almost) identical linear operators in terms of their role. Remember that when we discretized Laplacian operators in PDEs to simulate using CA (Eq. 13.5.21), we learned that a discrete version of a Laplacian can be calculated by the following principle:



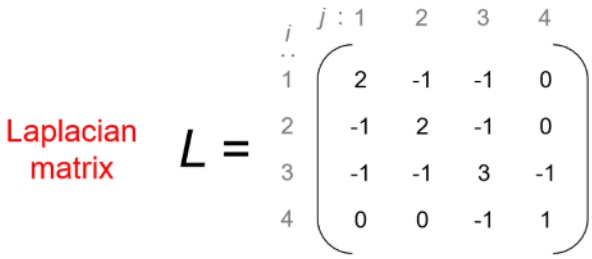


Figure **16.2.4**: Relationship among a degree matrix, an adjacent matrix, and a Laplacian matrix.

"Measure the difference between your neighbor and yourself, and then sum up all those differences."

Note that this is exactly what we did on a network in Equation **16.2.3**! So, essentially, the Laplacian matrix of a graph is a discrete equivalent of the Laplacian operator for continuous space. The only difference, which is quite unfortunate in my opinion, is that they are defined with opposite signs for historical reasons; compare the first term of Eq. 13.4.4 and Eq.**16.2.4**, and you will see that the Laplacian matrix did not absorb the minus sign inside it. So, conceptually,

$$\nabla^2 \Leftrightarrow -L. \tag{16.2.6}$$

I have always thought that this mismatch of signs was so confusing, but both of these "Laplacians" are already fully established in their respective fields. So, we just have to live with these inconsistent definitions of "Laplacians."

Now that we have the mathematical equations for diffusion on a network, we can discretize time to simulate their dynamics in Python. Equation **16.2.3** becomes

Erroneous nesting of equation structures

Or, equivalently, we can also discretize time in Equation **16.2.4**, i.e.,

Erroneous nesting of equation structures

where $\boldsymbol{c}$ is now the state vector for the entire network, and $\boldsymbol{I}$ is the identity matrix. Eqs.??? and ??? represent exactly the same diffusion dynamics, but we will use Equation ??? for the simulation in the following, because it won't require matrix representation (which could be inefficient in terms of memory use if the network is large and sparse).

We can reuse Code 16.4 for this simulation. We just need to replace the update function with the following:

Code 16.7: net-diffusion.py

```
alpha = 1 # diffusion constant
Dt = 0.01 # Delta t
```

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        ci = g.node[i]['state']
        nextg.node[i]['state'] = ci + alpha * ( \
            sum(g.node[j]['state'] for j in g.neighbors(i)) \
            - ci * g.degree(i)) * Dt
    g, nextg = nextg, g
```

And then we can simulate a nice smooth diffusion process on a network, as shown in Fig. 16.2.5, where continuous node states are represented by shades of gray. You can see that the diffusion makes the entire network converge to a homogeneous

configuration with the average node state (around 0.5, or half gray) everywhere.

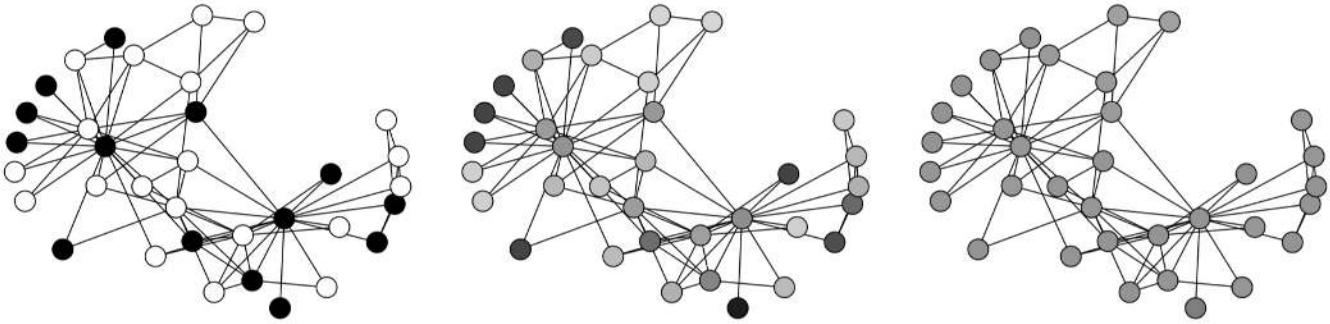


Figure* 16.2.5: Visual output of Code 16.7. Time flows from left to right.*

? Exercise 16.2.6

This diffusion model conserves the sum of the node states. Confirm this by revising the code to measure the sum of the node states during the simulation.

? Exercise 16.2.7

Simulate a diffusion process on each of the following network topologies, and then discuss how the network topology affects the diffusion on it. For example, does it make diffusion faster or slower?

- random graph
- barbell graph
- ring-shaped graph (i.e., degree-2 regular graph)

Continuous state/time models (2): Coupled oscillator model

Now that we have a diffusion model on a network, we can naturally extend it to reaction-diffusion dynamics as well, just like we did with PDEs. Its mathematical formulation is quite straightforward; you just need to add a local reaction term to Equation 16.2.3, to obtain

$$\frac{dc_i}{dt} = R_i(c_i) + \alpha \sum_{j \epsilon N_i} (c_j - c_i) \tag{16.2.7}$$

You can throw in any local dynamics to the reaction term R_i. If each node takes vector-valued states (like PDE-based reaction-diffusion systems), then you may also have different diffusion constants (α_1, α_2, ...) that correspond to multiple dimensions of the state space. Some researchers even consider different network topologies for different dimensions of the state space. Such networks made of superposed network topologies are called *multiplex networks*, which is a very hot topic actively studied right now (as of 2015), but we don't cover it in this textbook.

For simplicity, here we limit our consideration to scalar-valued node states only. Even with scalar-valued states, there are some very interesting dynamical network models. A classic example is *coupled oscillators*. Assume you have a bunch of oscillators, each of which tends to oscillate at its own inherent frequency in isolation. This inherent frequency is slightly different from oscillator to oscillator. But when connected together in a certain network topology, the oscillators begin to influence each other's oscillation phases. A key question that can be answered using this model is: *When and how do these oscillators synchronize?* This problem about *synchronization* of the collective is an interesting problem that arises in many areas of complex systems [69]: firing patterns of spatially distributed fireflies, excitation patterns of neurons, and behavior of traders in financial markets. In each of those systems, individual behaviors naturally have some inherent variations. Yet if the connections among them are strong enough and meet certain conditions, those individuals begin to orchestrate their behaviors and may show a globally synchronized behavior (Fig.16.6), which may be good or bad, depending on the context. This problem can be studied using network models. In fact,

addressing this problem was part of the original motivation for Duncan Watts' and Steven Strogatz's "small-world" paper published in the late 1990s [56], one of the landmark papers that helped create the field of network science.

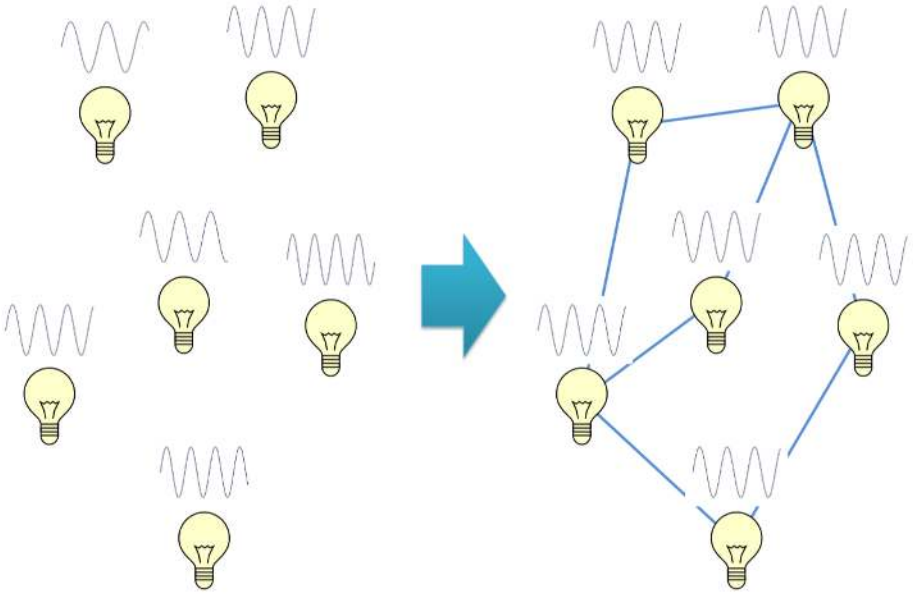


Figure **16.2.6**: Schematic illustration of coupled oscillators. Light bulbs that change light intensity periodically at different frequencies (left) may synchronize globally if connected (right).

Representing oscillatory behavior naturally requires two or more dynamical variables, as we learned earlier. But purely harmonic oscillation can be reduced to a simple linear motion with constant velocity, if the behavior is interpreted as a projection of uniform circular motion and if the state is described in terms of angle θ. This turns the reaction term in Equation 16.2.7 into just a constant angular velocity ω_i. In the meantime, this representation also affects the diffusion term, because the difference between two θ's is no longer computable by simple arithmetic subtraction like $\theta_j - \theta_i$, because there are infinitely many θ's that are equivalent (e.g., 0, 2π, -2π, 4π, -4π, etc.). So, the difference between the two states in the diffusion term has to be modified to focus only on the actual "phase difference" between the two nodes. You can imagine marching soldiers on a circular track as a visual example of this model (Fig. 16.2.7). Two soldiers who are far apart in θ may actually be close to each other in physical space (i.e., oscillation phase). The model should be set up in such a way that they try to come closer to each other in the physical space, not in the angular space.

To represent this type of phase-based interaction among coupled oscillators, a Japanese physicist Yoshiki Kuramoto proposed the following very simple, elegant mathematical model in the 1970s [70]:

$$\frac{d\theta_i}{dt} = w_i + \alpha \frac{\sum_{j \epsilon N_i} \sin(\theta_j - \theta i)}{|N_i|} \tag{16.2.8}$$

The angular difference part was replaced by a *sine function* of angular difference, which becomes positive if j is physically ahead of i, or negative if j is physically behind i on the circular track (because adding or subtracting 2π inside sin won't affect the result). These forces that soldier i receives from his or her neighbors will be averaged and used to determine the adjustment of his or her movement. The parameter α determines the strength of the couplings among the soldiers. Note that the original Kuramoto model used a fully connected network topology, but here we are considering the same dynamics on a network of a nontrivial topology.

Let's do some simulations to see if our networked soldiers can self-organize to march in sync. We can take the previous code for network diffusion (Code 16.7) and revise it for this Kuramoto model. There are several changes we need to implement. First, each node needs not only its dynamic state (θ_i) but also its static preferred velocity (ω_i), the latter of which should be initialized so that there are some variations among the nodes. Second, the visualization function should convert the states into some bounded values, since angular position θi continuously increases toward infinity. We can use $\sin(\theta_i)$ for this visualization purpose. Third, the updating rule needs to be revised, of course, to implement Equation 16.2.8.

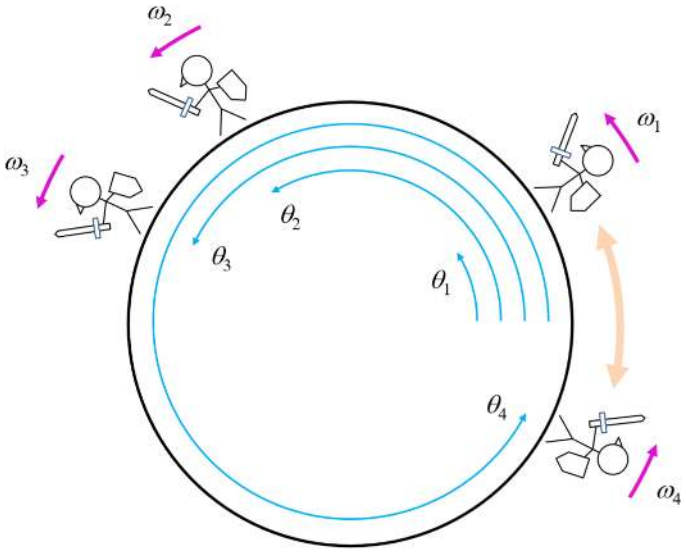


Figure **16.2.7**: Marching soldiers on a circular track, as a visual representation of coupled oscillators described in angular space, where soldier 1 and his or her neighbors onthenetwork(2,3,4)areillustrated. Each soldier has his or her own preferred speed (ω_i). The neighbors' physical proximity (not necessarily their angular proximity; see θ_1 and θ_4) should be used to determine in which direction soldier 1's angular velocity is adjusted.

Here is an example of the revised code, where I also changed the color map to `cm.hsv` so that the state can be visualized in a more cyclic manner:

Code 16.8: net-kuramoto.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['theta'] = 2 * pi * random()
        g.node[i]['omega'] = 1. + uniform(-0.05, 0.05)
    nextg = g.copy()
```

```
def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.hsv, vmin = -1, vmax = 1,
            node_color = [sin(g.node[i]['theta']) for i in g.nodes_iter()],
            pos = g.pos)

alpha = 1 # coupling strength
Dt = 0.01 # Delta t

def update():
    global g, nextg
    for i in g.nodes_iter():
        theta_i = g.node[i]['theta']
```

```
        nextg.node[i]['theta'] = theta_i + (g.node[i]['omega'] + alpha * ( \
            sum(sin(g.node[j]['theta'] - theta_i) for j in g.neighbors(i)) \
            / g.degree(i))) * Dt
    g, nextg = nextg, g
```

```
import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

Figure 16.2.8 shows a typical simulation run. The dynamics of this simulation can be better visualized if you increase the step size to 50. Run it yourself, and see how the inherently diverse nodes get self-organized to start asynchronized marching. Do you see any spatial patterns there?

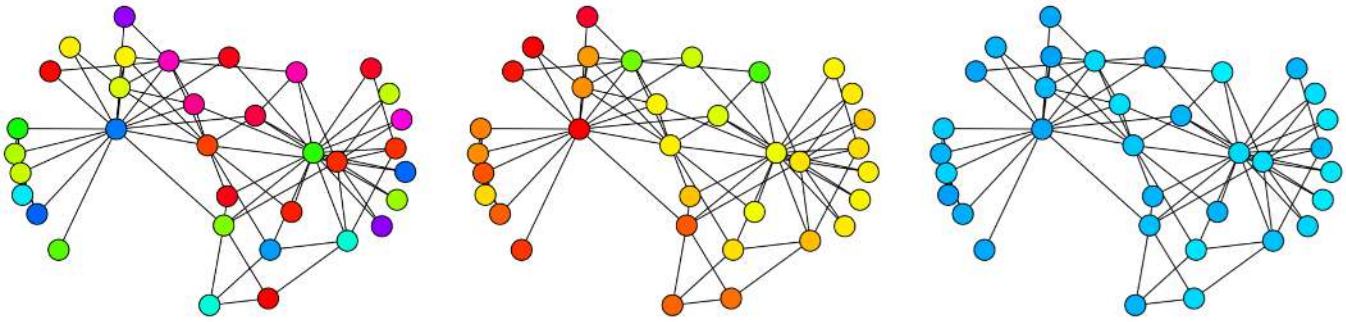


Figure **16.2.8**: Visual output of Code 16.8. Time flows from left to right.

? Exercise 16.2.8

It isknown that thelevel of synchronization in the Kuramoto model (and many other coupled oscillator models) can be characterized by the following measurement (called *phase coherence*):

$$r = |\frac{1}{n}\sum_i e^{i\theta_i}| \tag{16.2.9}$$

Here n is the number of nodes, θ_i is the state of node i, and i is the imaginary unit (to avoid confusion with node index i). This measurement becomes 1 if all the nodes are in perfect synchronization, or 0 if their phases are completely random and uniformly distributed in the angular space. Revise the simulation code so that you can measure and monitor how this phase coherence changes during the simulation.

? Exercise 16.2.9

Simulate the Kuramoto model on each of the following network topologies:

- random graph
- barbell graph
- ring-shaped graph (i.e., degree-2 regular graph)

Discuss how the network topology affects the synchronization. Will it make synchronization easier or more difficult?

? Exercise 16.2.10

Conduct simulations of the Kuramoto model by systematically increasing the amount of variations of ω_i (currently set to 0.05 in the `initialize` function) and see when/how the transition of the system behavior occurs.

Here are some more exercises of dynamics on networks models. Have fun!

? Exercise 16.2.11: Hopfield network (a.k.a. attractor network)

John Hopfield proposed a discrete state/time dynamical network model that can recover memorized arrangements of states from incomplete initial conditions [20, 21]. This was one of the pioneering works of artificial neural network research, and its basic principles are still actively used today in various computational intelligence applications. Here are the typical assumptions made in the Hopfield network model:

- The nodes represent artificial neurons, which take either -1 or 1 as dynamic states.
- Their network is fully connected (i.e., a complete graph).
- The edges are weighted and symmetric.
- The node states will update synchronously in discrete time steps according to the following rule:

$$s_i(t+1) = sign(\sum w_{ij} s_j(t)) \tag{16.2.10}$$

Here, $s_i(t)$ is the state of node i at time t, w_{ij} is the edge weight between nodes i and j ($w_{ij} = w_{ji}$ because of symmetry), and $sign(x)$ is a function that gives 1 if $x > 0$, -1 if $x < 0$, or 0 if $x = 0$.

- There are no self-loops (w_ii = 0 for all i).

What Hopfield showed is that one can "imprint" a finite number of pre-determined patterns into this network by carefully designing the edge weights using the following simple encoding formula:

$$w_{ij} = \sum s_{i,k} s_{j,k_k} \tag{16.2.11}$$

Here $s_{i,k}$ is the state of node i in the k-th pattern to be imprinted in the network. Implement a simulator of the Hopfield network model, and construct the edge weights w_{ij} from a few state patterns of your choice. Then simulate the dynamics of the network from a random initial condition and see how the network behaves. Once your model successfully demonstrates the recovery of imprinted patterns, try increasing the number of patterns to be imprinted, and see when/how the network loses the capability to memorize all of those patterns.

? Exercise 16.2.12: Cascading failure

This model is a continuous-state, discretetime dynamical network model that represents how a functional failure of a component in an infrastructure network can trigger subsequent failures and cause a large-scale systemic failure of the whole network. It is often used as a stylized model of massive power blackouts, financial catastrophe, and other (undesirable) socio-technological and socio-economical events.

Here are the typical assumptions made in the cascading failure model:

- The nodes represent a component of an infrastructure network, such as power transmitters, or financial institutions. The nodes take non-negative real numbers as their dynamic states, which represent the amount of load or burden they are handling. The nodes can also take a specially designated "dead" state.
- Each node also has its own capacity as a static property.
- Their network can be in any topology.
- The edges can be directed or undirected.
- The node states will update either synchronously or asynchronously in discrete time steps, according to the following simple rules:
  - If the node is dead, nothing happens.
  - If the node is not dead but its load exceeds its capacity, it will turn to a dead state, and the load it was handling will be evenly distributed to its neighbors that are still alive.

Implement a simulator of the cascading failure model, and simulate the dynamics of the network from an initial condition that is constructed with randomly assigned loads and capacities, with one initially overloaded node. Try increasing the average load level relative to the average capacity level, and see when/how the network begins to show a cascading failure. Investigate which node has the most significant impact when it is overloaded. Also try several different network topologies to see if topology affects the resilience of the networks.

? Exercise 16.2.13

Create a continuous state/time network model of coupled oscillators where each oscillator tries to *desynchronize* from its neighbors. You can modify the Kuramoto model to develop such a model. Then simulate the network from an initial condition where the nodes' phases are almost the same, and demonstrate that the network can spontaneously diversify node phases. Discuss potential application areas for such a model.

This page titled 16.2: Simulating Dynamics on Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

16.3: Simulating Dynamics of Networks

Dynamics *of* networks models capture completely different kinds of network dynamics, i.e., changes in network topologies. This includes the addition and removal of nodes and edges over time. As discussed in the previous chapter, such dynamic changes of the system's topology itself are quite unusual from a traditional dynamical systems viewpoint, because they would make it impossible to assume a well-defined static phase space of the system. But permitting such topological changes opens up a whole new set of possibilities to model various natural and social phenomena that were hard to capture in conventional dynamical systems frameworks, such as:

- Evolutionary changes of gene regulatory and metabolic networks
- Self-organization and adaptation of food webs
- Social network formation and evolution
- Growth of infrastructure networks (e.g., traffic networks, power grids, the Internet, WWW)
- Growth of scientific citation networks

As seen above, growth and evolution of networks in society are particularly well studied using dynamics *of* network models. This is partly because their temporal changes take place much faster than other natural networks that change at ecological/evolutionary time scales, and thus researchers can compile a large amount of temporal network data relatively easily.

One iconic problem that has been discussed about social networks is this: *Why is our world so "small" even though there are millions of people in it?* This was called the *"small world" problem* by social psychologist Stanley Milgram, who experimentally demonstrated this through his famous "six degrees of separation" experiment in the 1960s [71]. This empirical fact, that any pair of human individuals in our society is likely to be connected through a path made of only a small number of socialties, has been puzzling many people, including researchers. This problem doesn't sound trivial, because we usually have only the information about local social connections around ourselves, without knowing much about how each one of our acquaintances is connected to the rest of the world. But it is probably true that you are connected to, say, the President of the United States by fewer than ten social links.

This small-world problem already had a classic mathematical explanation. If the network is purely random, then the average distance between pairs of nodes are extremely small. In this sense, *Erdős–Rényi random graph models* were already able to explain the small-world problem beautifully. However, there is an issue with this explanation: *How come a person who has local information only can get connected to individuals randomly chosen from the entire society, who might be living on the opposite side of the globe?* This is a legitimate concern because, after all, most of our social connections are within a close circle of people around us. Most social connections are very local in both geographical and social senses, and there is no way we can create a truly random network in the real world. Therefore, we need different kinds of mechanisms by which social networks change their topologies to acquire the "small-world" property.

In the following, we will discuss two dynamics of networks models that can nicely explain the small-world problem in more realistic scenarios. Interestingly, these models were published at about the same time, in the late 1990s, and both contributed greatly to the establishment of the new field of network science.

Small-world networks by Random edge rewiring

In 1998, Duncan Watts and Steven Strogatz addressed this paradox, that social networks are "small" yet highly clustered locally, by introducing the *small-world network* model [56]. The *Watts-Strogatz model* is based on a random edge rewiring procedure to gradually modify network topology from a completely regular, local, clustered topology to a completely random, global, unclustered one. In the middle of this random rewiring process, they found networks that had the "small-world" property yet still maintained locally clustered topologies. They named these networks "small-world" networks. Note that having the "small-world" property is not a sufficient condition for a network to be called "small-world" in Watts-Strogatz sense. The network should also have high local clustering.

Watts and Strogatz didn't propose their random edge rewiring procedure as a dynamical process over time, but we can still simulate it as a dynamics on networks model. Here are their original model assumptions:

1. The initial network topology is a ring-shaped network made of n nodes. Each node is connected to k nearest neighbors (i.e., $k/2$ nearest neighbors clockwise and $k/2$ nearest neighbors counterclockwise) by undirected edges.
2. Each edge is subject to random rewiring with probability p. If selected for random rewiring, one of the two ends of the edge is reconnected to a node that is randomly chosen from the whole network. If this rewiring causes duplicate edges, the rewiring is canceled.

Watts and Strogatz's original model did the random edge rewiring (step 2 above) by sequentially visiting each node clockwise. But here, we can make a minor revision to the model so that the edge rewiring represents a more realistic social event, such as a random encounter at the airport of two individuals who live far apart from each other, etc. Here are the new, revised model assumptions:

1. The initial network topology is the same as described above.
2. In each edge rewiring event, a node is randomly selected from the whole network. The node drops one of its edges randomly, and then creates a new edge to a new node that is randomly chosen from the whole network (excluding those to which the node is already connected).

This model captures essentially the same procedure as Watts and Strogatz's, but the rewiring probability p is now represented implicitly by the length of the simulation. If the simulation continues indefinitely, then the network will eventually become completely random. Such a dynamical process can be interpreted as a model of social evolution in which an initially locally clustered society gradually accumulates more and more global connections that are caused by rare, random long-range encounters among people.

Let's simulate this model using Python. When you simulate the dynamics of networks, one practical challenge is how to visualize *topological changes* of the network. Unlike node states that can be easily visualized using colors, network topology is the shape of the network itself, and if the shape of the network is changing dynamically, we also need to simulate the physical movement of nodes in the visualization space as well. This is just for aesthetic visualization only, which has nothing to do with the real science going on in the network. But an effective visualization often helps us better understand what is going on in the simulation, so let's try some fancy animation here. Fortunately, NetworkX's `spring_layout` function can be used to update the current node positions slightly. For example:

Code 16.9:

```
g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)
```

This code calculates a new set of node positions by using `g.pos` as the initial positions and by applying the Fruchterman-Reingold force-directed algorithm to them for just five steps. This will effectively simulate a slight movement of the nodes from their current positions, which can be utilized to animate the topological changes smoothly.

Here is an example of the completed simulation code for the small-world network formation by random edge rewiring:

Code 16.10: small-world.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx
import random as rd

n = 30 # number of nodes
k = 4 # number of neighbors of each node

def initialize():
    global g
    g = nx.Graph()
    for i in xrange(n):
        for j in range(1, k/2 + 1):
            g.add_edge(i, (i + j) % n)
            g.add_edge(i, (i - j) % n)
    g.pos = nx.spring_layout(g)
    g.count = 0

def observe():
    global g
    cla()
    nx.draw(g, pos = g.pos)

def update():
    global g
    g.count += 1
    if g.count % 20 == 0: # rewiring once in every 20 steps
        nds = g.nodes()
```

```
        i = rd.choice(nds)
        if g.degree(i) > 0:
            g.remove_edge(i, rd.choice(g.neighbors(i)))
            nds.remove(i)
            for j in g.neighbors(i):
                nds.remove(j)
            g.add_edge(i, rd.choice(nds))

    # simulation of node movement
    g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

The `initialize` function creates a ring-shaped network topology by connecting each node to its k nearest neighbors ($j \in 1, \ldots, k/2$). "% n" is used to confine the names of nodes to the domain $[0, n-1]$ with periodic boundary conditions. An additional attribute `g.count` is also created in order to count time steps between events of topological changes. In the update function, `g.count` is incremented by one, and the random edge rewiring is simulated in every 20 steps as follows: First, node i is randomly chosen, and then one of its connections is removed. Then node `i` itself and its neighbors are removed from the candidate node list. A new edge is then created between i and a node randomly chosen from the remaining candidate nodes. Finally, regardless of whether a random edge rewiring occurred or not, node movement is simulated using the `spring_layout` function.

Running this code will give you an animated process of random edge rewiring, which gradually turns a regular, local, clustered network to a random, global, unclustered one (Fig. 16.3.1). Somewhere in this network dynamics, you will see the Watts-Strogatz smallworld network arise, but just temporarily (which will be discussed further in the next chapter).N

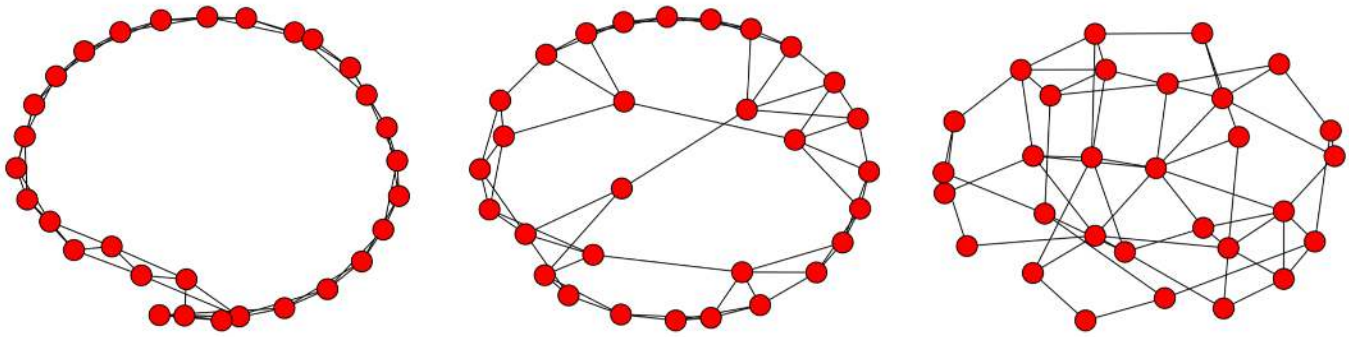


Figure **16.3.1**: Visual output of Code 16.10. Time flows from left to right.

? Exercise 16.3.1

Revise the small-world network formation model above so that the network is initially at two-dimensional grid in which each node is connected to its four neighbors (north, south, east, and west; except for those on the boundaries of the space). Then run the simulations, and see how random edge rewiring changes the topology of the network.

For your information, NetworkX already has a built-in graph generator function for Watts-Strogatz small-world networks, `watts_strogatz_graph(n, k, p)` . Here, n is the number of nodes, k is the degree of each node, and p is the edge

rewiring probability. If you don't need to simulate the formation of small-world networks iteratively, you should just use this function instead.

Scale-free networks by preferential attachment

The Watts-Strogatz small-world network model brought about a major breakthrough in explaining properties of real-world networks using abstract network models. But of course, it was not free from limitations. First, their model required a "Goldilocks" rewiring probability p (or such a simulation length in our revised model) in order to obtain a small-world network. This means that their model couldn't explain how such an appropriate p would be maintained in real social networks. Second, the small-world networks generated by their model didn't capture one important aspect of real-world networks: heterogeneity of connectivities. In every level of society,we often find a few very popular people as well as many other less-popular, "ordinary" people. In other words, there is always great variation in node degrees. However, since the Watts-Strogatz model modifies an initially regular graph by a series of random rewirings, the resulting networks are still quite close to regular, where each node has more or less the same number of connections. This is quite different from what we usually see in reality.

Just one year after the publication of Watts and Strogatz's paper, Albert-L´aszl´o Barab´asi and R´eka Albert published another very influential paper [57] about a new model that could explain both the small-world property and large variations of node degrees in a network. The *Barab´asi-Albert model* described self-organization of networks over time caused by a series of *network growth events* with *preferential attachment*. Their model assumptions were as follows:

1. The initial network topology is an arbitrary graph made of m_0 nodes. There is no specific requirement for its shape, as long as each node has at least one connection (so its degree is positive).
2. In each network growth event, a newcomer node is attached to the network by m edges ($m \leq m_0$). The destination of each edge is selected from the network of existing nodes using the following selection probabilities:

$$p(i) = \frac{\deg(1)}{\sum \deg(j)_j} \tag{16.3.1}$$

This preferential attachment mechanism captures "the rich get richer" effect in the growth of systems, which is often seen in many socio-economical, ecological, and physical processes. Such cumulative advantage is also called the "*Matthew effect*" in sociology. What Barab´asi and Albert found was that the network growing with this simple dynamical rule will eventually form a certain type of topology in which the distribution of the node degrees follows a *power law*

$$P(k) \approx k^{-\gamma}, \tag{16.3.2}$$

where $P(k)$ is the probability for a node to have degree k, and $-\gamma$ is the scaling exponent ($\gamma = 3$ in the Barab´asi-Albert model). Since the exponent is negative, this distribution means that most nodes have very small k, while only a few nodes have large k (Fig. 16.10). But the key implication is that such super-popular "*hub*" nodes *do exist* in this distribution, which wouldn't be possible if the distribution was a normal distribution, for example. This is because a power law distribution has a *long tail* (also called a fat tail, heavy tail, etc.), in which the probability goes down much more slowly than in the tail of a normal distribution as k gets larger. Barab´asi and Albert called networks whose degree distributions follow a power law *scale-free networks*, because there is no characteristic "scale" that stands out in their degree distributions.

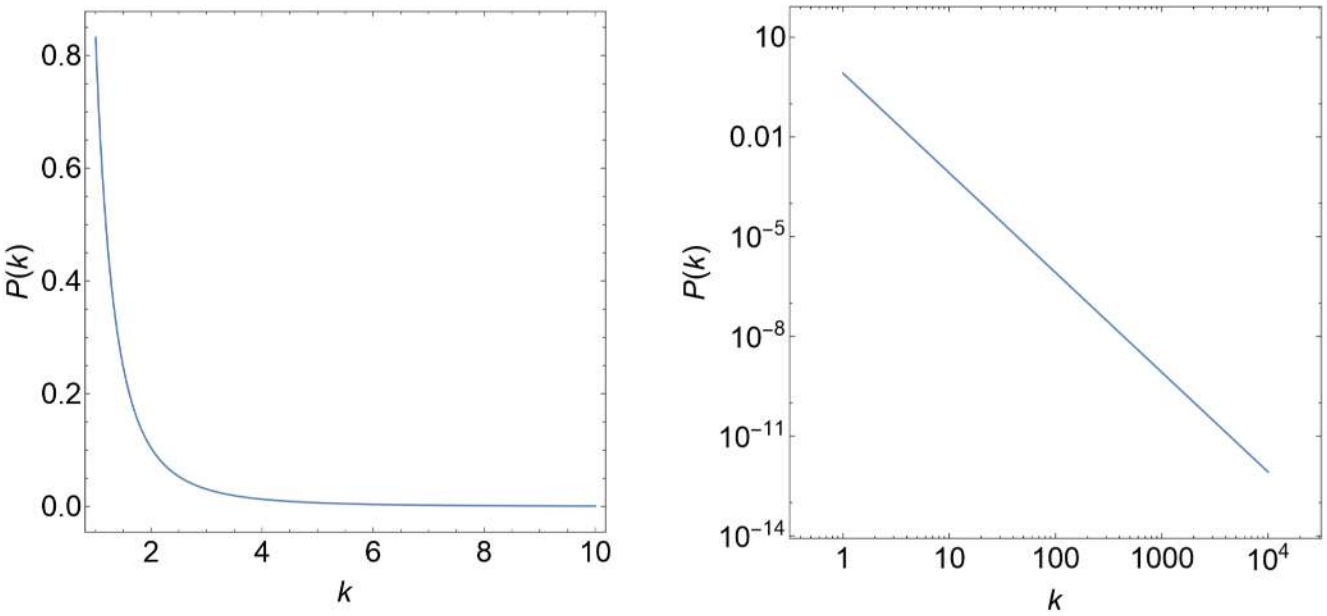


***Figure* 16.3.2**:*Plots of the power law distribution* $P(k) \sim k^{\gamma}$ *with* $\gamma = 3$. *Left: In linear scales. Right: In log-log scales.*

The network growth with preferential attachments is an interesting, fun process to simulate. We can reuse most of Code 16.10 for this purpose as well. One critical component we will need to newly define is the preferential node selection function that randomly chooses a node based on node degrees. This can be accomplished by *roulette selection*, i.e., creating a "roulette" from the node degrees and then deciding which bin in the roulette a randomly generated number falls into. In Python, this preferential node selection function can be written as follows:

Code 16.11:

```
def pref_select(nds):
    global g
    r = uniform(0, sum(g.degree(i) for i in nds))
    x = 0
    for i in nds:
        x += g.degree(i)
        if r <= x:
            return i
```

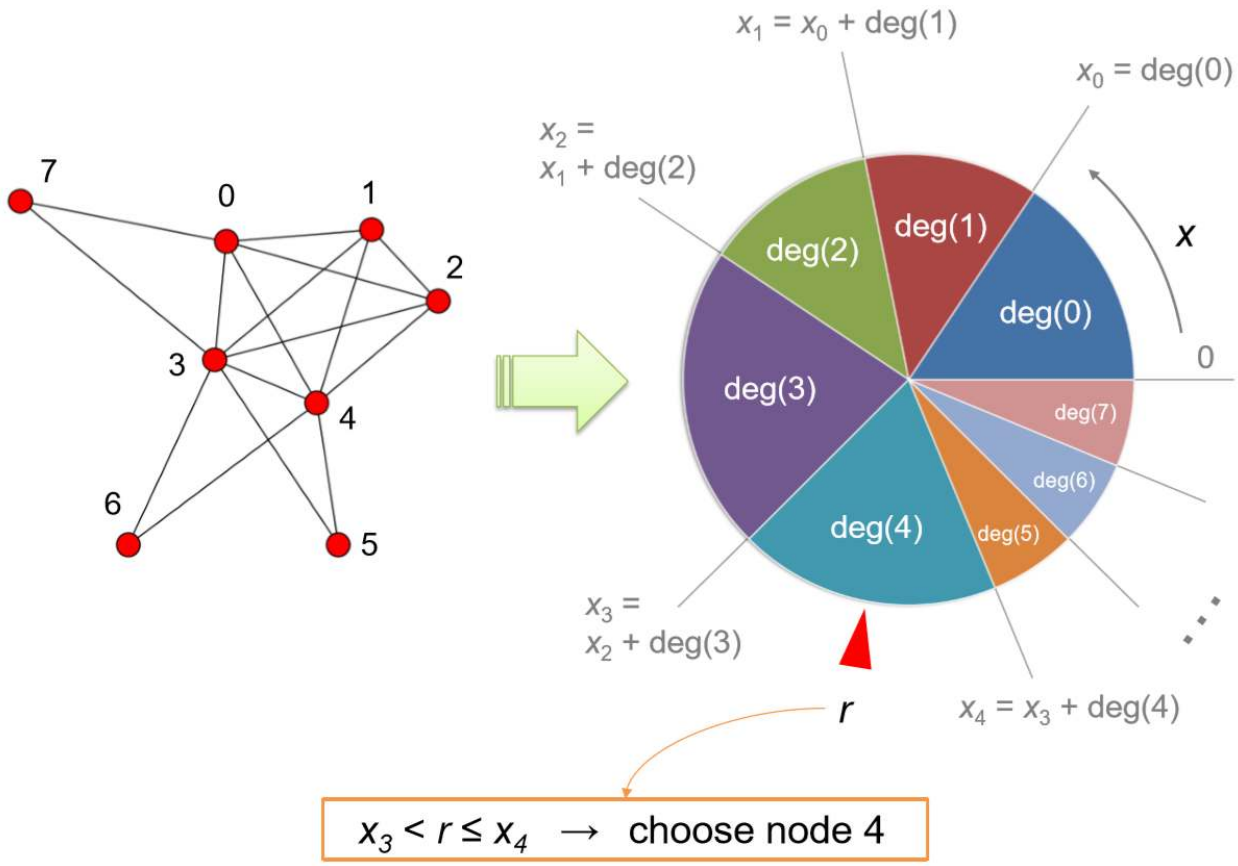


Figure **16.3.3**: Schematic illustration of the roulette selection implemented in Code 16.11.

With this preferential node selection function, the completed simulation code looks like this:

Code 16.12: barabasi-albert.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx

m0 = 5 # number of nodes in initial condition
m = 2 # number of edges per new node

def initialize():
    global g
    g = nx.complete_graph(m0)
    g.pos = nx.spring_layout(g)
```

```
def observe():
    global g
    cla()
    nx.draw(g, pos = g.pos)

def pref_select(nds):
    global g
    r = uniform(0, sum(g.degree(i) for i in nds))
    x = 0
    for i in nds:
        x += g.degree(i)
        if r <= x:
            return i
```

```
def update():
    global g
    g.count += 1
    if g.count % 20 == 0: # network growth once in every 20 steps
        nds = g.nodes()
```

```
        newcomer = max(nds) + 1
        for i in range(m):
            j = pref_select(nds)
            g.add_edge(newcomer, j)
            nds.remove(j)
        g.pos[newcomer] = (0, 0)

    # simulation of node movement
    g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)
```

```
import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

In this sample code, the network growth starts with a five-node complete graph. Each time a newcomer node is added to the network, two new edges are established. Note that every time an edge is created between the newcomer and an existing node `j`, node `j` is removed from the candidate list `nds` so that no duplicate edges will be created. Finally, a new position (0, 0) for the newcomer is added to `g.pos`. Since `g.pos` is a Python dictionary, a new entry can be inserted just by using `newcomer` as a key.

Figure 16.12 shows a typical simulation result in which you see highly connected hub nodes spontaneously arise. When you continue this simulation long enough, the resulting distribution of node degrees comes closer to a power law distribution with `γ = 3`.

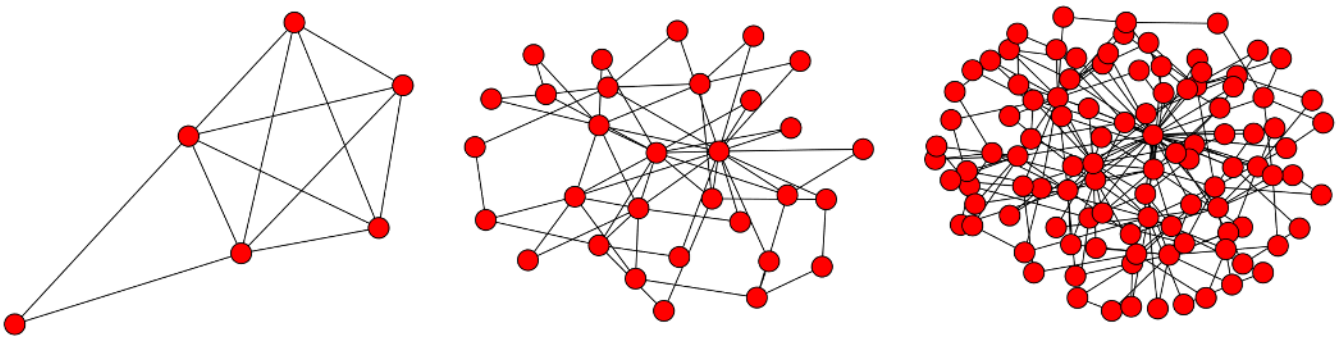


Figure **16.3.4**:Visual output of Code 16.12. Time flows from left to right.

? Exercise 16.3.2

Simulate the Barab´asi-Albert network growth model with $m = 1$, $m = 3$, and $m = 5$, and see how the growth process may be affected by the variation of this parameter.

? Exercise 16.3.3

Modify the preferential node selection function so that the node selection probability $p(i)$ is each of the following:

- independent of the node degree (random attachment)
- proportional to the square of the node degree(strong preferential attachment)
- inversely proportional to the node degree (negative preferential attachment) Conduct simulations for these cases and compare the resulting network topologies.

Note that NetworkX has a built-in graph generator function for the Barab´asi-Albert scale-free networks too, called `barabasi_albert_graph(n, m)`. Here, `n` is the number of nodes, and `m` the number of edges by which each newcomer node is connected to the network.

Here are some more exercises of dynamics of networks models, for your further exploration:

? Exercise 16.3.4: Closing triangles

This example is a somewhat reversed version of the Watts-Strogatz small-world network model. Instead of making a local, clustered network to a global, unclustered network, we can consider a dynamical process that makes an unclustered network more clustered over time. Here are the rules:

- The network is initially random, e.g., an Erd˝os-R´enyi random graph.
- In each iteration, a two-edge path (A-B-C) is randomly selected from the network.
- If there is no edge between A and C, one of them loses one of its edges (but not the one that connects to B), and A and C are connected to each other instead.

This is an edge rewiring process that closes a triangle among A, B, and C, promoting what is called a *triadic closure* in sociology. Implement this edge rewiring model, conduct simulations, and see what kind of network topology results from this triadic closure rule. You can also combine this rule with the random edge rewiring used in the Watts-Strogatz model, to explore various balances between these two competing rules (globalization and localization) that shape the self-organization of the network topology.

? Exercise 16.3.5: Preferential attachment with node division

We can consider a modification of the Barab´asi-Albert model where each node has a capacity limit in terms of the number of connections it can hold. Assume that if a node's degree exceeds its predefined capacity, the node splits into two, and each node inherits about half of the connections the original node had. This kind of node division can be considered a representation of a split-up of a company or an organization, or the evolution of different specialized genes from a single gene that had many functions. Implement this modified network growth model, conduct simulations, and see how the node division influences the resulting network topology.

This page titled 16.3: Simulating Dynamics of Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

16.4: Simulating Adaptive Networks

The final class of dynamical network models is that of adaptive networks. It is a hybrid of dynamics on and of networks, where states and topologies "co-evolve," i.e., they interact with each other and keep changing, often over the same time scales. Theword "adaptive" comes from the concept that states and topologies can adapt to each other in a co-evolutionary manner, although adaptation or co-evolution doesn't have to be biological in this context. Adaptive networks have been much less explored compared to the other two classes of models discussed above, bbut you can find many real-world examples of adaptive networks [67], such as:

- *Development of an organism.* The nodes are the cells and the edges are cell-cell adhesions and intercellular communications. The node states include intracellular gene regulatory and metabolic activities, which are coupled with topological changes caused by cell division, death, and migration.
- *Self-organization of ecological communities.* The nodes are the species and the edges are the ecological relationships (predation, symbiosis, etc.) amongt hem. The node states include population levels and within-species genetic diversities, which are coupled with topological changes caused by invasion, extinction, adaptation, and speciation.
- *Epidemiological networks.* The nodes are the individuals and the edges are the physical contacts among them. The node states include their pathologic states (healthy or sick, infectious or not, etc.), which are coupled with topological changes caused by death, quarantine, and behavioral changes of those individuals.
- *Evolution of social communities.* The nodes are the individuals and the edges are social relationships, conversations, and/or collaborations. The node states include socio-cultural states, political opinions, wealth, and other social properties of those individuals,whicharecoupledwithtopologicalchangescausedbythepeople'sentry into or withdrawal from the community, as well as their behavioral changes.

In these adaptive networks, state transitions of each node and topological transformation of networks are deeply coupled with each other, which could produce characteristic behaviors that are not found in other forms of dynamical network models. As I mentioned earlier in this chapter, I am one of the proponents of this topic [66], so you should probably take what I say with a pinch of salt. But I firmly believe that modeling and predicting state-topology co-evolution of adaptive networks has become one of the major critical challenges in complex network research, especially because of the continual flood of temporal network data [65] that researchers are facing today.

The field of adaptive network research is still very young, and thus it is a bit challenging to select "well established" models as classic foundations of this research yet. So instead, I would like to show, with some concrete examples, how you can revise traditional dynamical network models into adaptive ones. You will probably find it very easy and straightforward to introduce adaptive network dynamics. There is nothing difficult in simulating adaptive network models, and yet the resulting behaviors may be quite unique and different from those of traditional models. This is definitely one of those unexplored research areas where your open-minded creativity in model development will yield you much fruit.

Adaptive epidemic model When you find out that one of your friends has caught a flu, you will probably not visit his or her room until he or she recovers from the illness. This means that human behavior, in response to the spread of illnesses, can change the topology of social ties, which in turn will influence the pathways of disease propagation. This is an illustrative example of adaptive network dynamics, which we can implement into the SIS model (Code 16.6) very easily. Here is the additional assumption we will add to the model:

- When a susceptible node finds its neighbor is infected by the disease, it will sever the edge to the infected node with severance probability p_s.

This new assumption is inspired by a pioneering adaptive network model of epidemic dynamics proposed by my collaborator Thilo Gross and his colleagues in 2006 [72]. They assumed that the edge from the susceptible to the infected nodes would be rewired to another susceptible node in order to keep the number of edges constant. But here, we assume that such edges can just be removed for simplicity. For your convenience, Code 16.6 for the SIS model is shown below again. Can you see where you can/should implement this new edge removal assumption in this code?

Code 16.13:

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

There are several different ways to implement adaptive edge removal in this code. An easy option would be just to insert another if statement to decide whether to cut the edge right before simulating the infection of the disease (third-to-last line). For example:

Code 16.14:

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability
p_s = 0.5 # severance probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            if random() < p_s:
                g.remove_edge(a, b)
            else:
                g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

Note that we now have one more probability, `p_s`, with which the susceptible node can sever the edge connecting itself to an infected neighbor. Since this simulation is implemented using an asynchronous state updating scheme, we can directly remove the edge `(a, b)` from the network as soon as node `a` decides to cut it. Such asynchronous updating is particularly suitable for simulating adaptive network dynamics in general.

By the way, there is a minor bug in the code above. Because there is a possibility for the edges to be removed from the network, some nodes may eventually lose all of their neighbors. If this happens, the `rd.choice(g.neighbors(a))` function causes an error. To avoid this problem, we need to check if node a's degree is positive. Below is a slightly corrected code:

Code 16.15: SIS-model-adaptive.py

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability
p_s = 0.5 # severance probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        if g.degree(a) > 0:
            b = rd.choice(g.neighbors(a))
            if g.node[b]['state'] == 1: # if neighbor b is infected
                if random() < p_s:
                    g.remove_edge(a, b)
                else:
                    g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

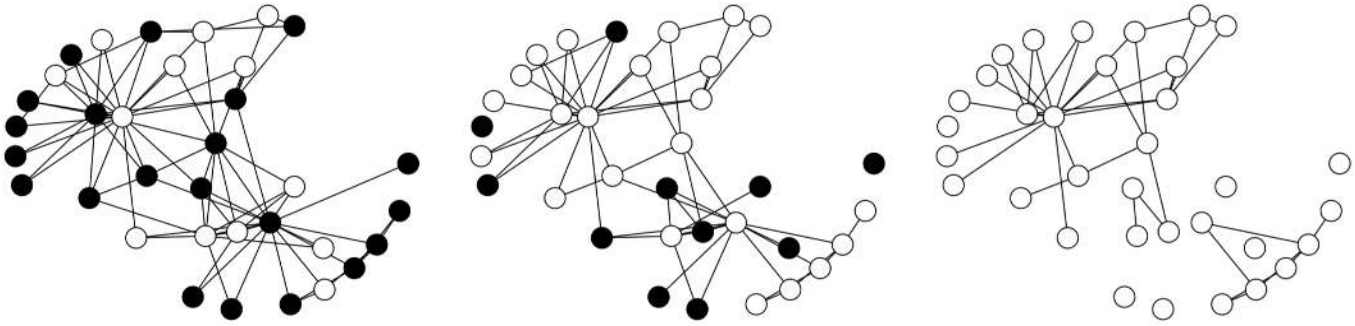


Figure **16.4.1**: Visual output of Code 16.15, with varied parameter settings ($p_i = 0.5$, $p_r = 0.2$, $p_s = 0.5$). Time flows from left to right.

? Exercise 16.4.1

Conduct simulations of the adaptive SIS model on a random network of a larger size with $p_i = 0.5$ and $p_r = 0.2$, while varying ps systematically. Determine the condition in which a pandemic will eventually be eradicated. Then try the same simulation on different network topologies (e.g, small-world, scale-free networks) and compare the results.

? Exercise 16.4.2

Implement a similar edge removal rule in the voter model so that an edge between two nodes that have opposite opinions can be removed probabilistically. Then conduct simulations with various edge severance probabilities to see how the consensus formation process is affected by the adaptive edge removal.

Adaptive diffusion model The final example in this chapter is the adaptive network version of the continuous state/time diffusion model, where the weight of a social tie can be strengthened or weakened according to the difference of the node states across the edge. This new assumption represents a simplified form of *homophily*, an empirically observed sociological fact that people tend to connect those who are similar to themselves. In contrast, the diffusion of node states can be considered a model of *social contagion*, another empirically observed sociological fact that people tend to become more similar to their social neighbors over time. There has been a lot of debate going on about which mechanism, homophily or social contagion, is more dominant in shaping the structure of social networks. This adaptive diffusion model attempts to combine these two mechanisms to investigate their subtle balance via computer simulations.

This example is actually inspired by an adaptive network model of a corporate merger that my collaborator Junichi Yamanoi and I presented recently [73]. We studied the conditions for two firms that recently underwent a merger to successfully assimilate and integrate their corporate cultures into a single, unified identity. The original model was a bit complex agent-based model that involved asymmetric edges, multi-dimensional state space, and stochastic decision making, so here we use a more simplified, deterministic, differential equation-based version. Here are the model assumptions:

- The network is initially made of two groups of nodes with two distinct cultural/ideological states.
- Each edge is undirected and has a weight, $w \in [0,1]$, which represents the strength of the connection. Weights are initially set to 0.5 for all the edges.
- The diffusion of the node states occurs according to the following equation:

$$\frac{dc_i}{dt} = \alpha \sum (c_j - c_i) w_{ij\, j \epsilon N_i} \tag{16.4.1}$$

Here α is the diffusion constant and w_{ij} is the weight of the edge between node i and node j. The inclusion of w_{ij} signifies that diffusion takes place faster through edges with greater weights.

- In the meantime, each edge also changes its weight dynamically, according to the following equation:

$$\frac{dw_{ij}}{dt} = \beta w_{ij}(1 - w_{ij})(1 - \gamma\, |c_i - c_j|) \tag{16.4.2}$$

Here β is the rate of adaptive edge weight change, and γ is a parameter that determines how intolerant, or "picky," each node is regarding cultural difference. For example, if $\gamma = 0$, w_{ij} always converges and stays at 1. But if γ is large (typically much larger than 1), the two nodes need to have very similar cultural states in order to maintain an edge between them, or otherwise the edge weight decreases. The inclusion of $w_{ij}(1 - w_{ij})$ is to confine the weight values to the range $[0,1]$ dynamically.

So, to simulate this model, we need a network made of two distinct groups. And this is the perfect moment to disclose a little more secrets about our favorite Karate Club graph (unless you have found it yourself already)! See the node attribute of the Karate Club graph, and you will find the following:

Code 16.16:

```
>>> nx.karate_club_graph().node
{0: {'club': 'Mr. Hi'}, 1: {'club': 'Mr. Hi'}, 2: {'club': 'Mr. Hi'},
  3: {'club': 'Mr. Hi'}, 4: {'club': 'Mr. Hi'}, 5: {'club': 'Mr. Hi'},
  6: {'club': 'Mr. Hi'}, 7: {'club': 'Mr. Hi'}, 8: {'club': 'Mr. Hi'},
  9: {'club': 'Officer'}, 10: {'club': 'Mr. Hi'}, 11: {'club': 'Mr. Hi'},
  12: {'club': 'Mr. Hi'}, 13: {'club': 'Mr. Hi'}, 14: {'club': 'Officer'},
  15: {'club': 'Officer'}, 16: {'club': 'Mr. Hi'}, 17: {'club': 'Mr. Hi'},
  18: {'club': 'Officer'}, 19: {'club': 'Mr. Hi'}, 20: {'club': 'Officer'},
  21: {'club': 'Mr. Hi'}, 22: {'club': 'Officer'}, 23: {'club': 'Officer'},
  24: {'club': 'Officer'}, 25: {'club': 'Officer'}, 26: {'club': 'Officer'},
  27: {'club': 'Officer'}, 28: {'club': 'Officer'}, 29: {'club': 'Officer'},
  30: {'club': 'Officer'}, 31: {'club': 'Officer'}, 32: {'club': 'Officer'},
  33: {'club': 'Officer'}}
```

Each node has an additional property, called `'club'`, and its values are either `'Mr. Hi'` or `'Officer'`! What are these?

The truth is, when Wayne Zachary studied this Karate Club, there was an intense political/ideological conflict between two factions. One was Mr. Hi, a part-time karate instructor hired by the club, and his supporters, while the other was the club president (Officer) John A., other officers, and their followers. They were in sharp conflict over the price of karate lessons. Mr. Hi wanted to raise the price, while the club president wanted to keep the price as it was. The conflict was so intense that the club eventually fired Mr. Hi, resulting in a fission of the club into two. The Karate Club graph is a snapshot of the club members' friendship network right before this fission, and therefore, each node comes with an attribute showing whether he or she was in Mr. Hi's camp or the Officer's camp. If you are interested in more details of this, you should read Zachary's original paper [59], which contains some more interesting stories.

his data looks perfect for our modeling purpose. We can set up the `initialize` function using this `'club'` property to assign binary states to the nodes. The implementation of dynamical equations are straightforward; we just need to discretize time and simulate them just as a discrete-time model, as we did before. Here is a completed code:

Code 16.17: net-diffusion-adaptive.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    for i, j in g.edges_iter():
        g.edge[i][j]['weight'] = 0.5
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if g.node[i]['club'] == 'Mr. Hi' else 0
    nextg = g.copy()

def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            edge_cmap = cm.binary, edge_vmin = 0, edge_vmax = 1,
            edge_color = [g.edge[i][j]['weight'] for i, j in g.edges_iter()],
            pos = g.pos)

alpha = 1 # diffusion constant
beta = 3 # rate of adaptive edge weight change
gamma = 3 # pickiness of nodes
Dt = 0.01 # Delta t
```

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        ci = g.node[i]['state']
        nextg.node[i]['state'] = ci + alpha * ( \
            sum((g.node[j]['state'] - ci) * g.edge[i][j]['weight']
                for j in g.neighbors(i))) * Dt
    for i, j in g.edges_iter():
        wij = g.edge[i][j]['weight']
        nextg.edge[i][j]['weight'] = wij + beta * wij * (1 - wij) * ( \
            1 - gamma * abs(g.node[i]['state'] - g.node[j]['state'])
            ) * Dt
    nextg.pos = nx.spring_layout(nextg, pos = g.pos, iterations = 5)
    g, nextg = nextg, g
```

```
import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

In the `initialize` function, the edge weights are all initialized as 0.5, while node states are set to 1 if the node belongs to Mr. Hi's faction, or 0 otherwise. There are also some modifications made to the observe function. Since the edges carry weights, we should color each edge in a different gray level. Therefore additional options(`edge_color` , `edge_cmap` , `edge_vmin` , and `edge_vmax`) were used to draw the edges in different colors. The update function is pretty straightforward, I hope.

Figure 16.4.1 shows the simulation result with $\alpha = 1$, $\beta = 3$, $\gamma = 3$. With this setting, unfortunately, the edge weights became weaker and weaker between the two political factions (while they became stronger within each), and the Karate Club was eventually split into two, which was what actually happened. In the meantime, we can also see that there were some cultural/ideological diffusion taking place across the two factions, because their colors became a little more gray-ish than at the beginning. So, there was apparently a competition not just between the two factions, but also between the two different dynamics: social assimilation and fragmentation. History has shown that the fragmentation won in the actual Karate Club case, but we can explore the parameter space of this model to see if there was any alternative future possible for the Karate Club!

? Exercise 16.4.3

Examine, by simulations, the sensitivity of the Karate Club adaptive diffusion model on the initial node state assignments. Can a different node

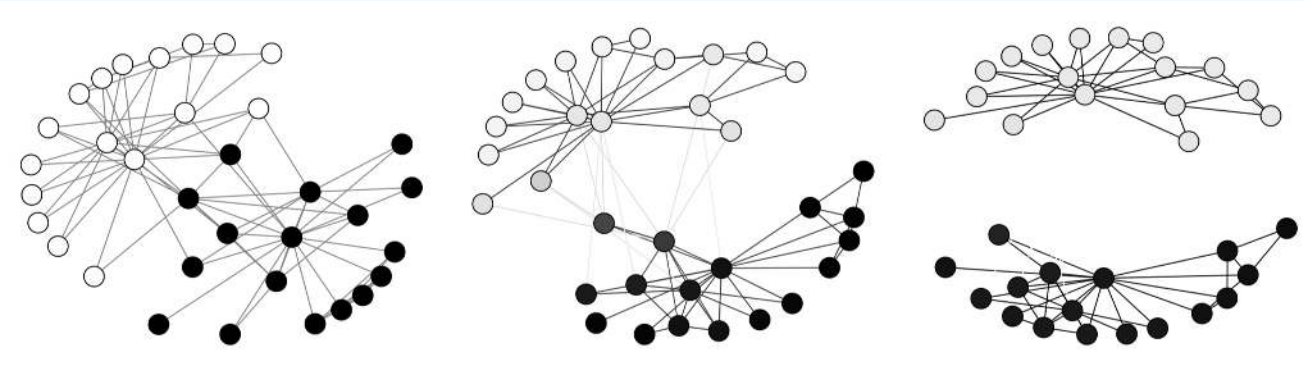


Figure 16.4.2: Visual output of Code 16.17. Time flows from left to right.

state assignment (with the same numbers of 0's and 1's) prevent the fission of the Club? If so, what are the effective strategies to assign the node states?

? Exercise 16.4.4

Conduct simulations of the Karate Club adaptive diffusion model (with the original node state assignment) by systematically varying α, β, and γ, to find the parameter settings with which homogenization of the node states can be achieved. Then think about what kind of actions the members of the Karate Club could have taken to achieve those parameter values in a real-world setting, in order to avoid the fragmentation of the Club.

This page titled 16.4: Simulating Adaptive Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

17: Dynamical Networks II - Analysis of Network Topologies

17.1: Network Size, Density, and Percolation

17.2: Shortest Path Length

17.3: Centralities and Coreness

17.4: Clustering

17.5: Degree Distribution

17.6: Assortativity

17.7: Community Structure and Modularity

This page titled 17: Dynamical Networks II - Analysis of Network Topologies is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.1: Network Size, Density, and Percolation

Networks can be analyzed in several different ways. One way is to analyze their structural features, such as size, density, topology, and statistical properties.

Let me first begin with the most basic structural properties, i.e., the size and density of a network. These properties are conceptually similar to the mass and composition of matter—they just tell us how much stuff is in it, but they don't tell us anything about how the matter is organized internally. Nonetheless, they are still the most fundamental characteristics, which are particularly important when you want to compare multiple networks. You should compare properties of two networks of the same size and density, just like chemists who compare properties of gold and copper of the same mass.

> The size of a network is characterized by the numbers of nodes and edges in it.

NetworkX's `Graph` objects have functions dedicated for measuring those properties:

Code 17.1:

```
>>> g = nx.karate_club_graph()
>>> g.number_of_nodes()
34
>>> g.number_of_edges()
78
```

> The density of a network is the fraction between 0 and 1 that tells us what portion of all possible edges are actually realized in the network. For a network G made of n nodes and m edges, the density $\rho(G)$ is given by
>
> $$p(G) = \frac{m}{\frac{n(n-1)}{2}} = \frac{2m}{n(n-1)} \tag{17.1.1}$$
>
> for an undirected network, or
>
> $$p(G) = \frac{m}{n-1} \tag{17.1.2}$$
>
> for a directed network.

NetworkX has a built-in function to calculate network density:

Code 17.2:

```
>>> g = nx.karate_club_graph()
>>> nx.density(g)
0.13903743315508021
```

Note that the size and density of a network don't specify much about the network's actual topology(i.e.,shape). There are many networks with different topologies that have exactly the same size and density.

But there are some things the size and density can still predict about networks. One such example is *network percolation*, i.e., whether or not the nodes are sufficiently connected to each other so that they form a *giant component* that is visible at macroscopic

scales. I can show you an example. In the code below, we generate Erd˝os-R´enyi random graphs made of 100 nodes with different connection probabilities:

Code 17.3: net-percolation.py

```
from pylab import *
import networkx as nx

for i, p in [(1, 0.0001), (2, 0.001), (3, 0.01), (4, 0.1)]:
    subplot(1, 4, i)
    title('p = ' + str(p))
    g = nx.erdos_renyi_graph(100, p)
    nx.draw(g, node_size = 10)
```

```
show()
```

The result is shown in Fig. 17.1.1, where we can clearly see a transition from a completely disconnected set of nodes to a single giant network that includes all nodes.

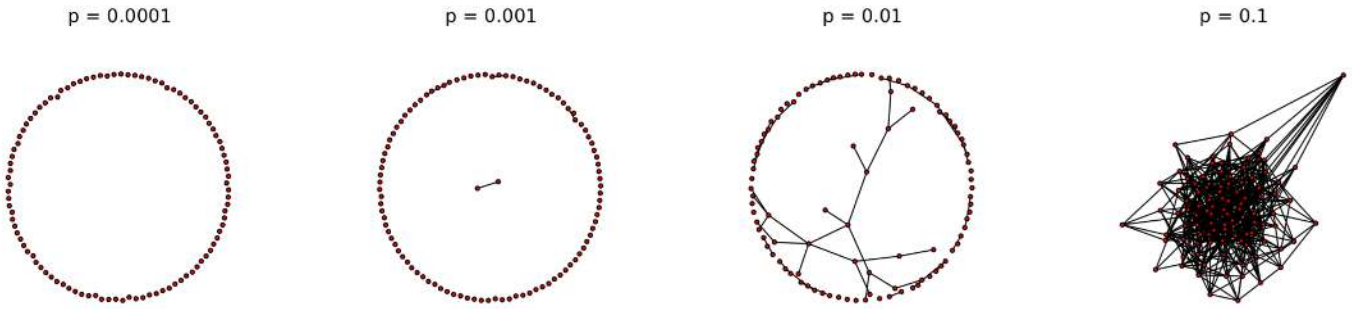


Figure **17.1.1**: Visual output of Code 17.3.

The following code conducts a more systematic parameter sweep of the connection probability:

Code 17.4: net-percolation-plot.py

```
from pylab import *
import networkx as nx

p = 0.0001
pdata = []
gdata = []

while p < 0.1:
    pdata.append(p)
    g = nx.erdos_renyi_graph(100, p)
    ccs = nx.connected_components(g)
    gdata.append(max(len(cc) for cc in ccs))
    p *= 1.1
```

```
loglog(pdata, gdata)
xlabel('p')
ylabel('size of largest connected component')
show()
```

In this code, we measure the size of the largest connected component in an Erd˝os-R´enyi graph with connection probability p, while geometrically increasing p. The result shown in Fig. 17.1.2 indicates that a percolation transition happened at around $p = 10^{-2}$.

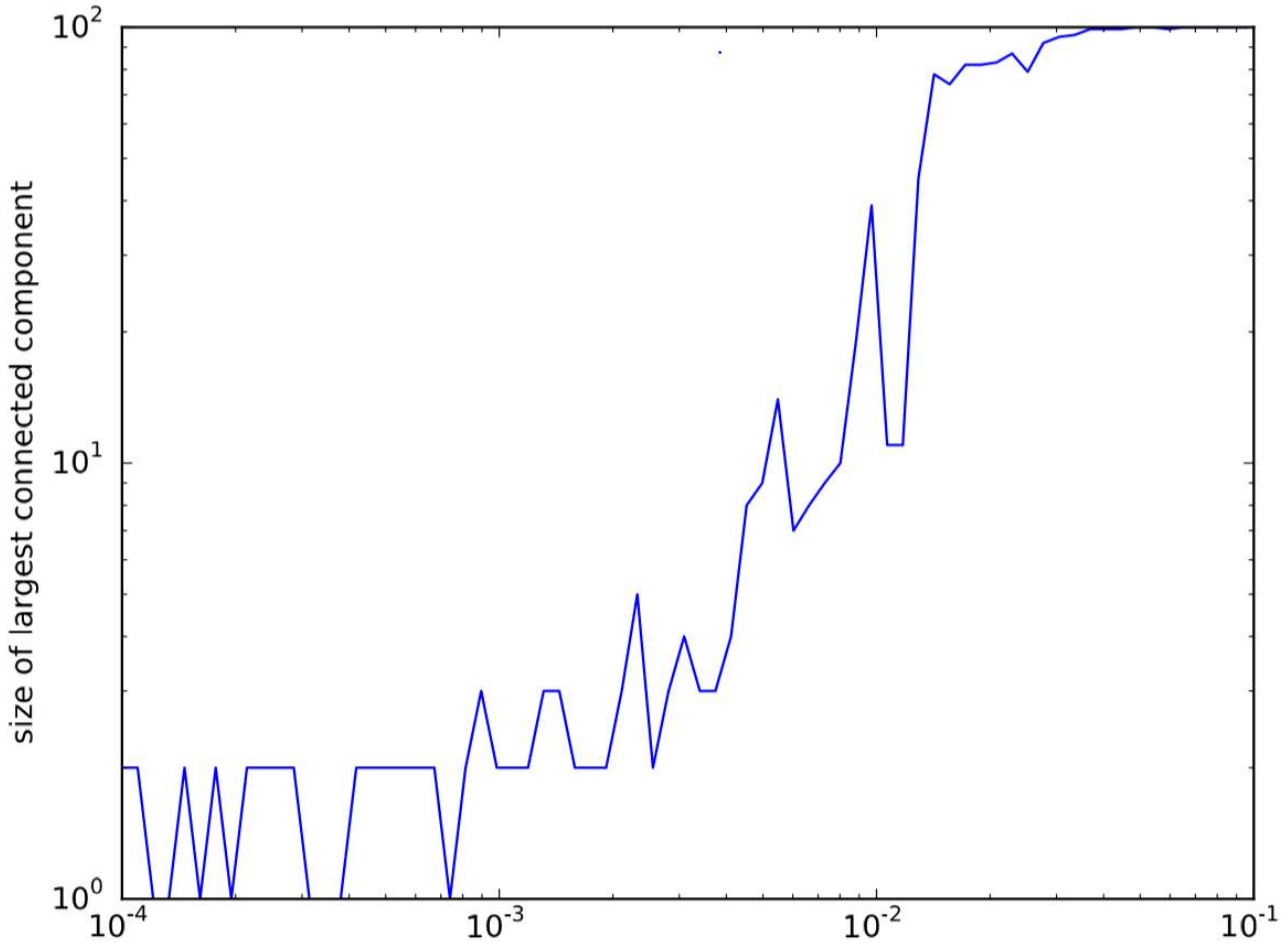


Figure 17.1.2: Visual output of Code 17.4.

Giant components are a special name given to the largest connected components that appear above this percolation threshold (they are seen in the third and fourth panels of Fig. 17.1.1), because their sizes are on the same order of magnitude as the size of the whole network. Mathematically speaking, they are defined as the connected components whose size $s(n)$ has the following property

$$\lim_{n\to\infty} \frac{s(n)}{c} = c > 0, \tag{17.1.3}$$

where n is the number of nodes. This limit would go to zero for all other non-giant components, so at macroscopic scales, we can only see giant components in large networks.

? Exercise 17.1.1

Revise Code 17.4 so that you generate multiple random graphs for each p, and calculate the average size of the largest connected components. Then run the revised code for larger network sizes (say, 1,000) to obtain a smoother curve.

We can calculate the network percolation threshold for random graphs as follows. Let q be the probability for a randomly selected node to not belong to the largest connected component (LCC) of a network. If $q < 1$, then there is a giant component in the network. In order for a randomly selected node to be disconnected from the LCC, all of its neighbors must be disconnected from the LCC too. Therefore, somewhat tautologically

$$q = q^k, \tag{17.1.4}$$

where k is the degree of the node in question. But in general, degree k is not uniquely determined in a random network, so the right hand side should be rewritten as an expected value, as

$$q = \sum_{k=0}^{\infty} P(k) q^k, \tag{17.1.5}$$

where $P(k)$ is the probability for the node to have degree k (called the degree distribution; this will be discussed in more detail later). In an Erd˝os-R´enyi random graph, this probability can be calculated by the following binomial distribution

$$P(k) = \begin{cases} \binom{n-1}{k} p^k (1-p)^{n-1-k} & \text{for } 0 \le k \le n-1, \\ 0 & \text{for } k \ge n, \end{cases} \tag{17.1.6}$$

because each node has $n-1$ potential neighbors and k out of $n-1$ neighbors need to be connected to the node (and the rest need to be disconnected from it) in order for it to have degree k. By plugging Equation 17.1.6 into Equation 17.1.5, we obtain

$$q = \sum_{k=0}^{n-1} \binom{n-1}{k} p^k (1-p)^{n-1-k} q^k \tag{17.1.7}$$

$$= \sum_{k=0}^{n-1} \binom{n-1}{k} (pq)^k (1-p)^{1-n-k} \tag{17.1.8}$$

$$= (pq + 1 - p)^{n-1} \tag{17.1.9}$$

$$= (1 + p(q-1))^{n-1}. \tag{17.1.10}$$

We can rewrite this as

$$q = (1 + \frac{\langle k \rangle (q-1)}{n})^{n-1}, \tag{17.1.11}$$

because the average degree $\langle k \rangle$ of an Erd˝os-R´enyi graph for a large n is given by

$$\langle k \rangle = np. \tag{17.1.12}$$

Since $\lim_{n\to\infty} (1 + x/n)^n = e^x$, Equation 17.1.11 can be further simplified for large n to

$$q = e^{\langle k \rangle (q-1)}. \tag{17.1.13}$$

Apparently, $q = 1$ satisfies this equation. If this equation also has another solution in $0 < q < 1$, then that means a giant component is possible in the network. Figure 17.1.3 shows the plots of $y = q$ and $y = e^{\langle k \rangle (q-1)}$ for $0 < q < 1$ for several values of $\langle k \rangle$.

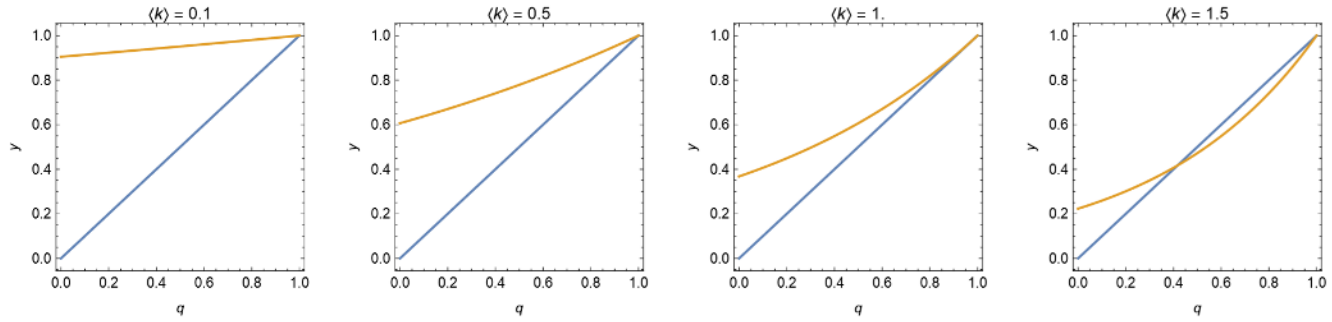


Figure 17.1.3: Plots of $y = q$ and $y = e^{\langle k \rangle (q-1)}$ for several values of $\langle k \rangle$.

These plots indicate that if the derivative of the right hand side of Equation 17.1.13 (i.e., the slope of the curve) at $q = 1$ is greater than 1, then the equation has a solution in $q < 1$. Therefore,

$$\frac{d}{dq} e^{\langle k \rangle (q-1)}|_{q=1} = \langle k \rangle e^{\langle k \rangle (q-1)}\_q = 1 = \langle k \rangle > 1, \tag{17.1.14}$$

i.e., if the average degree is greater than 1, then network percolation occurs.

A *giant component* is a connected component whose size is on the same order of magnitude as the size of the whole network. *Network percolation* is the appearance of such a giant component in a random graph, which occurs when the average node degree is above 1.

? Exercise 17.1.2

If Eq. (17.13) has a solution in $q < 1/n$, that means that all the nodes are essentially included in the giant component, and thus the network is made of a single connected component. Obtain the threshold of $\langle k \rangle$ above which this occurs.

This page titled 17.1: Network Size, Density, and Percolation is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.2: Shortest Path Length

Network analysis can measure and characterize various features of network topologies that go beyond size and density. Many of the tools used here are actually borrowed from social network analysis developed and used in sociology [60].

The first measurement we are going to discuss is the *shortest path length* from one node to another node, also called *geodesic distance* in a graph. If the network is undirected, the distance between two nodes is the same, regardless of which node is the starting point and which is the end point. But if the network is directed, this is no longer the case in general.

The shortest path length is easily measurable using NetworkX:

Code 17.5:

```
>>> g = nx.karate_club_graph()
>>> nx.shortest_path_length(g, 16, 25)
4
```

The actual path can also be obtained as follows:

Code 17.6:

```
>>> nx.shortest_path(g, 16, 25)
[16, 5, 0, 31, 25]
```

The output above is a list of nodes on the shortest path from node 16 to node 25. This can be visualized using `draw_networkx_edges` as follows:

Code 17.7: shortest-path.py

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()
positions = nx.spring_layout(g)

path = nx.shortest_path(g, 16, 25)
edges = [(path[i], path[i+1]) for i in xrange(len(path) - 1)]

nx.draw_networkx_edges(g, positions, edgelist = edges,
                       edge_color = 'r', width = 10)

nx.draw(g, positions, with_labels = True)
show()
```

The result is shown in Fig. 17.2.1.

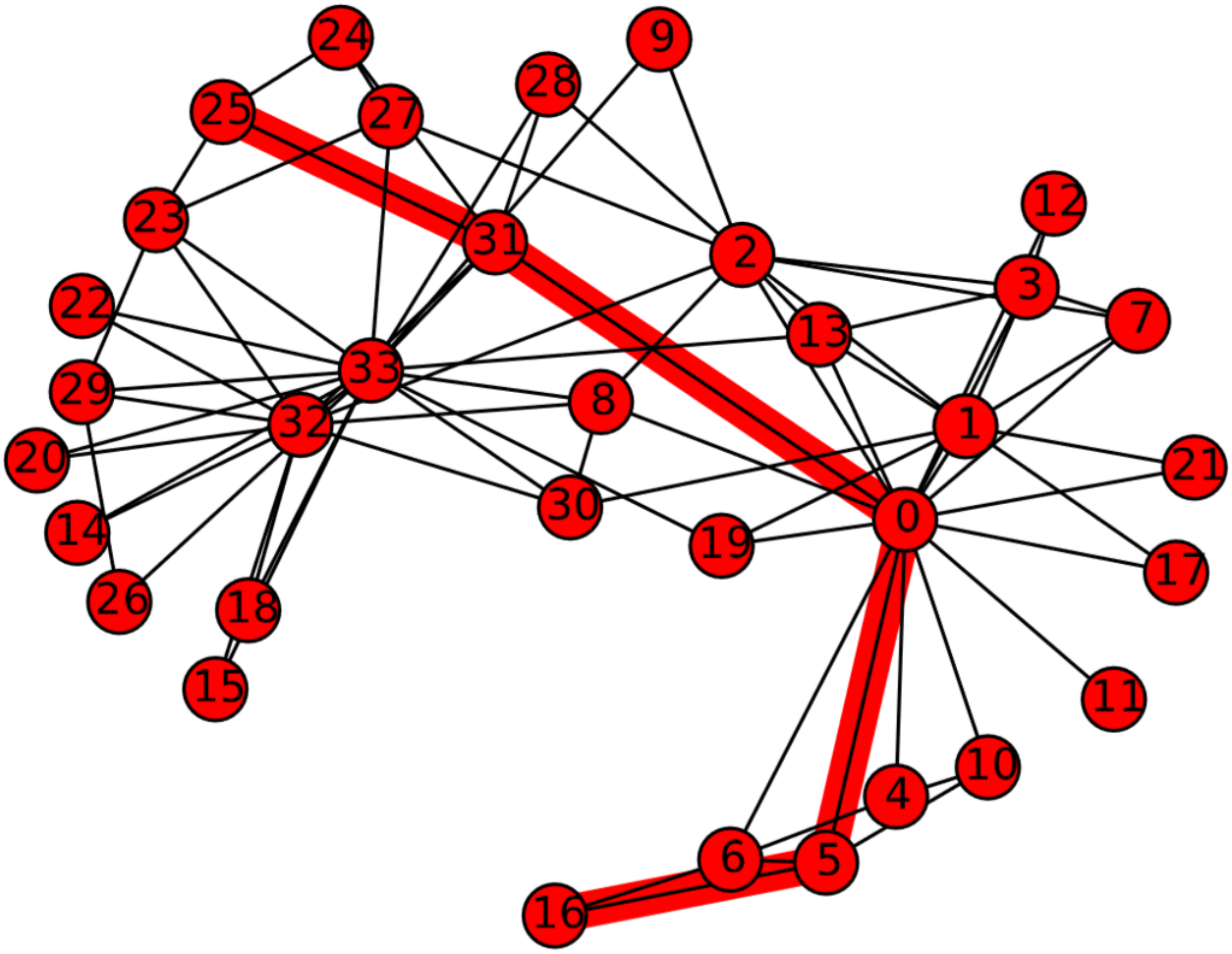


Figure 17.2.1: Visual output of Code 17.7.

We can use this shortest path length to define several useful metrics to characterize the network's topological properties. Let's denote a shortest path length from node i to node j as $d(i \to j)$. Clearly, $d(i \to i) = 0$ for all i. Then we can construct the following metrics:

Characteristic path length

$$L = \frac{\sum_{i,j} d(i \to j)}{n(n-1)} \tag{17.2.1}$$

where n is the number of nodes. This formula works for both undirected and directed networks. It calculates the average length of shortest paths for all possible node pairs in the network, giving an expected distance between two randomly chosen nodes. This is an intuitive characterization of how big (or small) the world represented by the network is.

Eccentricity

$$\epsilon(i) = \max_j d(1 \to j) \tag{17.2.2}$$

This metric is defined for each node and gives the maximal shortest path length a node can have with any other node in the network. This tells how far the node is to the farthest point in the network.

Diameter

$$D = \max_i \epsilon(i) \tag{17.2.3}$$

This metric gives the maximal eccentricity in the network. Intuitively, it tells us how far any two nodes can get from one another within the network. Nodes whose eccentricity is D are called *peripheries*.

Radius

$$R = \min_i \epsilon i \tag{17.2.4}$$

This metric gives the minimal eccentricity in the network. Intuitively, it tells us the smallest number of steps you will need to reach every node if you can choose an optimal node as a starting point. Nodes whose eccentricity is R are called *centers*.

In NetworkX, these metrics can be calculated as follows:

Code 17.8:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.average_shortest_path_length(g)
2.408199643493761
>>> nx.eccentricity(g)
{0: 3, 1: 3, 2: 3, 3: 3, 4: 4, 5: 4, 6: 4, 7: 4, 8: 3, 9: 4, 10: 4,
11: 4, 12: 4, 13: 3, 14: 5, 15: 5, 16: 5, 17: 4, 18: 5, 19: 3, 20: 5,
21: 4, 22: 5, 23: 5, 24: 4, 25: 4, 26: 5, 27: 4, 28: 4, 29: 5, 30: 4,
31: 3, 32: 4, 33: 4}
>>> nx.diameter(g)
5
>>> nx.periphery(g)
```

```
[14, 15, 16, 18, 20, 22, 23, 26, 29]
>>> nx.radius(g)
3
>>> nx.center(g)
[0, 1, 2, 3, 8, 13, 19, 31]
```

? Exercise 17.2.1

Visualize the Karate Club graph using the eccentricities of its nodes to color them.

? Exercise 17.2.2

Randomize the topology of the Karate Club graph while keeping its size and density, and then measure the characteristic path length, diameter, and radius of the randomized graph. Repeat this many times to obtain distributions of those metrics for randomized graphs. Then compare the distributions with the metrics of the original Karate Club graph. Discuss which metric is more sensitive to topological variations.

Note: Randomized graphs may be disconnected. If so, all of the metrics discussed above will be infinite, and NetworkX will give you an error. In order to avoid this, you should check whether the randomized network is connected or not before

> calculating its metrics. NetworkX has a function `nx.is_connected` for this purpose.

This page titled 17.2: Shortest Path Length is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.3: Centralities and Coreness

The eccentricity of nodes discussed above can be used to detect which nodes are most central in a network. This can be useful because, for example, if you send out a message from one of the center nodes with minimal eccentricity, the message will reach every single node in the network in the shortest period of time.

In the meantime, there are several other more statistically based measurements of node centralities, each of which has some benefits, depending on the question you want to study. Here is a summary of some of those centrality measures:

Degree centrality

$$c_D(i) = \frac{deg(i)}{n-1} \tag{17.3.1}$$

Degree centrality is simply a normalized node degree, i.e., the actual degree divided by the maximal degree possible $(n-1)$. For directed networks, you can define *in-degree centrality* and *out-degree centrality separately*.

Betweenness centrality

$$c_B(i) = \frac{1}{(n-1)(n-2)} \sum_{j \neq i, k \neq i, j \neq k} \frac{N_{sp}(j \to^i k)}{N_{sp}(j \to k)} \tag{17.3.2}$$

where $N_{sp}(j \to k)$ is the number of shortest paths from node j to node k, and $N_{sp}(j \to^i -k)$ is the number of the shortest paths from node j to node k that go through node i. Betweenness centrality of a node is the probability for the shortest path between two randomly chosen nodes to go through that node. This metric can also be defined for edges in a similar way, which is called *edge betweenness*.

Closeness centrality

$$C_c(i) = \left(\frac{\sum_j d(i \to j)}{n-1}\right)^{-1} \tag{17.3.3}$$

This is an inverse of the average distance from node i to all other nodes. If $c_C(i) = 1$, that means you can reach any other node from node i in just one step. For directed networks, you can also define another closeness centrality by swapping i and j in the formula above to measure how accessible node i is *from* other nodes.

Eigenvector centrality

$$c_E(i) = v_i \qquad (i-th\ element\ of\ the\ dominant\ eigenvector\ v\ of\ the\ networ\ k's\ adjacency\ matrix) \tag{17.3.4}$$

Eigenvector centrality measures the "importance" of each node by considering each incoming edge to the node an "endorsement" from its neighbor. This differs from degree centrality because, in the calculation of eigenvector centrality, endorsements coming from more important nodes count as more. Another completely different, but mathematically equivalent, interpretation of eigenvector centrality is that it counts the number of walks from any node in the network that reach node i in t steps, with t taken to infinity. Eigenvector v is usually chosen to be a non-negative unit vector $(v_i \geq 0, |v| = 1)$.

PageRank

$$c_P(i) = v_i \qquad (i-th\ element\ of\ the\ dominant\ eigenvector\ v\ of\ t\ he\ following\ transition\ probability\ matrix) \tag{17.3.5}$$

$$T = \alpha A D^{-1} + (1-\alpha)\frac{J}{n} \tag{17.3.6}$$

where A is the adjacency matrix of the network, D^{-1} is a diagonal matrix whose i-th diagonal component is $1/deg(i)$, J is an $n \times n$ all-one matrix, and α is the damping parameter ($\alpha = 0.85$ is commonly used by default).

PageRank is a variation of eigenvector centrality that was originally developed by Larry Page and Sergey Brin, the founders of Google, in the late 1990s [74, 75] to rank web pages. PageRank measures the asymptotic probability for a random walker on the network to be standing on node i, assuming that the walker moves to a randomly chosen neighbor with probability α, or jumps to any node in the network with probability $1-\alpha$, in each time step. Eigenvector v is usually chosen to be a probability distribution, i.e., $\sum_i v_i = 1$.

Note that all of these centrality measures give a normalized value between 0 and 1, where 1 means perfectly central while 0 means completely peripheral. In most cases, the values are somewhere in between. Functions to calculate these centrality measures are also available in NetworkX (outputs are omitted in the code below to save space):

Code 17.9:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.degree_centrality(g)
>>> nx.betweenness_centrality(g)
>>> nx.closeness_centrality(g)
>>> nx.eigenvector_centrality(g)
>>> nx.pagerank(g)
```

While those centrality measures are often correlated, they capture different properties of each node. Which centrality should be used depends on the problem you are addressing. For example, if you just want to find the most popular person in a social network, you can just use degree centrality. But if you want to find the most efficient person to disseminate a rumor to the entire network, closeness centrality would probably be a more appropriate metric to use. Or, if you want to find the most effective person to monitor and manipulate information flowing within the network, betweenness centrality would be more appropriate, assuming that information travels through the shortest paths between people. Eigenvector centrality and PageRank are useful for generating a reasonable ranking of nodes in a complex network made of directed edges.

? Exercise 17.3.1

Visualize the Karate Club graph using each of the above centrality measures to color the nodes, and then compare the visualizations to see how those centralities are correlated.

? Exercise 17.3.2

Generate (1) an Erd˝os-R´enyi random network, (2) a Watts-Strogat zmall-world network, and (3) a Barab´asi-Albert scale-free network of comparable size and density, and obtain the distribution of node centralities of your choice for each network. Then compare those centrality distributions to find which one is most/least heterogeneous.

? Exercise 17.3.3

Prove that PageRank with $\alpha = 1$ is essentially the same as the degree centrality for undirected networks.

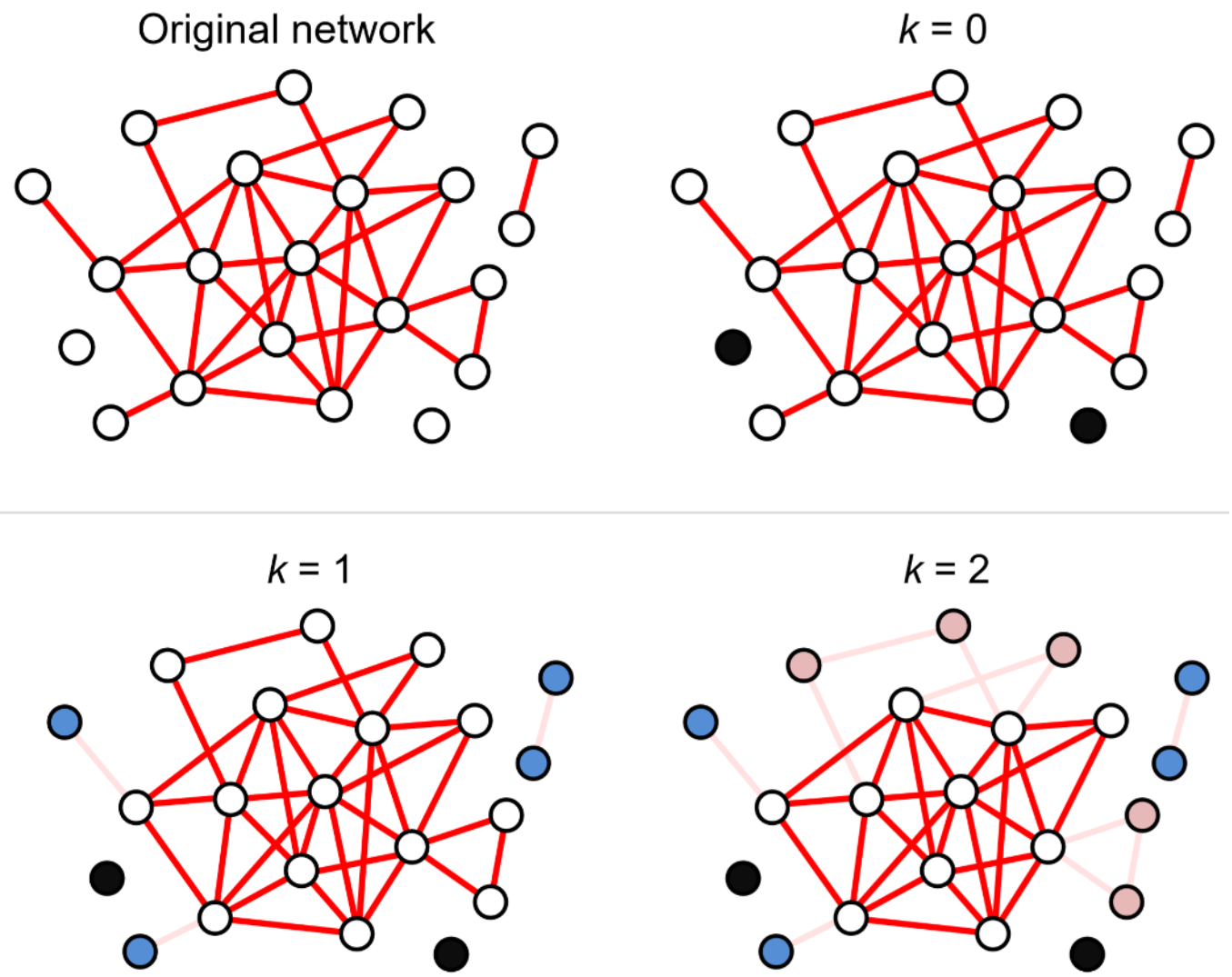
A little different approach to characterize the centrality of nodes is to calculate their *coreness*. This can be achieved by the following simple algorithm:

1. Let $k = 0$.
2. Repeatedly delete all nodes whose degree is k or less, until no such nodes exist. Those removed nodes are given a coreness k.

3. If there are still nodes remaining in the network, increase k by 1, and then go back to the previous step.

Figure 17.3.1 shows an example of this calculation. At the very end of the process, we see a cluster of nodes whose degrees are at least the final value of k. This cluster is called a k-core, which can be considered the central part of the network.

In NetworkX, you can calculate the corenesses of nodes using the `core_number function` . Also, the k-core of the network can be obtained by using the `k_core function` . Here is an example:



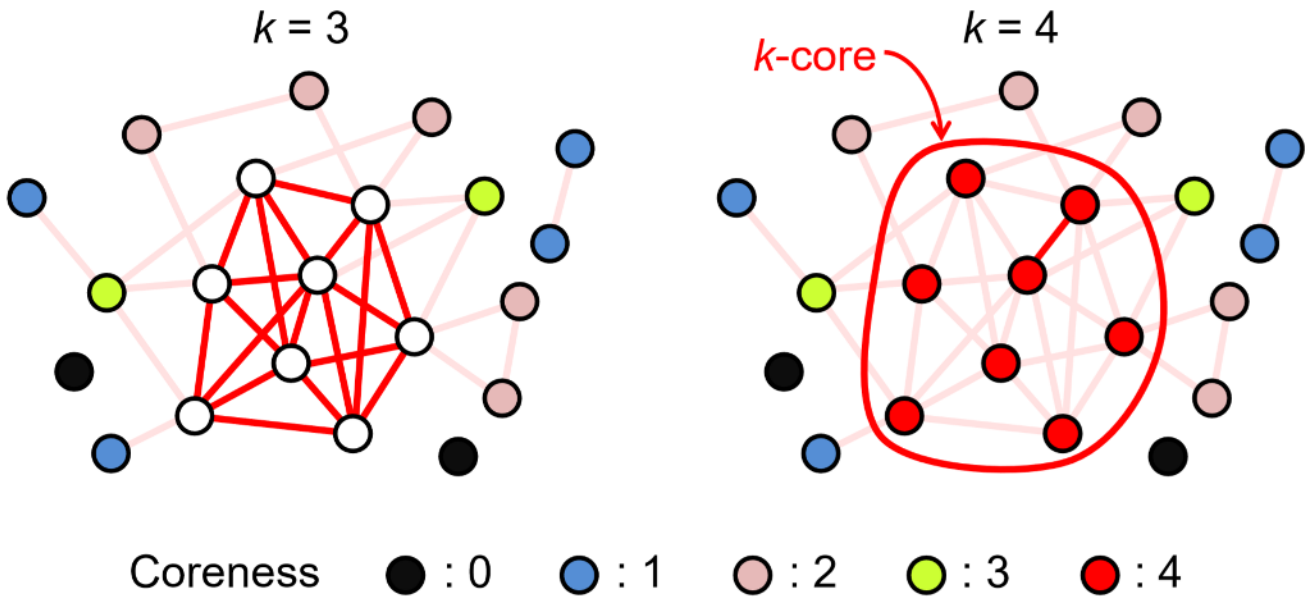


Figure **17.3.1**: Example of how coreness is calculated.

Code 17.10:

```
>>> from pylab import *
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.core_number(g)
{0: 4, 1: 4, 2: 4, 3: 4, 4: 3, 5: 3, 6: 3, 7: 4, 8: 4, 9: 2, 10: 3,
  11: 1, 12: 2, 13: 4, 14: 2, 15: 2, 16: 2, 17: 2, 18: 2, 19: 3,
  20: 2, 21: 2, 22: 2, 23: 3, 24: 3, 25: 3, 26: 2, 27: 3, 28: 3,
  29: 3, 30: 4, 31: 3, 32: 4, 33: 4}
>>> nx.draw(nx.k_core(g), with_labels = True)
>>> show()
```

The resulting k-core of the Karate Club graph is shown in Fig. 17.3.2.

https://math.libretexts.org/@go/page/7873

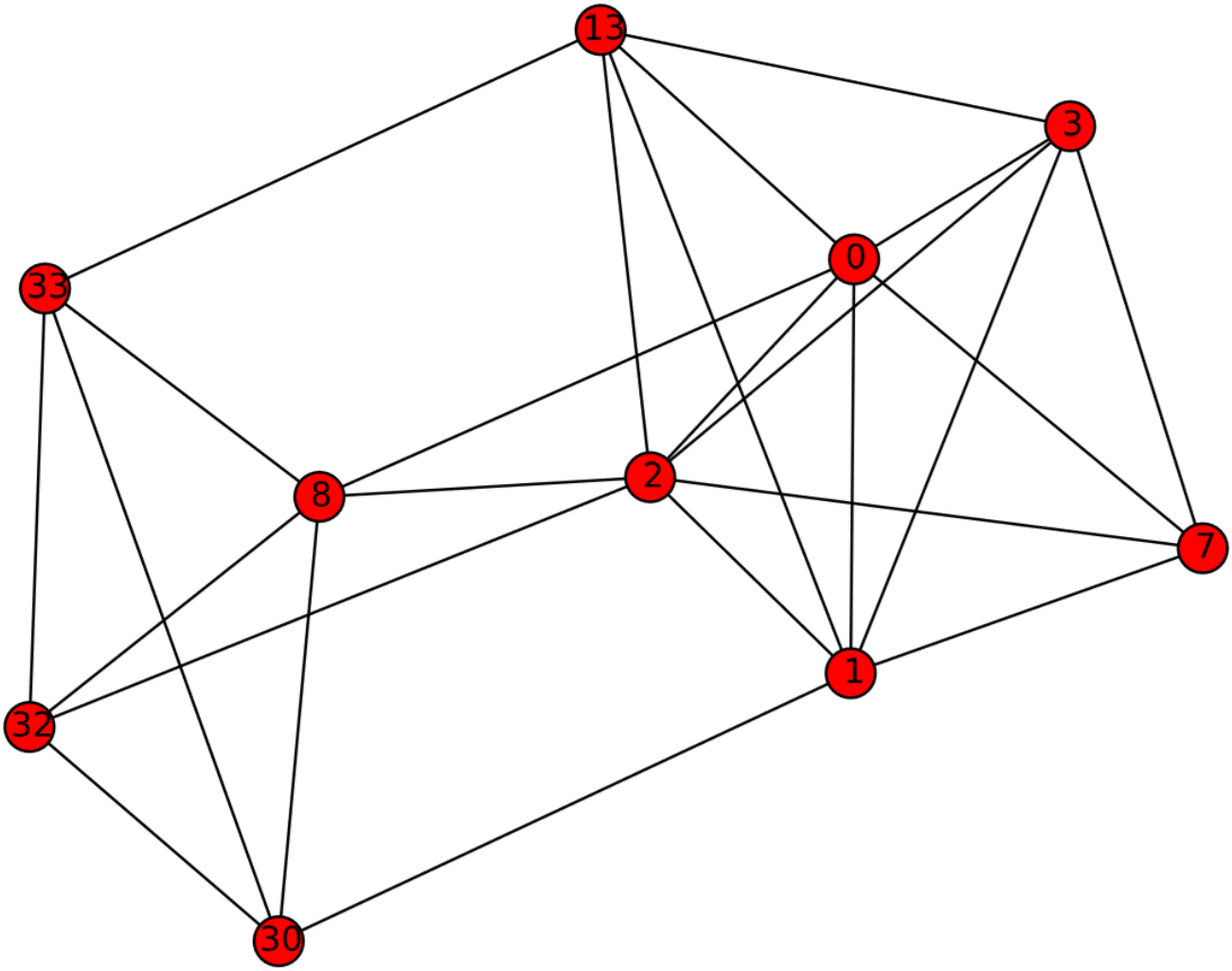


Figure **17.3.2**: Visual output of Code 17.10, showing the k-core of the Karate Club graph, with $k = 4$.

One advantage of using coreness over other centrality measures is its scalability. Because the algorithm to compute it is so simple, the calculation of coreness of nodes in a very large network is much faster than other centrality measures (except for degree centrality, which is obviously very fast too).

? Exercise 17.3.4

Import a large network data set of your choice from Mark Newman's Network Data website: http://www-personal.umich.edu/~mejn/netdata/

Calculate the coreness of all of its nodes and draw their histogram. Compare the speed of calculation with, say, the calculation of betweenness centrality. Also visualize the k-core of the network.

This page titled 17.3: Centralities and Coreness is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.4: Clustering

Eccentricity, centralities, and coreness introduced above all depend on the whole network topology (except for degree centrality). In this sense, they capture some macroscopic aspects of the network, even though we are calculating those metrics for each node. In contrast, there are other kinds of metrics that only capture local topological properties. This includes metrics of clustering, i.e., how densely connected the nodes are to each other in a localized area in a network. There are two widely used metrics for this:

Clustering coefficient

$$C(i) = \frac{|\{\{j,k\}|d(i,j) = d(i,k) = d(j,k) = 1\}|}{deg(i)(deg(i)-1)/2} \tag{17.4.1}$$

The denominator is the total number of possible node pairs within node i's neighborhood, while the numerator is the number of actually connected node pairs among them. Therefore, the clustering coefficient of node i calculates the probability for its neighbors to be each other's neighbors as well. Note that this metric assumes that the network is undirected. The following average clustering coefficient is often used to measure the level of clustering in the entire network:

$$C = \frac{\sum_i C(i)}{n} \tag{17.4.2}$$

Transitivity

$$C_T = \frac{|\{(i,j,k)|d(i,j) = d(i,k) = d(j,k) = 1\}|}{|\{(i,j,k)|d(i,j) = d(i,k) = 1|\}} \tag{17.4.3}$$

This is very similar to clustering coefficients, but it is defined by counting connected node triplets over the entire network. The denominator is the number of connected node triplets (i.e., a node, i, and two of its neighbors, j and k), while the numerator is the number of such triplets where j is also connected to k. This essentially captures the same aspect of the network as the average clustering coefficient, i.e., how locally clustered the network is, but the transitivity can be calculated on directed networks too. It also treats each triangle more evenly, unlike the average clustering coefficient that tends to underestimate the contribution of triplets that involve highly connected nodes.

Again, calculating these clustering metrics is very easy in NetworkX:

Code 17.11:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.clustering(g)
{0: 0.15, 1: 0.3333333333333333, 2: 0.24444444444444444, 3:
  0.6666666666666666, 4: 0.6666666666666666, 5: 0.5, 6: 0.5, 7: 1.0,
  8: 0.5, 9: 0.0, 10: 0.6666666666666666, 11: 0.0, 12: 1.0, 13: 0.6,
  14: 1.0, 15: 1.0, 16: 1.0, 17: 1.0, 18: 1.0, 19: 0.3333333333333333,
  20: 1.0, 21: 1.0, 22: 1.0, 23: 0.4, 24: 0.3333333333333333, 25:
  0.3333333333333333, 26: 1.0, 27: 0.16666666666666666, 28:
  0.3333333333333333, 29: 0.6666666666666666, 30: 0.5, 31: 0.2, 32:
  0.19696969696969696, 33: 0.11029411764705882}
>>> nx.average_clustering(g)
0.5706384782076823
>>> nx.transitivity(g)
0.2556818181818182
```

? Exercise 17.4.1

Generate (1) an Erd˝os-R´enyi random network, (2) a WattsStrogatz small-world network, and(3) a Barab´asi-Albert scale-free network of comparable size and density, and compare them with regard to how locally clustered they are.

The clustering coefficient was first introduced by Watts and Strogatz [56], where they showed that their small-world networks tend to have very high clustering compared to their random counterparts. The following code replicates their computational experiment, varying the rewiring probability p:

Code 17.12: small-world-experiment.py

```
from pylab import *
import networkx as nx

pdata = []
Ldata = []
Cdata = []

g0 = nx.watts_strogatz_graph(1000, 10, 0)
L0 = nx.average_shortest_path_length(g0)
C0 = nx.average_clustering(g0)

p = 0.0001
while p < 1.0:
    g = nx.watts_strogatz_graph(1000, 10, p)
    pdata.append(p)
    Ldata.append(nx.average_shortest_path_length(g) / L0)
    Cdata.append(nx.average_clustering(g) / C0)
    p *= 1.5

semilogx(pdata, Ldata, label = 'L / L0')
semilogx(pdata, Cdata, label = 'C / C0')
xlabel('p')
legend()
show()
```

The result is shown in Fig. 17.4.1, where the characteristic path length `(L)` and the average clustering coefficient `(C)` are plotted as their fractions to the baseline values `(L0, C0)` obtained from a purely regular network `g0`. As you can see in the figure, the network becomes very small (low `L`) yet remains highly clustered (high `C`) at the intermediate value of `p` around 10^{-2}. This is the parameter regime where the Watts-Strogatz small-world networks arise

***Figure* 17.4.1**: Visual output of Code 17.12.

This page titled 17.4: Clustering is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.5: Degree Distribution

Another local topological property that can be measured locally is, as we discussed already, the degree of a node. But if we collect them all for the whole network and represent them as a distribution, it will give us another important piece of information about how the network is structured:

A *degree distribution* of a network is a probability distribution

$$P(k) = \frac{|\{i|deg(i) = k\}|}{n}. \tag{17.5.1}$$

i.e., the probability for a node to have degree k.

The degree distribution of a network can be obtained and visualized as follows:

Code 17.13:

```
>>> from pylab import *
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> hist(g.degree().values(), bins = 20)
```

```
(array([ 1., 11., 6., 6., 0., 3., 2., 0., 0., 0., 1., 1., 0., 1., 0.,
  0., 0., 0., 1., 1.]), array([ 1. , 1.8, 2.6, 3.4, 4.2, 5. , 5.8,
  6.6, 7.4, 8.2, 9. , 9.8, 10.6, 11.4, 12.2, 13. , 13.8, 14.6, 15.4,
  16.2, 17. ]), <a list of 20 Patch objects>)
>>> show()
```

The result is shown in Fig. 17.5.1.

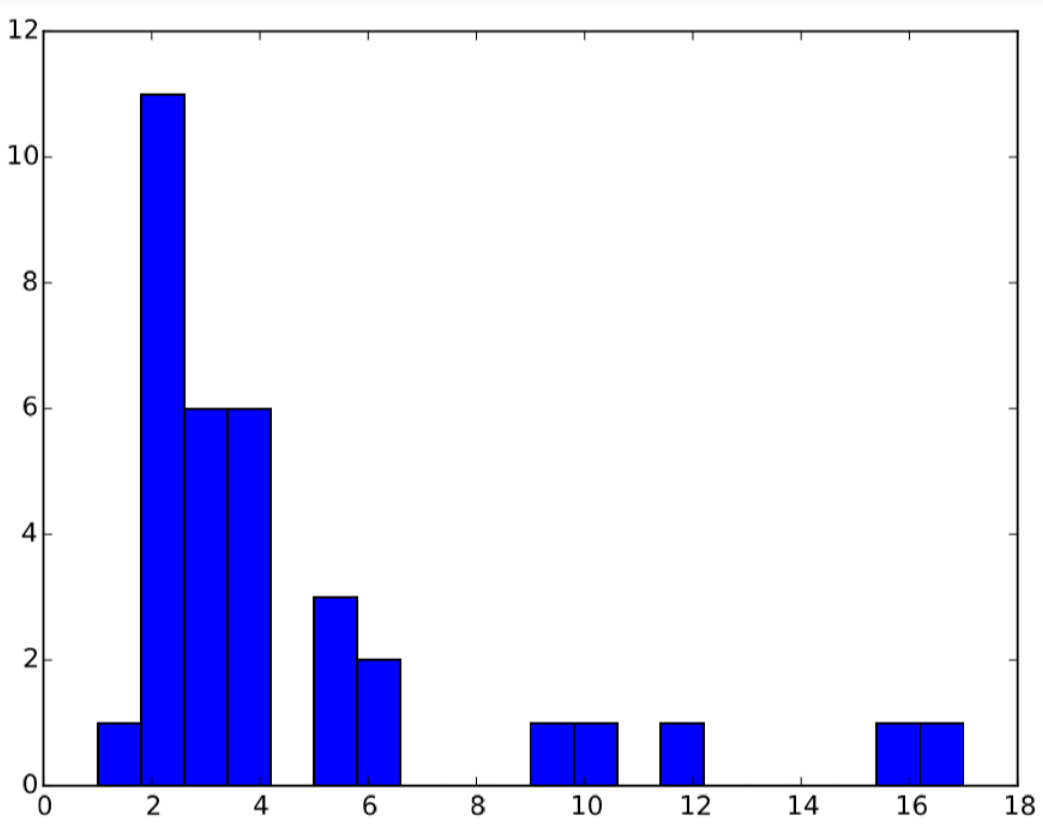


Figure 17.5.1: Visual output of Code 17.13.

You can also obtain the actual degree distribution P(k) as follows:

Code 17.14:

```
>>> nx.degree_histogram(g)
[0, 1, 11, 6, 6, 3, 2, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1]
```

This list contains the value of (unnormalized) $P(k) for k = 0, 1, \ldots, k_{max}$, in this order. For larger networks, it is often more useful to plot a normalized degree histogram list in a log-log scale:

Code 17.15: degree-distributions-loglog.py

```
from pylab import *
import networkx as nx

n = 1000
```

```
er = nx.erdos_renyi_graph(n, 0.01)
ws = nx.watts_strogatz_graph(n, 10, 0.01)
ba = nx.barabasi_albert_graph(n, 5)

Pk = [float(x) / n for x in nx.degree_histogram(er)]
domain = range(len(Pk))
loglog(domain, Pk, '-', label = 'Erdos-Renyi')

Pk = [float(x) / n for x in nx.degree_histogram(ws)]
domain = range(len(Pk))
loglog(domain, Pk, '--', label = 'Watts-Strogatz')

Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
loglog(domain, Pk, ':', label = 'Barabasi-Albert')
```

```
xlabel('k')
ylabel('P(k)')
legend()
show()
```

The result is shown in Fig. 17.5.2, which clearly illustrates differences between the three network models used in this example. The Erd˝os-R´enyi random network model has a bell-curved degree distribution, which appears as a skewed mountain in the log-log scale (blue solid line). The Watts-Strogatz model is nearly regular, and thus it has a very sharp peak at the average degree (green dashed line; \(k = 10\0 in this case). The Barab´asi-Albert model has a power-law degree distribution, which looks like a straight line with a negative slope in the log-log scale (red dotted line).

Moreover, it is often visually more meaningful to plot not the degree distribution itself but its *complementary cumulative distribution function (CCDF)*, defined as follows:

$$F(k) = \sum_{k'=k}^{\infty} P(k') \tag{17.5.2}$$

This is a probability for a node to have a degree k or higher. By definition, $F(0) = 1$ and $F(k_{max} + 1) = 0$, and the function decreases monotonically along k. We can revise Code 17.15 to draw CCDFs:

Code 17.16: ccdfs-loglog.py

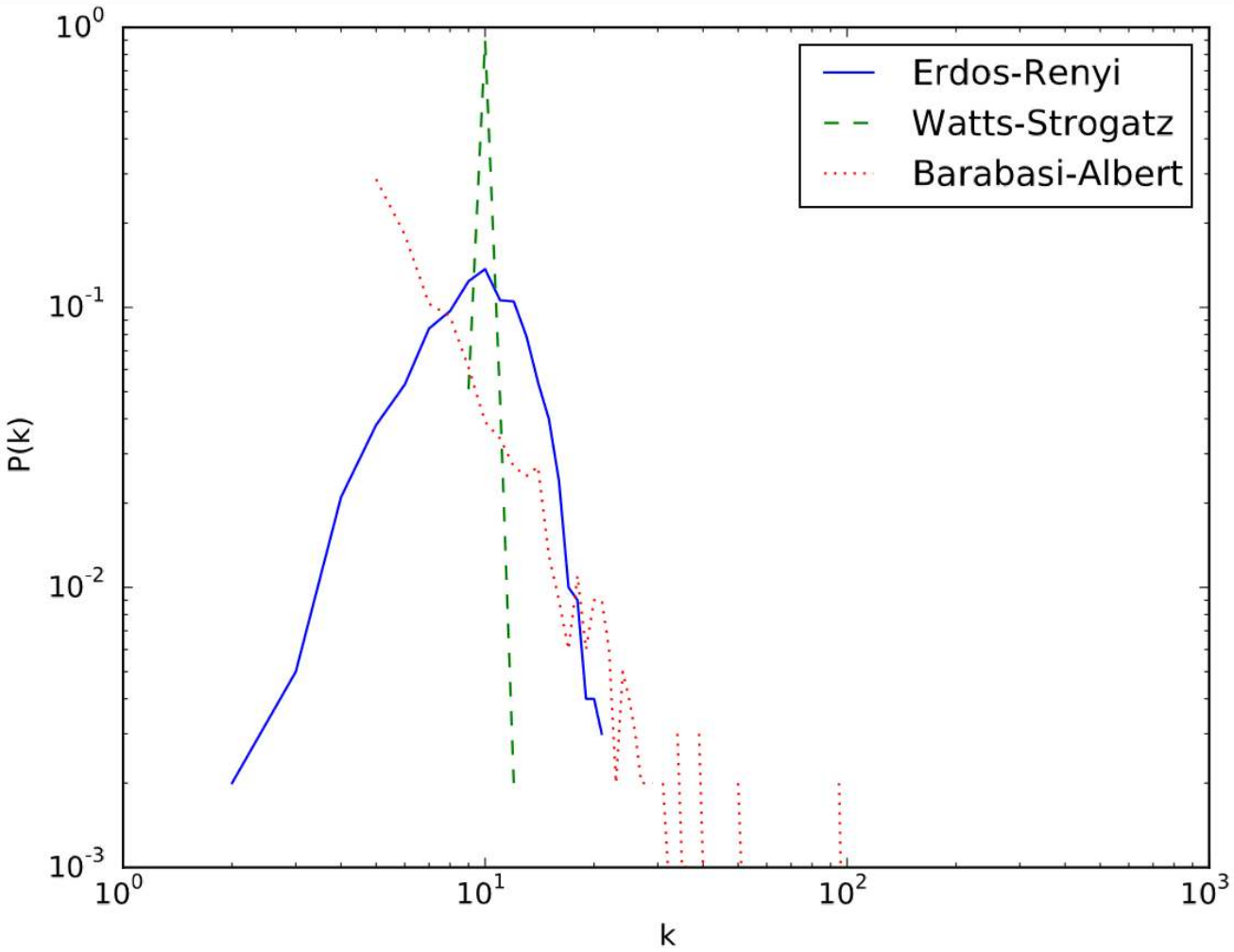


Figure 17.5.2: Visual output of Code 17.15.

```
from pylab import *
import networkx as nx

n = 1000
er = nx.erdos_renyi_graph(n, 0.01)
ws = nx.watts_strogatz_graph(n, 10, 0.01)
ba = nx.barabasi_albert_graph(n, 5)

Pk = [float(x) / n for x in nx.degree_histogram(er)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, '-', label = 'Erdos-Renyi')

Pk = [float(x) / n for x in nx.degree_histogram(ws)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, '--', label = 'Watts-Strogatz')

Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, ':', label = 'Barabasi-Albert')

xlabel('k')
ylabel('F(k)')
legend()
show()
```

In this code, we generate `ccdf`'s from `Pk` by calculating the sum of `Pk` after dropping its first `k` entries. The result is shown in Fig. 17.5.2.

As you can see in the figure, the power law degree distribution remains as a straight line in the CCDF plot too, because $F(k)$ will still be a power function of k, as shown below:

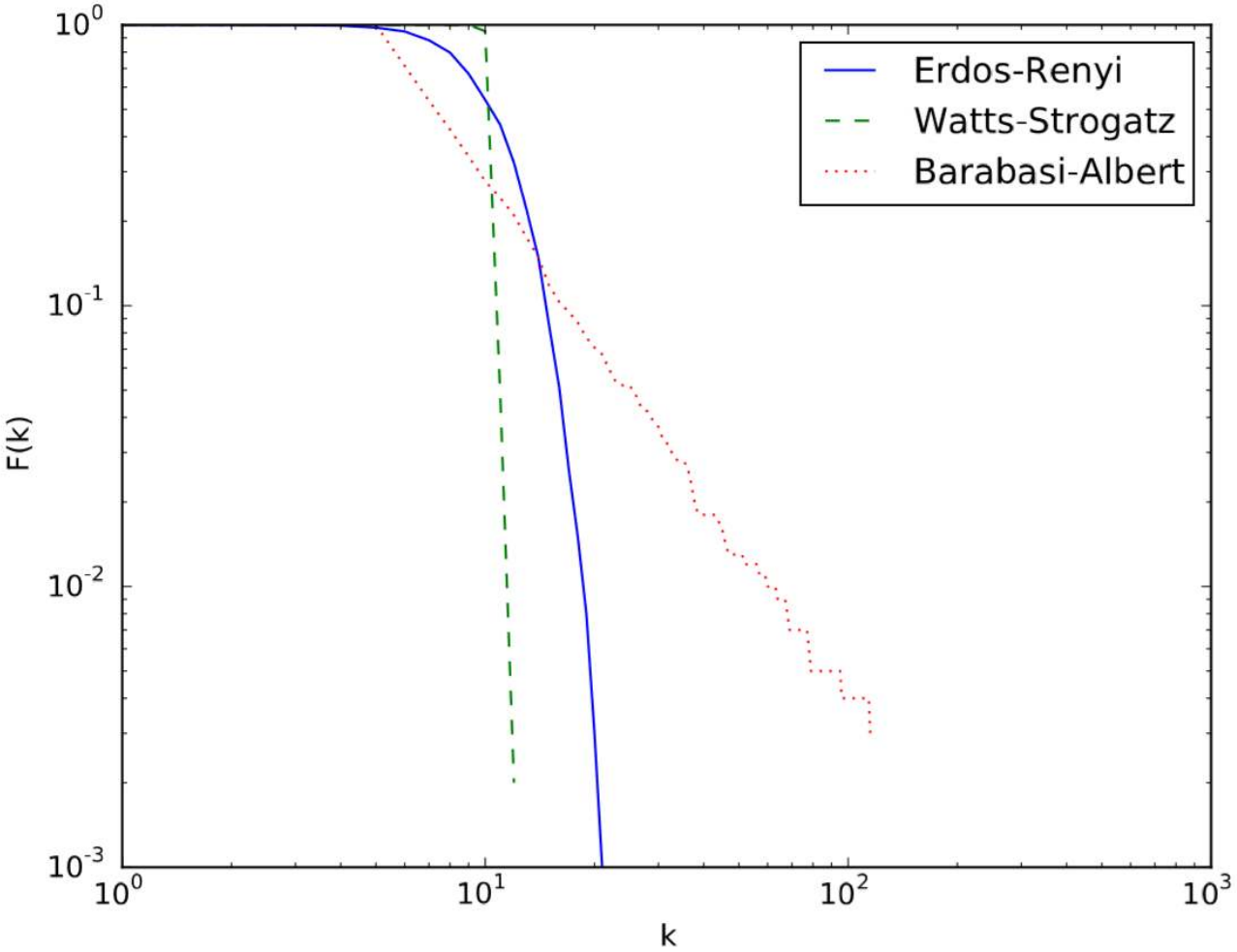


Figure **17.5.3**: Visual output of Code 17.16.

Erroneous nesting of equation structures

This result shows that the scaling exponent of $F(k)$ for a power law degree distribution is less than that of the original distribution by 1, which can be visually seen by comparing their slopes between Figs. 17.5.2 and 17.5.3.

? Exercise 17.5.1

Import a large network data set of your choice from Mark Newman's Network Data website: http://www-personal.umich.edu/~mejn/netdata/

Plot the degree distribution of the network, as well as its CCDF. Determine whether the network is more similar to a random, a small-world, or a scale-free network model.

If the network's degree distribution shows a power law behavior, you can estimate its scaling exponent from the distribution by simple linear regression. You should use a CCDF of the degree distribution for this purpose, because CCDFs are less noisy than the original degree distributions. Here is an example of scaling exponent estimation applied to a Barab´asi-Albert network, where the `linregress` function in SciPy's `stats` module is used for linear regression:

Code 17.17: exponent-estimation.py

```
from pylab import *
import networkx as nx
from scipy import stats as st

n = 10000
ba = nx.barabasi_albert_graph(n, 5)
Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
```

```
logkdata = []
logFdata = []
prevF = ccdf[0]
for k in domain:
    F = ccdf[k]
    if F != prevF:
        logkdata.append(log(k))
        logFdata.append(log(F))
        prevF = F

a, b, r, p, err = st.linregress(logkdata, logFdata)
print 'Estimated CCDF: F(k) =', exp(b), '* k^', a
print 'r =', r
print 'p-value =', p

plot(logkdata, logFdata, 'o')
kmin, kmax = xlim()
plot([kmin, kmax],[a * kmin + b, a * kmax + b])
xlabel('log k')
ylabel('log F(k)')
show()
```

In the second code block, the `domain` and `ccdf` were converted to log scales for linear fitting. Also, note that the original `ccdf` contained values for all `k`' s, even for those for which $P(k) = 0$. This would cause unnecessary biases in the linear regression toward the higher `k` end where actual samples were very sparse. To avoid this, only the data points where the value of F changed (i.e., where there were actual nodes with degree `k`) are collected in the `logkdata` and `logFdata` lists.

The result is shown in Fig. 17.5.4, and also the following output comes out to the terminal, which indicates that this was a pretty good fit:

Code 17.18:

```
Estimated CCDF: F(k) = 27.7963518947 * k^ -1.97465957944
r = -0.997324657337
p-value = 8.16476416778e-127
```

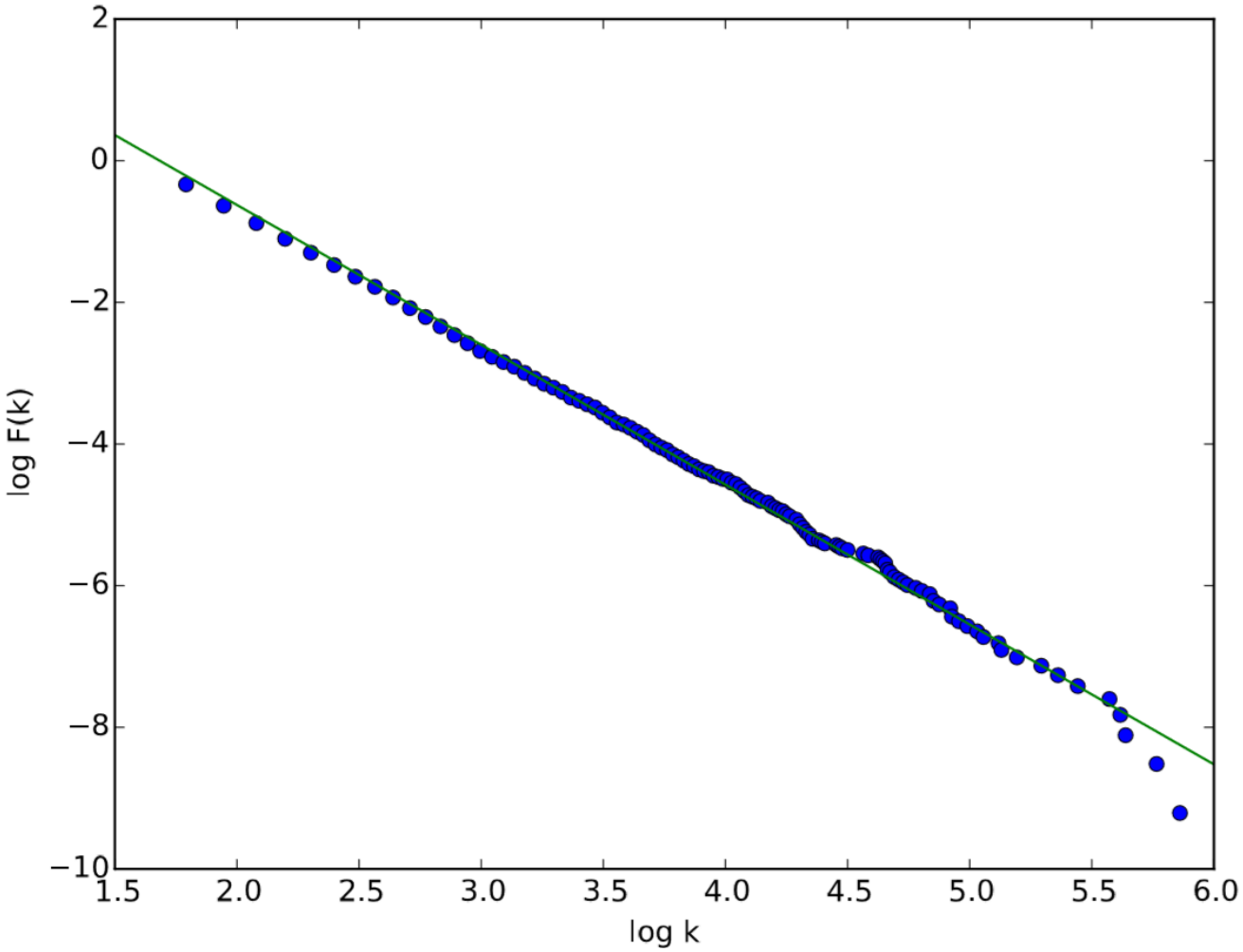


Figure **17.5.4**: Visual output of Code 17.17.

According to this result, the CCDF had a negative exponent of about -1.97. Since this value corresponds to $-(\gamma-1)$, the actual scaling exponent γ is about 2.97, which is pretty close to its theoretical value, 3.

? Exercise 17.5.2

Obtain a large network data set whose degree distribution appears to follow a power law, from any source (there are tons available online, including Mark Newman's that was introduced before). Then estimate its scaling exponent using linear regression.

This page titled 17.5: Degree Distribution is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

17.6: Assortativity

Degrees are a metric measured on individual nodes. But when we focus on the edges, there are always two degrees associated with each edge, one for the node where the edge originates and the other for the node to where the edge points. So if we take the former for x and the latter for y from all the edges in the network, we can produce a scatter plot that visualizes a possible degree correlation between the nodes across the edges. Such correlations of node properties across edges can be generally described with the concept of assortativity:

Assortativity (positive assortativity) The tendency for nodes to connect to other nodes with similar properties within a network.
Disassortativity (negative assortativity) The tendency for nodes to connect to other nodes with dissimilar properties within a network.

Assortativity coefficient

$$r = \frac{\sum_{(i,j)\epsilon E}(f(i) - \bar{f}_1)(f(j) - \bar{f}_2)}{\sqrt{\sum_{(i,j)\epsilon E}(f(j) - \bar{f}_2)}\sqrt{\sum_{(i,j)\epsilon E}(f(j) - \bar{f}_2)^2}} \tag{17.6.1}$$

where E is the set of directed edges (undirected edges should appear twice in E in two directions), and

$$\bar{f}_1 = \frac{\sum_{i,j}\epsilon E f(i)}{|E|}, \qquad \bar{f}_2 = \frac{\sum_{i,j}\epsilon E f(j)}{|E|}. \tag{17.6.2}$$

The assortativity coefficient is a Pearson correlation coefficient of some node property f between pairs of connected nodes. Positive coefficients imply assortativity, while negative ones imply disassortativity.

If the measured property is a node degree (i.e., $f = deg$), this is called the *degree assortativity coefficient*. For directed networks, each of $\bar{f}_1$ an $\bar{f}_2$ can be either in-degree or out-degree, so there are four different degree assortativities you can measure: *in-in*, *in-out*, *out-in*, and *out-out*.

Let's look at some example. Here is how to draw a degree-degree scatter plot:

Code 17.19: degree-correlation.py

```
from pylab import *
import networkx as nx

n = 1000
ba = nx.barabasi_albert_graph(n, 5)

xdata = []
ydata = []
for i, j in ba.edges_iter():
    xdata.append(ba.degree(i)); ydata.append(ba.degree(j))
```

```
    xdata.append(ba.degree(j)); ydata.append(ba.degree(i))

plot(xdata, ydata, 'o', alpha = 0.05)
show()
```

In this example, we draw a degree-degree scatter plot for a Barab´asi-Albert network with 1,000 nodes. For each edge, the degrees of its two ends are stored in `xdata` and `ydata` twice in different orders, because an undirected edge can be counted in two directions. The markers in the plot are made transparent using the `alpha` option so that we can see the density variations in the plot.

The result is shown in Fig. 17.6.1, where each dot represents one directed edge in the network (so, an undirected edge is represented by two dots symmetrically placed across a diagonal mirror line). It can be seen that most edges connect low-degree nodes to each other, with some edges connecting low-degree and high-degree nodes, but it is quite rare that high-degree nodes are connected to each other. Therefore, there is a mild negative degree correlation in this case.

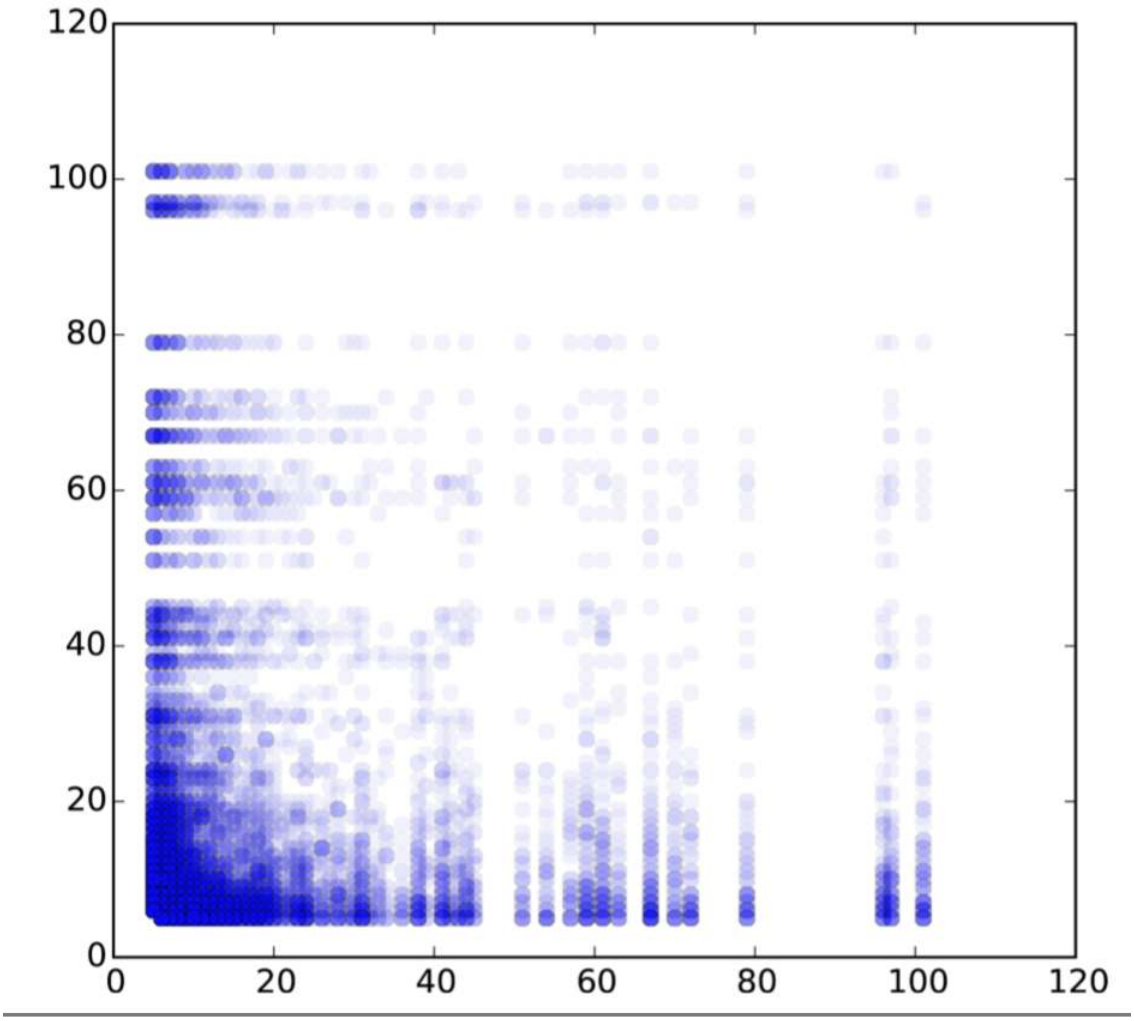


Figure* 17.6.1*: Visual output of code 17.19.

We can confirm this observation by calculating the degree assortativity coefficient as follows:

https://math.libretexts.org/@go/page/7876

Code 17.20:

```
>>> nx.degree_assortativity_coefficient(ba)
-0.058427968508962938
```

This function also has options x (for $\bar{f}_1$) and y (for $\bar{f}_2$) to specify which degrees you use in calculating coefficients for directed networks:

Code 17.21:

```
>>> import networkx as nx
>>> g = nx.DiGraph()
>>> g.add_edges_from([(0,1), (0,2), (0,3), (1,2), (2,3), (3,0)])
>>> nx.degree_assortativity_coefficient(g, x = 'in', y = 'in')
-0.250000000000001
>>> nx.degree_assortativity_coefficient(g, x = 'in', y = 'out')
0.63245553203367555
>>> nx.degree_assortativity_coefficient(g, x = 'out', y = 'in')
0.0
>>> nx.degree_assortativity_coefficient(g, x = 'out', y = 'out')
-0.44721359549995793
```

There are also other functions that can calculate assortativities of node properties other than degrees. Check out NetworkX's online documentation for more details.

? Exercise 17.6.1

Measure the degree assortativity coefficient of the Karate Club graph. Explain the result in view of the actual topology of the graph.

It is known that real-world networks show a variety of assortativity. In general, social networks of human individuals, such as collaborative relationships among scientists or corporate directors, tend to show positive assortativity, while technological networks (power grid, the Internet, etc.) and biological networks (protein interactions, neural networks, food webs, etc.) tend to show negative assortativity [76].

In the meantime, it is also known that scale-free networks of a finite size (which many real-world networks are) naturally show negative disassortativity purely because of inherent structural limitations, which is called a *structural cutoff* [25]. Such disassortativity arises because there are simply not enough hub nodes available for themselves to connect to each other to maintain assortativity. This means that the positive assortativities found in human social networks indicate there is definitely some assortative mechanism driving their self-organization, while the negative assortativities found in technological and biological networks may be explained by this simple structural reason. In order to determine whether or not a network showing negative assortativity is fundamentally disassortative for non-structural reasons, you will need to conduct a control experiment by randomizing its topology and measuring assortativity while keeping the same degree distribution.

? Exercise 17.6.2

Randomize the topology of the Karate Club graph while keeping its degree sequence, and then measure the degree assortativity coefficient of the randomized graph. Repeat this many times to obtain a distribution of the coefficients for randomized graphs. Then compare the distribution with the actual assortativity of the original Karate Club graph. Based on the result, determine whether or not the Karate Club graph is truly assortative or disassortative.

This page titled 17.6: Assortativity is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

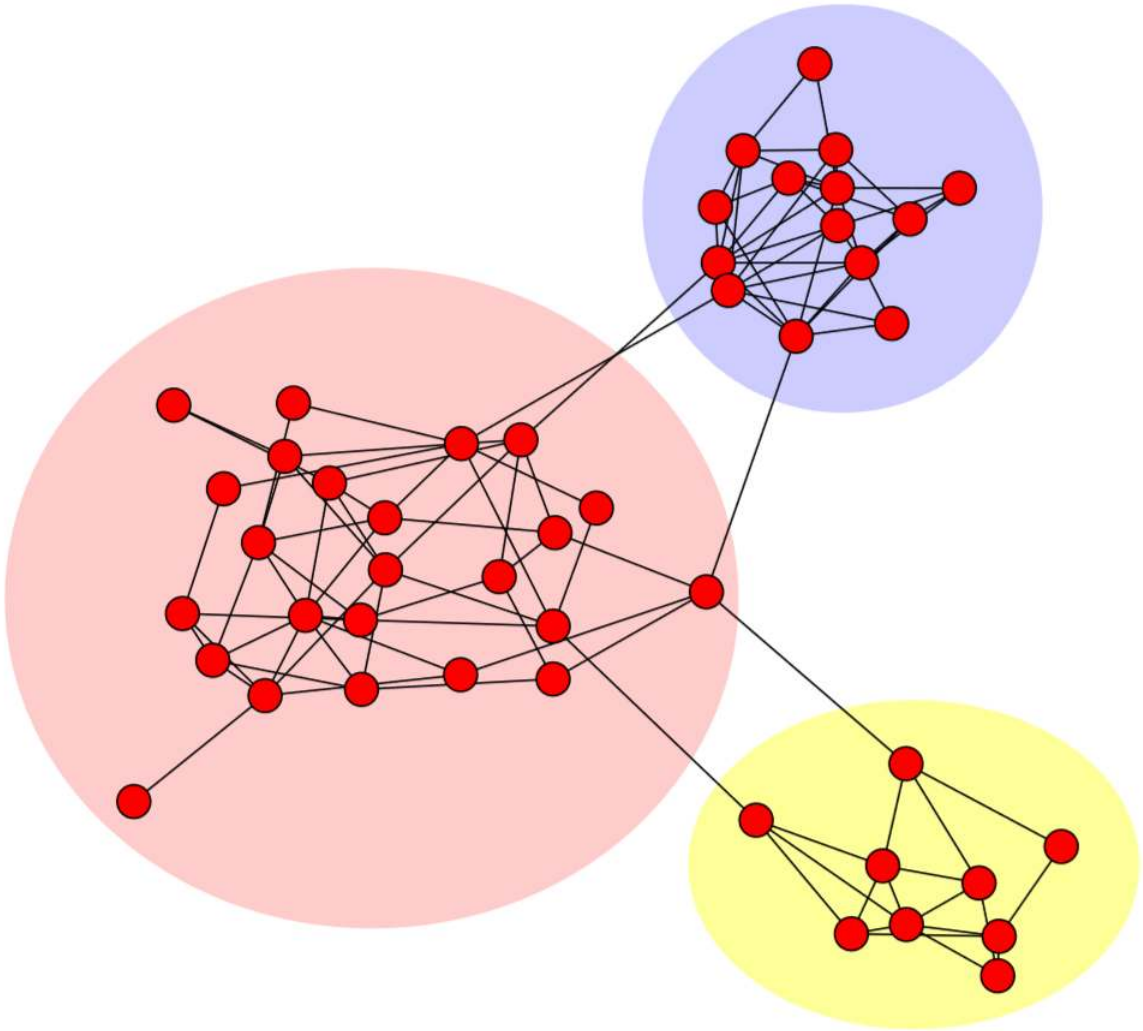
17.7: Community Structure and Modularity

The final topics of this chapter are the *community structure* and *modularity* of a network. These topics have been studied very actively in network science for the last several years. These are typical *mesoscopic properties* of a network; neither microscopic (e.g., degrees or clustering coefficients) nor macroscopic (e.g., density, characteristic path length) properties can tell us how a network is organized at spatial scales intermediate between those two extremes, and therefore, these concepts are highly relevant to the modeling and understanding of complex systems too.

> **Community** A set of nodes that are connected more densely to each other than to the rest of the network. Communities may or may not overlap with each other, depending on their definitions.
>
> **Modularity** The extent to which a network is organized into multiple communities.

Figure 17.7.1 shows an example of communities in a network.

There are literally dozens of different ways to define and detect communities in a network. But here, we will discuss just one method that is now widely used by network science researchers: the Louvain method, proposed by Vincent Blondel et al. in 2008 [77]. It is a very fast, efficient heuristic algorithm that maximizes the modularity of nonoverlapping community structure through an iterative, hierarchical optimization process.



The modularity of a given set of communities in a network is defined as follows [78]:

$$Q = \frac{|E_{in}| - \langle E_{in} \rangle}{|E|} \tag{17.7.1}$$

Here, $|E|$ is the number of edges, $|E_{in}|$ is the number of within-community edges (i.e., those that don't cross boundaries between communities), and $\langle |E_{in}| \rangle$ is the expected number of within-community edges if the topology were purely random. The subtraction of $\langle |E_{in}| \rangle$ on the numerator penalizes trivial community structure, such as considering the entire network a single community that would trivially maximize $|E_{in}|$.

The Louvain method finds the optimal community structure that maximizes the modularity in the following steps:

1. Initially, each node is assigned to its own community where the node itself is the only community member. Therefore the number of initial communities equals the number of nodes.
2. Each node considers each of its neighbors and evaluates whether joining to the neighbor's community would increase the modularity of the community structure. After evaluating all the neighbors, it will join the community of the neighbor that achieves the maximal modularity increase (only if the change is positive; otherwise the node will remain in its own community). This will be repeatedly applied for all nodes until no more positive gain is achievable.
3. The result of Step 2 is converted to a new meta-network at a higher level, by aggregating nodes that belonged to a single community into a meta-node, representing edges that existed within each community as the weight of a self-loop attached to the meta-node, and representing edges that existed between communities as the weights of meta-edges that connect meta-nodes.
4. The above two steps are repeated until no more modularity improvement is possible.

One nice thing about this method is that it is parameter-free; you don't have to specify the number of communities or the criteria to stop the algorithm. All you need is to provide network topology data, and the algorithm heuristically finds the community structure that is close to optimal in achieving the highest modularity.

Unfortunately, NetworkX doesn't have this Louvain method as a built-in function, but its Python implementation has been developed and released freely by Thomas Aynaud, which is available from perso.crans.org/aynaud/communities/. Once you install it, a new `community` module becomes available in Python. Here is an example:

Code 17.22:

```
>>> import networkx as nx
>>> import community as comm
>>> g = nx.karate_club_graph()
>>> bp = comm.best_partition(g)
>>> bp
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 0, 8: 2, 9: 0, 10: 1,
  11: 0, 12: 0, 13: 0, 14: 2, 15: 2, 16: 1, 17: 0, 18: 2, 19: 0,
  20: 2, 21: 0, 22: 2, 23: 3, 24: 3, 25: 3, 26: 2, 27: 3, 28: 3,
  29: 2, 30: 2, 31: 3, 32: 2, 33: 2}
>>> comm.modularity(bp, g)
0.4188034188034188
```

Here, the two important functions in the `community` module are tested. The first one is best_partition, which generates community structure using the Louvain method. The result is given as a dictionary where keys and values are node IDs and community IDs, respectively. The second function shown above is `modularity`, which receives community structure and a network and returns the modularity value achieved by the given communities.

? Exercise 17.7.1

Visualize the community structure in the Karate Club graph using the community IDs as the colors of the nodes.

? Exercise 17.7.2

Import a large network data set of your choice from Mark Newman's Network Data website: http://www-personal.umich.edu/~mejn/netdata/ Detect its community structure using the Louvain method and visualize it if possible.

? Exercise 17.7.3

Do a quick online literature search for other community detection algorithms (e.g., Girvan-Newman method, k-clique percolation method, random walk method, etc.). Choose one of them and read the literature to learn how it works. If the software is available, try it yourself on the Karate Club graph or any other network and see how the result differs from that of the Louvain method.

17.7: Community Structure and Modularity is shared under a CC BY-NC-SA license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

18: Dynamical Networks III - Analysis of Network Dynamics

18.1: Dynamics of Continuous-State Networks

18.2: Diffusion on Networks

18.3: Synchronizability

18.4: Mean-Field Approximation of Discrete-State Networks

18.5: Mean-Field Approximation on Random Networks

18.6: Mean-Field Approximation on Scale-Free Networks

This page titled 18: Dynamical Networks III - Analysis of Network Dynamics is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.1: Dynamics of Continuous-State Networks

We will now switch gears to the analysis of dynamical properties of networks. We will first discuss how some of the analytical techniques we already covered in earlier chapters can be applied to dynamical network models, and then we will move onto some additional topics that are specific to networks.

First of all, I would like to make it clear that we were already discussing dynamical network models in earlier chapters. A typical autonomous discrete-time dynamical system

$$x_t = F(x_{t-1}) \tag{18.1.1}$$

or a continuous-time one

$$\frac{dx}{dt} = F(x), \tag{18.1.2}$$

can be considered a dynamical network if the state space is multidimensional. For example, a system with a five-dimensional state space can be viewed as a dynamical network made of five nodes, each having a scalar state that changes dynamically based on the mathematical rule determined in function F (Fig. 18.1). More specifically, the dynamics of node i's state is determined by the i-th dimensional part of F, and if that part refers to the state vector's j-th component, then node j is connected to node i, and so on.

This means that dynamical networks are not fundamentally different from other dynamical systems. Therefore, if the node states are continuous, then all the analytical

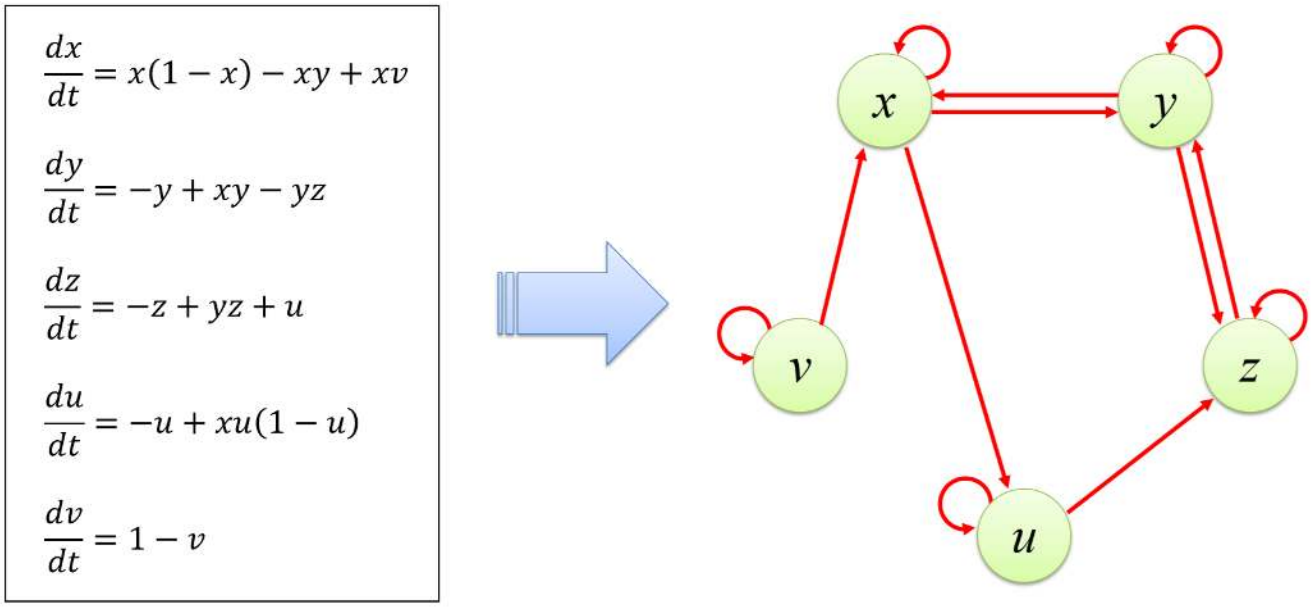


Figure 18.1.1: Schematic illustration of how a multidimensional dynamical system can be viewed as a dynamical network. Left: Actual dynamical equations. Right: Interdependent relationships among variables, represented as a dynamical network.

techniques we discussed before—finding equilibrium points, linearizing dynamics around an equilibrium point, analyzing the stability of the system's state using eigenvalues of a Jacobian matrix, etc.—will apply to dynamical network models without any modification.

Analysis of dynamical networks is easiest when the model is linear, i.e.,

$$x_t = Ax_{t-1} \tag{18.1.3}$$

or

$$\frac{dx}{dt} = Ax \tag{18.1.4}$$

If this is the case, all you need to do is to find eigenvalues of the coefficient matrix A, identify the dominant eigenvalue(s) λ_d (with the largest absolute value for discrete-time cases, or the largest real part for continuous-time cases), and then determine the stability of the system's state around the origin by comparing $|\lambda_d|\_with1 for discrete-time cases, or \(Re(\lambda_d)$ with 0 for continuous-time cases. The dominant eigenvector(s) that correspond to λ_d also tell us the asymptotic state of the network. While this methodology doesn't apply to other more general nonlinear network models, it is still quite useful, because many important network dynamics can be written as linear models. One such example is diffusion, which we will discuss in the following section in more detail.

This page titled 18.1: Dynamics of Continuous-State Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.2: Diffusion on Networks

Many important dynamical network models can be formulated as a linear dynamical system. The first example is the diffusion equation on a network that we discussed in Chapter 16:

$$\frac{dc}{dt} = -\alpha L c \tag{18.2.1}$$

This is a continuous-time version, but you can also write a discrete-time equivalent. As we discussed before, $L = D - A$ is the Laplacian matrix of the network. It is a symmetric matrix in which diagonal components are all non-negative (representing node degrees) while other components are all non-positive. This matrix has some interesting, useful dynamical properties:

> A *Laplacian matrix* of an undirected network has the following properties:
>
> 1. At least one of its eigenvalues is zero.
> 2. All the other eigenvalues are either zero or positive.
> 3. The number of its zero eigenvalues corresponds to the number of connected components in the network.
> 4. If the network is connected, the dominant eigenvector is a homogeneity vector $h = (11\ldots1)^T$[a].
> 5. The smallest non-zero eigenvalue is called the *spectral gap* of the network, which determines how quickly the diffusion takes place on the network.
>
> [a]Even if the network is not connected, you can still take the homogeneity vector as one of the bases of its dominant eigenspace.

The first property is easy to show, because $Lh = (D - A)h = d - d = 0$, where d is the vector made of node degrees. This means that h can be taken as the eigenvector that corresponds to eigenvalue 0. The second property comes from the fact that the Laplacian matrix is positive-semidefinite, because it can be obtained by $L = M^T M$ where M is the signed incidence matrix of the network (not detailed in this textbook).

To understand the rest of the properties, we need to consider how to interpret the eigenvalues of the Laplacian matrix. The actual coefficient matrix of the diffusion equation is $-\alpha L$, and its eigenvalues are $-\alpha\lambda_i$, where λ_i are L's eigenvalues. According to the first two properties discussed above, the coefficient matrix of the equation has at least one eigenvalue 0, and all the eigenvalues are 0 or negative. This means that the zero eigenvalue is actually the dominant one in this case, and its corresponding dominant eigenvector (h, or eigenspace, if the network is not connected) will tell us the asymptotic state of the network.

Let's review the other properties. The third property arises intuitively because, if the network is made of multiple connected components, each of those components behaves as a separate network and shows diffusion independently, converging to a different asymptotic value, and therefore, the asymptotic state of the whole network should have as many degrees of freedom as the connected components. This requires that the dominant eigenspace have as many dimensions, which is why there should be as many degenerate dominant eigenvalue 0's as the connected components in the network. The fourth property can be derived by combining this with the argument on property 1 above.

And finally, the spectral gap. It is so called because the list of eigenvalues of a matrix is called a *matrix spectrum* in mathematics. The spectral gap is the smallest non-zero eigenvalue of L, which corresponds to the largest non-zero eigenvalue of $-\alpha L$ and thus to the mode of the network state that shows the slowest exponential decay over time. Ifthe spectral gap is close to zero, this decay takes a very long time, resulting in slow diffusion. Or if the spectral gap is far above zero, the decay occurs quickly, and so does the diffusion. In this sense, the spectral gap of the Laplacian matrix captures some topological aspects of the network, i.e., how well the nodes are connected to each other from a dynamical viewpoint. The spectral gap of a connected graph (or, the second smallest eigenvalue of a Laplacian matrix in general) is called the *algebraic connectivity* of a network.

Here is how to obtain a Laplacian matrix and a spectral gap in NetworkX:

Code 18.1:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.laplacian_matrix(g)
<34x34 sparse matrix of type '<type 'numpy.int32'>'
       with 190 stored elements in Compressed Sparse Row format>
>>> nx.laplacian_spectrum(g)
array([ 2.84494649e-15, 4.68525227e-01, 9.09247664e-01,
        1.12501072e+00, 1.25940411e+00, 1.59928308e+00,
        1.76189862e+00, 1.82605521e+00, 1.95505045e+00,
        2.00000000e+00, 2.00000000e+00, 2.00000000e+00,
        2.00000000e+00, 2.00000000e+00, 2.48709173e+00,
        2.74915718e+00, 3.01396297e+00, 3.24206748e+00,
        3.37615409e+00, 3.38196601e+00, 3.47218740e+00,
        4.27587682e+00, 4.48000767e+00, 4.58079267e+00,
        5.37859508e+00, 5.61803399e+00, 6.33159222e+00,
        6.51554463e+00, 6.99619703e+00, 9.77724095e+00,
        1.09210675e+01, 1.33061223e+01, 1.70551712e+01,
        1.81366960e+01])
>>> sorted(nx.laplacian_spectrum(g))[1]
0.46852522670139884
```

So, the spectral gap (= algebraic connectivity in this case) of the Karate Club graph is about 0.4685. This value doesn't tell us much about the speed of the diffusion on that network. We will need to compare spectral gaps between different network topologies.

? Exercise 18.2.1

Randomize the topology of the Karate Club graph several times and measure their spectral gaps. Compare the result with the original value obtained above and discuss what it means in terms of the speed of diffusion.

? Exercise 18.2.2

Generate the following network topologies with comparable size and density:

- random graph
- barbell graph
- ring-shaped graph (i.e., degree-2 regular graph)

Measure their spectral gaps and see how topologies quantitatively affect the speed of diffusion.

This page titled 18.2: Diffusion on Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.3: Synchronizability

An interesting application of the spectral gap/algebraic connectivity is to determine the synchronizability of linearly coupled dynamical nodes, which can be formulated as follows:

$$\frac{dx_i}{dt} = R(x_i) + \alpha \sum_{j \epsilon N_i} (H(x_j) - H(x_i)) \tag{18.3.1}$$

Here x_i is the state of node i, R is the local reaction term that produces the inherent dynamica lbehavior of individual nodes, and N_i is the neighborhood of node i. We assume that R is identical for all nodes, and it produces a particular trajectory $x_s(t)$ if there is no interaction with other nodes. Namely, $x_s(t)$ is given as the solution of the differential equation $dx/dt = R(x)$. H is called the output function that homogeneously applies to all nodes. The output function is used to generalize interaction and diffusion among nodes; instead of assuming that the node states themselves are directly visible to others, we assume that a certain aspect of node states (represented by $H(x)$) is visible and diffusing to other nodes.

Equation 18.3.1 can be further simplified by using the Laplacian matrix, as follows:

$$\frac{dx_i}{dt} = R(x_i) - \alpha L \begin{pmatrix} H(x_1) \\ H(x_2) \\ \vdots \\ H(x_n) \end{pmatrix} \tag{18.3.2}$$

Now we want to study whether this network of coupled dynamical nodes can synchronize or not. Synchronization is possible if and only if the trajectory $x_i(t) = x_s(t)$ for all i is stable. This is a new concept, i.e., to study the stability of a dynamic trajectory, not of a static equilibrium state. But we can still adopt the same basic procedure of linear stability analysis: represent the system's state as the sum of the target state and a small perturbation, and then check if the perturbation grows or shrinks over time. Here we represent the state of each node as follows:

$$x_i(t) = x_s(t) + \Delta x_i(t) \tag{18.3.3}$$

By plugging this new expression into Equation 18.3.2, we obtain

$$\frac{d(x_s + \Delta x_i)}{dt} = R(x_s + \Delta x_i) - \alpha L \begin{pmatrix} H(x_s + \Delta x_1) \\ H(x_s + \Delta x_2) \\ \vdots \\ H(x_s + \Delta x_n) \end{pmatrix} \tag{18.3.4}$$

Since Δx_i are very small, we can linearly approximate R and H as follows:

$$\frac{dx_s}{dt} + \frac{d\Delta x_i}{dt} = R(x_s) + R'(x_s)\Delta x_i - \alpha L \begin{pmatrix} H(x_s) + H'(x_s)\Delta x_1 \\ H(x_s) + H'(x_s)\Delta x_2 \\ \vdots \\ H(x_s) + H'(x_s)\Delta x_n \end{pmatrix} \tag{18.3.5}$$

The first terms on both sides cancel out each other because xs is the solution of $dx/dt = R(x)$ by definition. But what about those annoying $H(x_s)$'s included in the vector in the last term? Is there any way to eliminate them? Well, the answer is that we don't have to do anything, because the *Laplacian matrix will eat them all*. Remember that a Laplacian matrix always satisfies $Lh = 0$. In this case, those $H(x_s)$'s constitute a homogeneous vector $H(x_s)h$ altogether. Therefore,$L(H(x_s)h) = H(x_s)Lh$ vanishes immediately, and we obtain

$$\frac{d\Delta x}{dt} = R'(x_s)\Delta x_i - \alpha H'(x_s) L \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_n \end{pmatrix}, \tag{18.3.6}$$

or, by collecting all the Δx_i's into a new perturbation vector Δx,

$$\frac{d\Delta x}{dt} = (R'(x_s)I - \alpha H'(x_s)L)\Delta x, \tag{18.3.7}$$

as the final result of linearization. Note that x_s still changes over time, so in order for this trajectory to be stable, all the eigenvalues of this rather complicated coefficient matrix $(R'(x_s)I - \alpha H'(x_s)L)$ should always indicate stability at any point in time.

We can go even further. It is known that the eigenvalues of a matrix $aX + bI$ are $a\lambda_i + b$, where λ_i are the eigenvalues of X. So, the eigenvalues of $(R'(x_s)I - \alpha H'(x_s)L$ are

$$-\alpha\lambda_i H'(x_s) + R'(x_s), \tag{18.3.8}$$

where λ_i are L'sneigenvalues. The eigenvalue that corresponds to the smallest eigenvalue of L, 0, is just $R'(x_s)$, which is determined solely by the inherent dynamics of $R(x)$ (and thus the nature of $x_s(t)$), so we can't do anything about that. But all the other $n-1$ eigenvalues must be negative all the time, in order for the target trajectory $x_s(t)$ to be stable. So, if we represent the second smallest eigenvalue (the spectral gap for connected networks) and the largest eigenvalue of L by λ_2 and λ_n, respectively, then the stability criteria can be written as

$$\alpha\lambda_i H'(x_s(t)) > R'(x_s(t)) \qquad \text{for all t, and} \tag{18.3.9}$$

$$\alpha\lambda_n H'(x_s(t)) > R'(x_s(t)) \qquad \text{for all t,} \tag{18.3.10}$$

because all other intermediate eigenvalues are "sandwiched" by λ_2 and λ_n. These inequalities provide us with a nice intuitive interpretation of the stability condition: the influence of diffusion of node outputs (left hand side) should be stronger than the internal dynamical drive (right hand side) all the time.

Note that, although α and λ_i are both non-negative, $H'(xs(t))$ could be either positive or negative, so which inequality is more important depends on the nature of the output function H and the trajectory $x_s(t)$ (which is determined by the reaction term R). If $H'(x_s(t))$ always stays non-negative, then the first inequality is sufficient (since the second inequality naturally follows as $\lambda_2 \leq \lambda_n$), and thus the spectral gap is the only relevant information to determine the synchronizability of the network. But if not, we need to consider both the spectral gap and the largest eigenvalue of the Laplacian matrix.

Here is a simple example. Assume that a bunch of nodes are oscillating in an exponentially accelerating pace:

$$\frac{d\theta_i}{dt} = \beta\theta_i + \alpha \sum_{j\epsilon N_i(\theta_j - \theta_i)} \tag{18.3.11}$$

Here, θ_i is the phase of node i, and β is the rate of exponential acceleration that homogeneously applies to all nodes. We also assume that the actual values of θi diffuse to and from neighbor nodes through edges. Therefore, $R(\theta) = \beta\theta$ and $H(\theta) = \theta$ in this model.

We can analyze the synchronizability of this model as follows. Since $H'(\theta) = 1 > 0$, we immediately know that the inequality 18.3.9 is the only requirement in this case. Also, $R'(\theta) = \beta$, so the condition for synchronization is given by

$$\alpha\lambda_2 > \beta, \text{or} \lambda_2 > \frac{\beta}{\alpha}. \tag{18.3.12}$$

Very easy. Let's check this analytical result with numerical simulations on the Karate Club graph. We know that its spectral gap is 0.4685, so if β/α is below (or above) this value, the synchronization should (or should not) occur. Here is the code for such simulations:

Code 18.2: net-sync-analysis.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['theta'] = random()

    nextg = g.copy()

def observe():
    global g, nextg
    subplot(1, 2, 1)
    cla()
    nx.draw(g, cmap = cm.hsv, vmin = -1, vmax = 1,
            node_color = [sin(g.node[i]['theta']) for i in g.nodes_iter()],
            pos = g.pos)
    axis('image')

    subplot(1, 2, 2)
    cla()
```

```
    plot([cos(g.node[i]['theta']) for i in g.nodes_iter()],
         [sin(g.node[i]['theta']) for i in g.nodes_iter()], '.')
    axis('image')
    axis([-1.1, 1.1, -1.1, 1.1])

alpha = 2 # coupling strength
beta = 1 # acceleration rate
Dt = 0.01 # Delta t

def update():
    global g, nextg
    for i in g.nodes_iter():
        theta_i = g.node[i]['theta']
        nextg.node[i]['theta'] = theta_i + (beta * theta_i + alpha * \
            sum(sin(g.node[j]['theta'] - theta_i) for j in g.neighbors(i)) \
            ) * Dt
    g, nextg = nextg, g

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

Here I added a second plot that shows the phase distribution in a $(x, y) = (cos\theta, sin\theta)$ space, just to aid visual understanding.

In the code above, the parameter values are set to `alpha = 2` and `beta = 1`, so $\lambda_2 0.4685 < \beta/\alpha = 0.5$. Therefore, it is predicted that the nodes won't get synchronized. And, indeed, the simulation result confirms this prediction (Fig. 18.3.1(a)), where the nodes initially came close to each other in their phases but, as their oscillation speed became faster and faster, they eventually got dispersed again and the network failed to achieve synchronization. However, if `beta` is slightly lowered to 0.9 so that $\lambda_2 = 0.4685 > \beta/\alpha = 0.45$, the synchronized state becomes stable, which is confirmed again in the numerical simulation (Fig. 18.3.1(b)). It is interesting that such a slight change in the parameter value can cause a major difference in the global dynamics of the network. Also, it is rather surprising that the linear stability analysis can predict this shift so precisely. Mathematical analysis rocks!

? Exercise 18.3.1

Randomize the topology of the Karate Club graph and measure its spectral gap. Analytically determine the synchronizability of the accelerating oscillators model discussed above with $\alpha = 2$, $\beta = 1$ on the randomized network. Then confirm your prediction by numerical simulations. You can also try several other network topologies.

? Exercise 18.3.2

he following is a model of coupled harmonic oscillators where complex node states are used to represent harmonic oscillation in a single-variable differential equation:

$$\frac{dx_i}{dt} = i\omega x_i + \alpha \sum j\epsilon N_i (x_j^\gamma - x_i^y) \tag{18.3.13}$$

Here i is used to denote the imaginary unit to avoid confusion with node index i. Since the states are complex, you will need to use $Re(\cdot)$ on both sides of the inequalities 18.3.9 and 18.3.10 to determine the stability.

Analyze the synchnonizability of this model on the Karate Club graph, and obtain the condition for synchronization regarding the output exponent γ. Then confirm your prediction by numerical simulations.

You may have noticed the synchronizability analysis discussed above is somewhat similar to the stability analysis of the continuous field models discussed in Chapter 14. Indeed, they are essentially the same analytical technique (although we didn't cover stability analysis of dynamic trajectories back then). The only difference is whether the space is represented as a continuous field or as a discrete network. For the former, the diffusion

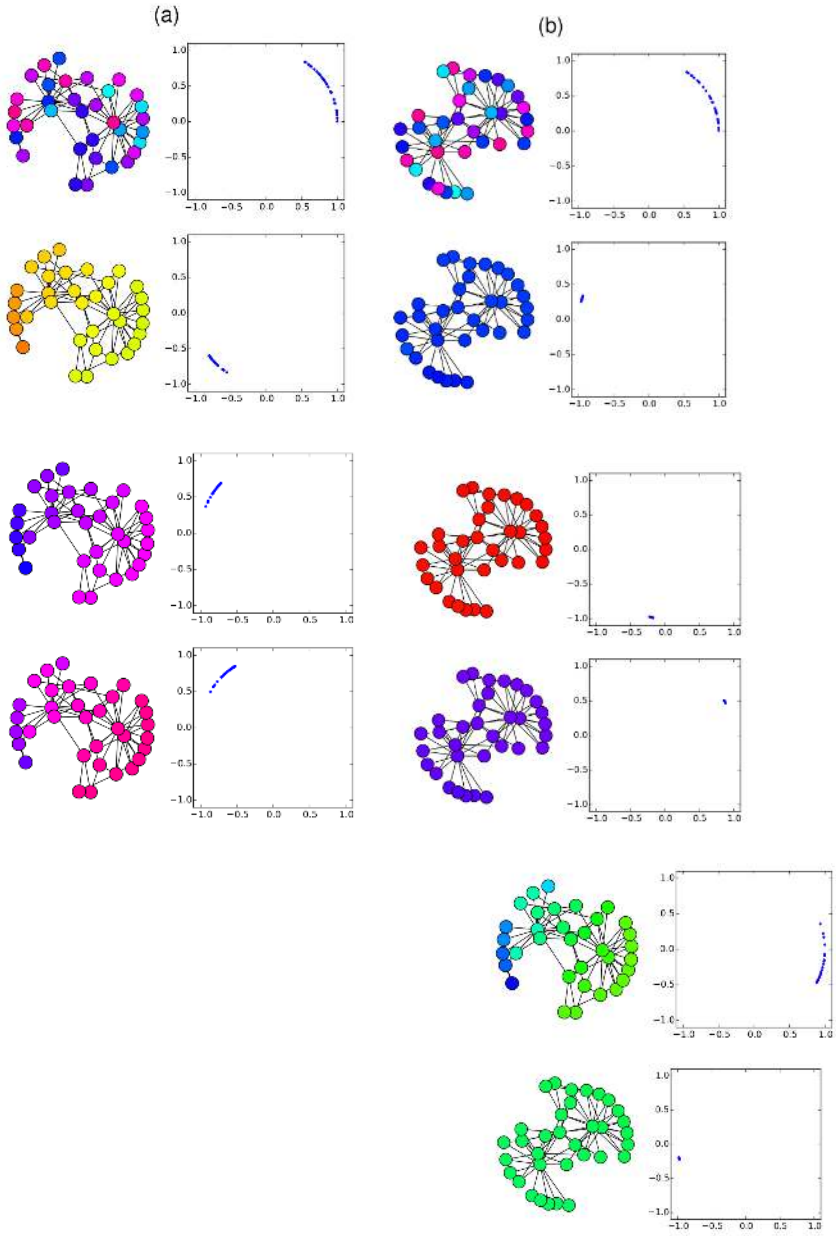


Figure 18.3.1: Visual outputs of Code 18.2. Time flows from top to bottom. (a) Result with α = 2, β = 1 (λ2 < β/α.). (b) Result with $\alpha = 2, \beta = 0.9$ ($\lambda_2 > \beta/\alpha$.).

is represented by the Laplacian operator ∇^2, while for the latter, it is represented by the Laplacian matrix $\boldsymbol{L}$ (note again that their signs are opposite for historical misfortune!). Network models allows us to study more complicated, nontrivial spatial structures,

but there aren't any fundamentally different aspects between these two modeling frameworks. This is why the same analytical approach works for both.

Note that the synchronizability analysis we covered in this section is still quite limited in its applicability to more complex dynamical network models. It relies on the assumption that dynamical nodes are homogeneous and that they are linearly coupled, so the analysis can't generalize to the behaviors of heterogeneous dynamical networks with nonlinear couplings, such as the Kuramoto model discussed in Section 16.2 in which nodes oscillate in different frequencies and their couplings are nonlinear. Analysis of such networks will need more advanced nonlinear analytical techniques, which is beyond the scope of this textbook.

This page titled 18.3: Synchronizability is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.4: Mean-Field Approximation of Discrete-State Networks

Analyzing the dynamics of discrete-state network models requires a different approach, because the assumption of smooth, continuous state space, on which the linear stability analysis is based on, no longer applies. This difference is similar to the difference between continuous field models and cellular automata (CA). In Section 12.3, we analyzed CA models using mean-field approximation. Since CA are just a special case of discretestate dynamical networks, we should be able to apply the same analysis to dynamical networks as well.

In fact, mean-field approximation works on dynamical networks almost in the same way as on CA. But one important issue we need to consider is how to deal with heterogeneous sizes of the neighborhoods. In CA, every cell has the same number of neighbors, so the mean-field approximation is very easy. But this is no longer the case on networks in which nodes can have any number of neighbors. There are a few different ways to deal with this issue.

In what follows, we will work on a simple binary-state example, the *Susceptible-Infected Susceptible (SIS) model*, which we discussed in Section 16.2. As you may remember, the state transition rules of this model are fairly simple: A susceptible node can get infected by an infected neighbor node with infection probability p_i (per infected neighbor), while an infected node can recover to a susceptible node with recovery probability p_r. In the previous chapter, we used asynchronous updating in simulations of the SIS model, but here we assume synchronous, simultaneous updating, in order to make the mean-field approximation more similar to the approximation we applied to CA.

For the mean-field approximation, we need to represent the state of the system by a macroscopic variable, i.e., the probability (= density, fraction) of the infected nodes in the network (say, q) in this case, and then describe the temporal dynamics of this variable by assuming that this probability applies to everywhere in the network homogeneously (i.e., the “mean field”). In the following sections, we will discuss how to apply the mean field approximation to two different network topologies: random networks and scale-free networks.

This page titled 18.4: Mean-Field Approximation of Discrete-State Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.5: Mean-Field Approximation on Random Networks

If we can assume that the network topology is random with connection probability p_e, then infection occurs with a joint probability of three events: that a node is connected to another neighbor node (p_e), that the neighbor node is infected by the disease (q), and that the disease is actually transmitted to the node (p_i). Therefore, $1 - p_e q p_i$ is the probability that the node is not infected by another node. In order for a susceptible node to remain susceptible to the next time step, it must avoid infection in this way $n - 1$ times, i.e., for all other nodes in the network. The probability for this to occur is thus (\(1 − p_{e}qp_{i})^{n−1}. Using this result, all possible scenarios of state transitions can be summarized as shown in Table 18.5.1.

***Table* 18.5.1**: Possible scenarios of state transitions in the network SIS model.

Current state	Next state	Probabiltiy of this transition
0 (susceptible)	0 (suceptible)	$(1-q)(1-p_e q p_i)^{n-1}$
0 (susceptible)	1 (infected)	$(1-q)(1-(1-p_e q p_i)^{n-1})$
1 (infected)	0 (susceptible)	qp_r
1 (infected)	1 (infected)	$q(1-p_r)$

We can combine the probabilities of the transitions that turn the next state into 1, to write the following difference equation for q_t (whose subscript is omitted on the right hand side for simplicity):

$$q_{t+1} = (1-q)(1-(1-p_e q p_i)^{n-1})q(1-p_r) \tag{18.5.1}$$
$$= 1 - q - (1-q)(1-p_e q p_i)^{n-1} + q - qp_r \tag{18.5.2}$$
$$\approx 1 - (1-q)(1-(n-1)p_e q p_i) - qp_r \quad (because\ p_e q p_i\ is\ small) \tag{18.5.3}$$
$$= q((1-q)(n-1)p_e p_i + 1 - p_r) \tag{18.5.4}$$
$$= q((1-q)s + 1 - p_r) = f(q) \quad (with\ s = (n-1)p_e p_i) \tag{18.5.5}$$

Now this is a simple iterative map about qt, which we already know how to study. By solving $f(q_{eq}) = q_{eq}$, we can easily find that there are the following two equilibrium points:

$$q_{eq} = 0, 1 - \frac{p_r}{s} \tag{18.5.6}$$

And the stability of each of these points can be studied by calculating the derivative of $f(q)$:

$$\frac{df(q)}{dq} = 1 - p_r + (1-2q)s \tag{18.5.7}$$

$$\frac{df(q)}{dq}|_{q=0} = 1 - p_r + s \tag{18.5.8}$$

$$\frac{df(q)}{dq}|_{q=1-p_r/s} = 1 + p_r - s \tag{18.5.9}$$

So, it looks like $p_r - s$ is playing an important role here. Note that $0 \le p_r \le 1$ because it is a probability, and also $0 \le s \le 1$ because $(1 - s)$ is an approximation of $(1 - p_e q p_i)n - 1$, which is also a probability. Therefore, the valid range of $p_r - s$ is between -1 and 1. By comparing the absolute values of Eqs. 18.5.8 and 18.5.9 to 1 within this range, we find the stabilities of those two equilibrium points as summarized in Table 18.5.2.

Table 18.5.2: Stabilities of the two equilibrium points in the network SIS model.

Equilibrium point	$-1 \leq p_r - s \leq 0$	$0 < p_r - s \leq 1$
$q = 0$	unstable	**stable**
$q = 1 - \frac{P_r}{s}$	**stable**	unstable

Now we know that there is a critical *epidemic threshold* between the two regimes. If $p_r > s = (n-1)p_e p_i$, the equilibrium point $q_{eq} = 0$ becomes stable, so the disease should go away quickly. But otherwise, the other equilibrium point becomes stable instead, which means that the disease will never go away from the network. This epidemic threshold is often written in terms of the infection probability, as

$$p_i > \frac{p_r}{(n-1)p_e} = \frac{p_r}{\langle k \rangle} \tag{18.5.10}$$

as a condition for the disease to persist, where $\langle k \rangle$ is the average degree. It is an important characteristic of epidemic models on random networks that there is a positive lower bound for the disease's infection probability. In other words, a disease needs to be "contagious enough" in order to survive in a randomly connected social network.

? Exercise 18.5.1

Modify Code 16.6 to implement a synchronous, simultaneous updating version of the network SIS model. Then simulate its dynamics on an Erd˝osR´enyi random network for the following parameter settings:

- n = 100, $p_e = 0.1$, $p_i = 0.5$, $p_r = 0.5 (pr < (n-1)p_e p_i)$
- n = 100, $p_e = 0.1$, $p_i = 0.04$, $p_r = 0.5 (p_r > (n-1)p_e p_i)$
- n = 200, $p_e = 0.1$, $p_i = 0.04$, $p_r = 0.5 (p_r < (n-1)p_e p_i)$
- n = 200, $p_e = 0.05$, $p_i = 0.04$, $p_r = 0.5 (p_r > (n-1)p_e p_i)$

Discuss how the results compare to the predictions made by the mean-field approximation.

As you can see in the exercise above, the mean-field approximation works much better on random networks than on CA. This is because the topologies of random networks are not locally clustered. Edges connect nodes that are randomly chosen from the entire network, so each edge serves as a global bridge to mix the states of the system, effectively bringing the system closer to the "mean-field" state. This, of course, will break down if the network topology is not random but locally clustered, such as that of the Watts-Strogatz small-world networks. You should keep this limitation in mind when you apply mean-field approximation.

? Exercise 18.5.2

If you run the simulation using the original Code 16.6 with asynchronous updating, the result may be different from the one obtained with synchronous updating. Conduct simulations using the original code for the same parameter settings as those used in the previous exercise. Compare the results obtained using the two versions of the model, and discuss why they are so different.

This page titled 18.5: Mean-Field Approximation on Random Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

18.6: Mean-Field Approximation on Scale-Free Networks

What if the network topology is highly heterogeneous, like in scale-free networks, so that the random network assumption is no longer applicable? A natural way to reconcile such heterogeneous topology and mean-field approximation is to adopt a specific degree distribution $P(k)$. It is still a non-spatial summary of connectivities within the network, but you can capture some heterogeneous aspects of the topology in $P(k)$.

One additional complication the degree distribution brings in is that, because the nodes are now different from each other regarding their degrees, they can also be different from each other regarding their state distributions too. In other words, it is no longer reasonable to assume that we can represent the global state of the network by a single "mean field" q. Instead, we will need to represent q as a function of degree k (i.e., a bunch of mean fields, each for a specific k), because heavily connected nodes may become infected more often than poorly connected nodes do. So, here is the summary of the new quantities that we need to include in the approximation:

- $P(k)$: Probability of nodes with degree k
- $q(k)$: Probability for a node with degree k to be infected

Let's consider how to revise Table 18.5.1 using $P(k)$ and $q(k)$. It is obvious that all the q's should be replaced by $q(k)$. It is also apparent that the third and fourth row (probabilities of transitions for currently infected states) won't change, because they are simply based on the recovery probability p_r. And the second row can be easily obtained once the first row is obtained. So, we can just focus on calculating the probability in the first row: What is the probability for a susceptible node with degree k to remain susceptible in the next time step?

We can use the same strategy as what we did for the random network case. That is, we calculate the probability for the node to get infected from another node, and then calculate one minus that probability raised to the power of the number of neighbors, to obtain the probability for the node to avoid any infection. For random networks, all other nodes were potential neighbors, so we had to raise $(1-p_e q p_i)$ to the power of $n-1$. But this is no longer necessary, because we are now calculating the probability for a node with the specific degree k. So, the probability that goes to the first row of the table should look like:

$$1-(q(k))(1-something)^k \tag{18.6.1}$$

Here, "something" is the joint probability of two events, that the neighbor node is infected by the disease and that the disease is actually transmitted to the node in question. The latter is still p_i, but the former is no longer represented by a single mean field. We need to consider all possible degrees that the neighbor might have, and then aggregate them all to obtain an overall average probability for the neighbor to be in the infected state. So, here is a more elaborated probability:

$$(1-q(k))(1-\sum)k' P_n(k'|k) q(k') p_i)^k \tag{18.6.2}$$

Here, k' is the neighbor's degree, and the summation is to be conducted for all possible values of k'. $P_n(k'|k)$ is the conditional probability for a neighbor of a node with degree k to have degree k'. This could be any probability distribution, which could represent assortative or disassortative network topology. But if we assume that the network is neither assortative nor disassortative, then $P_n(k'|k)$) doesn't depend on k at all, so it becomes just $P_n(k')$: the *neighbor degree distribution*.

Wait a minute. Do we really need such a special distribution for neighbors' degrees? The neighbors are just ordinary nodes, after all, so can't we use the original degree distribution $P(k')$ instead of $P_n(k')$? As strange as it may sound, the answer is an astonishing NO. This is one of the most puzzling phenomena on networks, but your neighbors are *not quite ordinary people*. The average degree of neighbors is actually higher than the average degree of all nodes, which is often phrased as *"your friends have more friends than you do."* As briefly discussed in Section 16.2, this is called the *friendship paradox*, first reported by sociologist Scott Feld in the early 1990s [68].

We can analytically obtain the neighbor degree distribution for non-assortative networks. Imagine that you randomly pick one edge from a network, trace it to one of its end nodes, and measure its degree. If you repeat this many times, then the distribution you get is the neighbor degree distribution. This operation is essentially to randomly choose one "hand," i.e, half of an edge, from the entire network. The total number of hands attached to the nodes with degree k' is given by $k'nP(k')$, and if you sum this over all k', you

will obtain the total number of hands in the network. Therefore, if the sampling is purely random, the probability for a neighbor (i.e., a node attached to a randomly chosen hand) to have degree k' is given by

$$P_n(k') = \frac{k' n P(k')}{\sum_k k' m P(k')} = \frac{k' P(k')}{\sum_{k'} k' P(k')} = \frac{k'}{\langle k \rangle} P(k'), \tag{18.6.3}$$

where $\langle k \rangle$ is the average degree. As you can clearly see in this result, the neighbor degree distribution is a modified degree distribution so that it is proportional to degree k'. This shows that higher-degree nodes have a greater chance to appear as neighbors. If we calculate the average neighbor degree $\langle k_n \rangle$, we obtain

$$\langle k_n \rangle = \sum_{k'} k' P_n(k') = \sum_{k'} \frac{k'^2 P(k')}{\langle k \rangle} = \frac{\langle k^2 \rangle}{\langle k \rangle} = \frac{\langle k \rangle^2 + \sigma(k)^2}{\langle k \rangle} = \langle k \rangle + \frac{\sigma(k)^2}{\langle k \rangle}, \tag{18.6.4}$$

where $\sigma(k)^2$ is the variance of the degree distribution. This mathematically shows that, if there is any variance in the degree distribution (which is true for virtually all real-world networks), your friends have more friends than you do, on average. This may be a bit depressing, but true. Just face it.

Okay, enough on the friendship paradox. Let's plug Equation 18.6.3 back into Equation 18.6.2, which gives

$$(1 - q(k))(1 - \sum_{k'} \frac{k'}{\langle k \rangle} P(k') q(k') p_i)^k. \tag{18.6.5}$$

With this, we can finally complete the state transition probability table as shown in Table 18.6.1.

Table 18.6.1: Possible scenarios of state transitions in the network SIS model, with degree distribution $P(k)$ and degree-dependent infection probability $q(k)$.

Current state	Next state	Probability of this transition
0 (susceptible)	0 (susceptible)	$(1 - q(k))$ $(1 - \sum_{k'} \frac{k'}{\langle k \rangle} P(k') q(k') p_i)^k$
0 (susceptible)	1 (infected)	$(1 - q(k))$ $(1 - (1 - \sum_{k'} \frac{k'}{\langle k \rangle} P(k') q(k') p_i)^k)$
1 (infected)	0 (susceptible)	$q(k) p_r$
1 (infected)	1 (infected)	$q(k)()1 - p_r$

We can add probabilities of state transitions that turn the next state into 1, to obtain a difference equation for $q_t(k)$ (again, the subscripts are omitted on the right hand side), i.e.,

$$q_{t+1}(k) = (1 - q(k))(1 - q(k))(1 - (1 - \sum_{k'} \frac{k'}{\langle k \rangle} P(k') q(k') p_i)^k) + q(k)(1 - p_r) \tag{18.6.6}$$

$$= (1 - q(k))(1 - (1 - q_n p_i)^k) + q(k)(1 - p_r) \tag{18.6.7}$$

with

$$q_n = \frac{\sum_{k'} k' P(k') q(k')}{\langle k \rangle}. \tag{18.6.8}$$

q_n is the probability that a neighbor node is infected by the disease. Thus $q_n p_i$ corresponds to the "something" part in Equation 18.6.1. It can take any value between 0 and 1, but for our purpose of studying the epidemic threshold of infection probability, we

can assume that $q_n p_i$ is small. Therefore, using the approximation $(1-x)^k \approx 1-kx$ for small x again, the iterative map above becomes

$$q_{t+1} = (1-q(k))(1-(1-kq_n p_i)) + q(k)(1-p_r) \tag{18.6.9}$$
$$= (1-q(k))kq_n p_i + p(k) - q(k)p_r = f(q(k)). \tag{18.6.10}$$

Then we can calculate the equilibrium state of the nodes with degree k as follows:

$$q_{eq}(k) = (q - q_{eq}(k))kq_n p_i + q_{eq}(k) - q_{eq}(k)p_r \tag{18.6.11}$$

$$q_{eq}p_r = kq_n p_i - kq_n p_i q_{eq}(k) \tag{18.6.12}$$

$$q_{eq}(k) = \frac{kq_n p_i}{kq_n p_i + p_r} \tag{18.6.13}$$

We can apply this back to the definition of q_n to find the actual equilibrium state:

$$q_n = \frac{1}{\langle k \rangle} \sum_{k'} k' P(k') \frac{kq_n p_i}{kq_n p_i + p_r} \tag{18.6.14}$$

Clearly, $q_n = 0$ (i.e., $q_{eq}(k) = 0$ for all k) satisfies this equation, so the extinction of the disease is still an equilibrium state of this system. But what we are interested in is the equilibrium with $q_n \neq 0$(i.e., $q_{eq}(k) > 0$ for some k), where the disease continues to exist, and we want to know whether such a state is stable. To solve Equation 18.6.14, we will need to assume a certain degree distribution $P(k)$.

For example, if we assume that the network is large and random, we can use the following crude approximate degree distribution:

$$P(k) \approx \begin{cases} 1 \text{ if } k = \langle k \rangle \\ 0 \text{otherwise} \end{cases} \tag{18.6.15}$$

We can use this approximation because, as the network size increases, the degree distribution of a random graph becomes more and more concentrated at the average degree (relative to the network size), due to the law of large numbers well known in statistics. Then Equation 18.6.14 is solved as follows:

$$q_n = \frac{\langle k \rangle q_n p_i}{\langle k \rangle q_n p_i + p_r} \tag{18.6.16}$$

$$\langle k \rangle q_n p_i + p_r = \langle k \rangle p_i \tag{18.6.17}$$

$$q_n = 1 - \frac{p_r}{\langle k \rangle p_i} \tag{18.6.18}$$

We can check the stability of this equilibrium state by differentiating Equation 18.6.10 by $q(k)$ and then applying the results above. In so doing, we should keep in mind that q_n contains $q(k)$ within it (see Equation 18.6.8). So the result should look like this:

$$\frac{df(q(k))}{dq(k)} = -kq_n p_i + (1-q(k))kp_i \frac{dq_n}{dq(k)} + 1 - p_r \tag{18.6.19}$$
$$= -kq_n p_i + (1-q(k)) \frac{k^2 P(k) p_i}{\langle k \rangle} + 1 - p_r \tag{18.6.20}$$

At the equilibrium state Equation 18.6.13, this becomes

$$\frac{df(q(k))}{dq(k)}\Big|_{q(k)=\frac{kq_n p_i}{kq_n p_i + p_r}} = -kq_n p_i + \frac{p_r}{kq_n p_i + p_r} \frac{k^2 P(k) p_i}{\langle k \rangle} + 1 - p_r = r(k). \tag{18.6.21}$$

Then, by applying $P(k)$ and q_n with a constraint $k \to \langle k \rangle$ for large random networks, we obtain

$$r(\langle k \rangle) = -\langle k \rangle (1 - \frac{p_r}{\langle k \rangle p_i}) p_i + \frac{p_r}{\langle k \rangle (1 - \frac{p_r}{\langle k \rangle p_i}) p_i + p_r} \frac{\langle k \rangle^2 p_i}{\langle k \rangle} + -p_r \tag{18.6.22}$$
$$= -\langle k \rangle p_i + p_r + p_r + 1 - p_r \tag{18.6.23}$$
$$= -\langle k \rangle p_i + p_r + 1. \tag{18.6.24}$$

In order for the non-zero equilibrium state to be stable:

$$-1 < -\langle k \rangle p_i + p_r + 1 < 1 \tag{18.6.25}$$

$$\frac{p_r}{\langle k \rangle} < p_i < \frac{p_r + 2}{\langle k \rangle} \tag{18.6.26}$$

Note that the lower bound indicated above is the same as the epidemic threshold we obtained on random networks before (Eq. (18.5.18)). So, it seems our analysis is consistent so far. And for your information, the upper bound indicated above is another critical threshold at which this non-zero equilibrium state loses stability and the system begins to oscillate.

What happens if the network is scale-free? Here, let's assume that the network is a Barab´asi-Albert scale-free network, whose degree distribution is known to be $P(k) = 2m^2k^{-3}$ where m is the number of edges by which each newcomer node is attached to the network, and thus $\langle k \rangle = 2m$ and $k_{min} = m$ [57]. Then, from Equation 18.6.14[1], we obtain

$$q_n = \frac{1}{2m} \sum_{k'=m}^{\infty} k' \cdot 2m^2 k'^{-3} \frac{k' q_n p_i}{k' q_n p_i + p_r} \tag{18.6.27}$$

$$1 = m p_i \sum_{k'=m}^{\infty} \frac{1}{k'(k' q_n p_i + p_r)}. \tag{18.6.28}$$

The summation can be approximated by a continuous integral, which results in

$$1 \approx m p_i \int_m^{\infty} \frac{dk'}{k'(k' q_n p_i + p_r)} \tag{18.6.29}$$

$$= \frac{m p_i}{p_r} \int_m^{\infty} \left(\frac{1}{k'} - \frac{1}{k' + p_r/(q_n p_i)}\right) dk' \tag{18.6.30}$$

$$= \frac{m p_i}{p_r} \left[\log\left(\frac{k'}{k' + p_{r/(q_n p_i)}}\right)\right]_m^{\infty} \tag{18.6.31}$$

$$= \frac{m p_i}{p_r} \log\left(1 + \frac{p_r}{m q_n p_i}\right), \tag{18.6.32}$$

$$q_n \approx \frac{p_r}{\left(e^{\frac{p_r}{m p_i}} - 1\right) m p_i}. \tag{18.6.33}$$

We can apply this result (together with $P(k) = 2m^2k^{-3}$) to $r(k)$ in Equation 18.6.21 to check the stability of this non-zero equilibrium state:

$$r(k) = -k \frac{p_r}{\left(e^{\frac{p_r}{m p_i}} - 1 m p_i\right)} p_t + \frac{p_r}{k \frac{p_r}{\left(e^{\frac{p_r}{m p_i}} - 1\right) m p_i} p_i + p_r} \tag{18.6.34}$$

$$= -\frac{k p_r}{\left(e^{\frac{p_r}{m p_i}} - 1\right) m} + \frac{m p_i}{\left(e^{\frac{p_r}{m_i}} - 1\right) m} + 1 - p_r \tag{18.6.35}$$

This is rather complex, and it would be difficult to solve it in terms of p_i. But still, we can show something very interesting using this formula. If we lower the infection probability p_i down to zero, this occurs asymptotically:

$$\lim_{p_i \to 0} r(k) = -\frac{k p_r}{\left(\left[e^{\frac{p_r}{m p_i}} \to \infty\right] m\right.} + \frac{m \left[p_i \to 0\right]}{\left(\left[e^{\frac{p_r}{m p_i} \to \infty}\right] - k\right.} + 1 - p_r \tag{18.6.36}$$

$$1 - p_r \tag{18.6.37}$$

This shows $0 < lim_{pi \to 0} r(k) < 1$, i.e., the non-zero equilibrium state remains stable even if p_i approaches to zero. In other words, *there is no epidemic threshold if the network is scale-free!* This profound result was discovered by statistical physicists Romualdo Pastor Satorras and Alessandro Vespignani in the early 2000s[79], which illustrates an important fact that, on networks whose topologies are scale-free, diseases can linger around indefinitely, no matter how weak their infectivity is. This is a great example of how complex network topologies can fundamentally change the dynamics of a system, and it also illustrates how misleading the predictions made using random models can be sometimes. This finding and other related theories have lots of real-world applications, such as understanding, modeling, and prevention of epidemics of contagious diseases in society as well as those of computer viruses on the Internet.

? Exercise 18.6.1

Simulate the dynamics of the network SIS model on Barab´asi Albert scale-free networks to check if there is indeed no epidemic threshold as the theory predicts. In simulations on any finite-sized networks, there is always the possibility of accidental extinction of diseases, so you will need to make the network size as large as possible to minimize this "finite size" effect. You should compare the results with those obtained from control experiments that use random networks.

Finally, I should mention that all the analytical methods discussed above are still quite limited, because we didn't consider any degree assortativity or disassortativity, any possible state correlations across edges, or any coevolutionary dynamics that couple state changes and topological changes. Real-world networks often involve such higher-order complexities. In order to better capture them, there are more advanced analytical techniques available, such as *pair approximation and moment closure*. If you want to learn more about these techniques, there are some more detailed technical references available [80, 81, 82, 83].

? Exercise 18.6.2

Consider a network with extremely strong assortativity, so that

$$P(k'|k) \approx \begin{cases} 1 & \text{if } k' = k \\ 0 & \text{otherwise} \end{cases} \tag{18.6.38}$$

Use this new definition of $P(k'|k)$ in Equation 18.6.2 to conduct a mean-field approximation, and determine whether this strongly assortative network has an epidemic threshold or not.

[1]Here we assume that Barab´asi-Albert networks are non-assortative.

This page titled 18.6: Mean-Field Approximation on Scale-Free Networks is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

CHAPTER OVERVIEW

19: Agent-Based Models

Agent-Based Models (ABMs) are arguably the most generalized framework for modeling and simulation of complex systems, which actually include both cellular automata and dynamical networks as special cases. ABMs are widely used in a variety of disciplines to simulate dynamical behaviors of systems made of a large number of entities, such as traders' behaviors in a market (in economics), migration of people (in social sciences), interaction among employees and their performance improvement (in organizational science), flocking/schooling behavior of birds/fish (in behavioral ecology), cell growth and morphogenesis (in developmental biology), and collective behavior of granular materials (in physics).

19.1: What Are Agent-Based Models?

19.2: Building an Agent-Based Model

19.3: Agent-Environment Interaction

19.4: Ecological and Evolutionary Models

This page titled 19: Agent-Based Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

19.1: What Are Agent-Based Models?

At last, we have reached the very final chapter, on *agent-based models (ABMs)*. ABMs are arguably the most generalized framework for modeling and simulation of complex systems, which actually include both cellular automata and dynamical networks as special cases. ABMs are widely used in a variety of disciplines to simulate dynamical behaviors of systems made of a large number of entities, such as traders' behaviors in a market (in economics), migration of people (in social sciences), interaction among employees and their performance improvement (in organizational science), flocking/schooling behavior of birds/fish (in behavioral ecology), cell growth and morphogenesis (in developmental biology), and collective behavior of granular materials (in physics). Figure 19.1 shows a schematic illustration of an ABM.

It is a little challenging to define precisely what an agent-based model is, because its modeling assumptions are wide open, and thus there aren't many fundamental constraints that characterize ABMs. But here is what I hope to be a minimalistic definition of them:

> *Agent-based models* are computational simulation models that involve many discrete agents.

There are a few keywords in this definition that are important to understand ABMs.

The first keyword is "computational." ABMs are usually implemented as simulation models in a computer, where each agent's behavioral rules are described in an algorithmic fashion rather than a purely mathematical way. This allows modelers to implement complex internal properties of agents and their nontrivial behavioral rules. Such representations of complex individual traits are highly valued especially in social, organizational

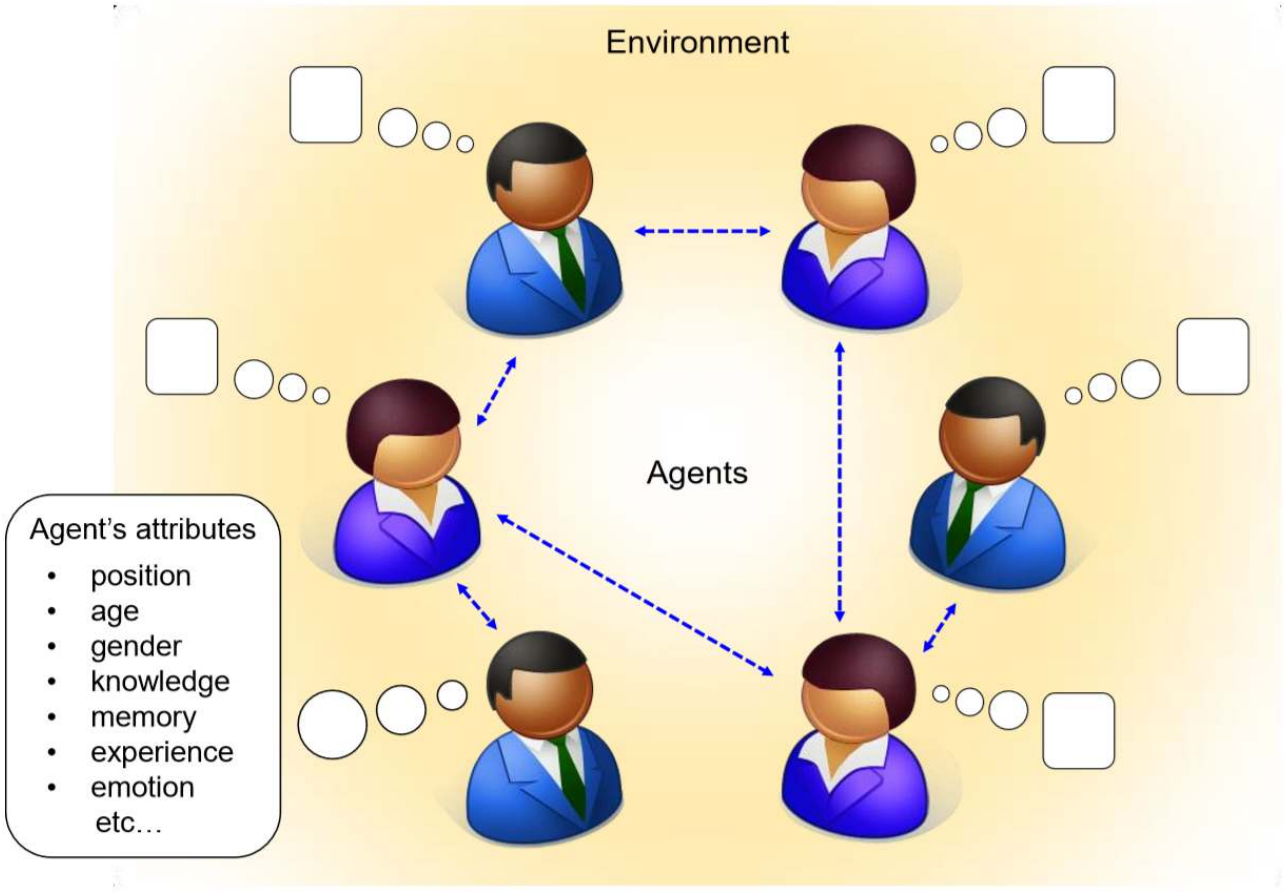


Figure **19.1.1**: Schematic illustration of what an agent-based model (ABM) looks like.

and management sciences, where the modelers need to capture complex, realistic behaviors of human individuals. This is why ABMs are particularly popular in those research areas.

This, of course, comes at the cost of analytical tractability. Since agents can have any number of complex properties and behavioral rules, it is generally not easy to conduct an elegant mathematical analysis of an ABM (which is why there is no "Analysis" chapter on ABMs after this one). Therefore, the analysis of ABMs and their simulation results are usually carried out using more conventional statistical analysis commonly used in social sciences, e.g., by running Monte Carlo simulations to obtain distributions of outcome measurements under multiple experimental conditions, and then conducting statistical hypothesis testing to see if there was any significant difference between the different experimental conditions. In this sense, ABMs could serve as a virtual replacement of experimental fields for researchers.

The second keyword in the definition above is "many." Although it is technically possible to create an ABM made of just a few agents, there would be little need for such a model, because the typical context in which an ABM is needed is when researchers want to study the collective behavior of a large number of agents (otherwise it would be sufficient to use a more conventional equation-based model with a small number of variables). Therefore, typical ABMs contain a population of agents, just like cells in CA or nodes in dynamical networks, and their dynamical behaviors are studied using computational simulations.

The third keyword is "discrete." While there are some ambiguities about how to rigorously define an agent, what is commonly accepted is that an agent should be a discrete individual entity, which has a clear boundary between self and the outside. CA and network , so they qualify as special cases of ABMs. In the meantime, continuous field models adopt continuous spatial functions as a representation of the system's state, so they are not considered ABMs.

There are certain properties that are generally assumed in agents and ABMs, which collectively define the "agent-ness" of the entities in a model. Here is a list of such properties:

Typical properties generally assumed in agents and ABMs

- Agents are discrete entities.
- Agents may have internal states.
- Agents may be spatially localized.
- Agents may perceive and interact with the environment.
- Agents may behave based on predefined rules.
- Agents may be able to learn and adapt.
- Agents may interact with other agents.
- ABMs often lack central supervisors/controllers.
- ABMs may produce nontrivial "collective behavior" as a whole.

Note that these are not strict requirements for ABMs (perhaps except for the first one). Some ABMs don't have internal states of agents; some don't have space or environment; and some *do* have central controllers as special kinds of agents. Therefore, which model properties should be incorporated into an ABM is really up to the objective of your model.

Before we move on to actual ABM building, I would like to point out that there are a few things we need to be particularly careful about when we build ABMs. One is about coding. Implementing an ABM is usually much more coding-intense than implementing other simpler models, partly because the ABM framework is so open-ended. The fact that there aren't many constraints on ABMs also means that you have to take care of all the details of the simulation yourself. This naturally increases the amount of coding you will need to do. And, the more you code, the more likely an unexpected bug or two will sneak into your code. It is thus very important to keep your code simple and organized, and to use the best practices in computer programming (e.g., modularizing sections, adding plenty of comments, implementing systematic tests, etc.), in order to minimize the risks of having bugs in your simulation. If possible, it is desirable to have multiple people test and thoroughly check your code.

Another issue we should be aware of is that, since ABMs are so open-ended and flexible, modelers are tempted to add more and more complex settings and assumptions into their ABMs. This is understandable, as ABMs are such nice playgrounds to try testing "what-if" scenarios in a virtual world, and the results can be obtained quickly. I have seen many people who became fascinated by such an interactive modeling experience and tried adding more and more details into their own ABMs to make them more "realistic." But beware—the increased model complexity means the increased difficulty of analyzing and justifying the model and its results. If we want to derive a useful, reliable conclusion from our model, we should resist the temptation to unnecessarily add

complexity to our ABMs. We need to strike the right balance between simplicity, validity, and robustness, as discussed in Section 2.4.

? Exercise 19.1.1

Do a quick online literature search to learn how ABMs are used in various scientific disciplines. Choose a few examples of your interest and learn more about how researchers developed and used their ABMs for their research.

This page titled 19.1: What Are Agent-Based Models? is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

19.2: Building an Agent-Based Model

Let's get started with agent-based modeling. In fact, there are many great tutorials already out there about how to build an ABM, especially those by Charles Macal and Michael North, renowned agent-based modelers at Argonne National Laboratory [84]. Macal and North suggest considering the following aspects when you design an agent-based model:

1. Specific problem to be solved by the ABM
2. Design of agents and their static/dynamic attributes
3. Design of an environment and the way agents interact with it
4. Design of agents' behaviors
5. Design of agents' mutual interactions
6. Availability of data
7. Method of model validation

Among those points, 1, 6, and 7 are about fundamental scientific methodologies. It is important to keep in mind that just building an arbitrary ABM and obtaining results by simulation wouldn't produce any scientifically meaningful conclusion. In order for an ABM to be scientifically meaningful, it has to be built and used in either of the following two complementary approaches:

A. Build an ABM using model assumptions that are derived from empirically observed phenomena, and then produce previously unknown collective behaviors by simulation.
B. Build an ABM using hypothetical model assumptions, and then reproduce empirically observed collective phenomena by simulation.

The former is to use ABMs to make predictions using validated theories of agent behaviors, while the latter is to explore and develop new explanations of empirically observed phenomena. These two approaches are different in terms of the scales of the known and the unknown (A uses micro-known to produce macro-unknown, while B uses micro-unknown to reproduce macro-known), but the important thing is that one of those scales should be grounded on well-established empirical knowledge. Otherwise, the simulation results would have no implications for the real-world system being modeled. Of course, a free exploration of various collective dynamics by testing hypothetical agent behaviors to generate hypothetical outcomes is quite fun and educational, with lots of intellectual benefits of its own. My point is that we shouldn't misinterpret outcomes obtained from such exploratory ABMs as a validated prediction of reality.

In the meantime, items 2, 3, 4, and 5 in Macal and North's list above are more focused on the technical aspects of modeling. They can be translated into the following design tasks in actual coding using a programming language like Python:

Design tasks you need to do when you implement an ABM

1. Design the data structure to store the attributes of the agents.
2. Design the data structure to store the states of the environment.
3. Describe the rules for how the environment behaves on its own.
4. Describe the rules for how agents interact with the environment.
5. Describe the rules for how agents behave on their own.
6. Describe the rules for how agents interact with each other.

Representation of agents in Python

It is often convenient and customary to define both agents' attributes and behaviors using a *class* in object-oriented programming languages, but in this textbook, we won't cover object-oriented programming in much detail. Instead, we will use Python's dynamic class as a pure data structure to store agents' attributes in a concise manner. For example, we can define an empty `agent` class as follows:

Code 19.1:

```
class agent:
    pass
```

The `class` command defines a new class under which you can define various attributes (variables, properties) and methods (functions, actions). In conventional object-oriented programming, you need to give more specific definitions of attributes and methods available under this class. But in this textbook, we will be a bit lazy and exploit the dynamic, flexible nature of Python's classes. Therefore, we just threw pass into the class definition above. pass is a dummy keyword that doesn't do anything, but we still need something there just for syntactic reasons.

Anyway, once this `agent` class is defined, you can create a new empty agent object as follows:

Code 19.2:

```
>>> a = agent()
```

Then you can dynamically add various attributes to this `agent` object `a` :

Code 19.3:

```
>>> a.x = 2
>>> a.y = 8
>>> a.name = 'John'
>>> a.age = 21
>>> a.x
2
>>> a.name
'John'
```

This flexibility is very similar to the flexibility of Python's dictionary. You don't have to predefine attributes of a Python object. As you assign a value to an attribute (written as "object's name"."attribute"), Python automatically generates a new attribute if it hasn't been defined before. If you want to know what kinds of attributes are available under an object, you can use the `dir` command:

Code 19.4:

```
>>> dir(a)
['__doc__', '__module__', 'age', 'name', 'x', 'y']
```

The first two attributes are Python's default attributes that are available for any objects (you can ignore them for now). Aside from those, we see there are four attributes defined for this object `a` .

n the rest of this chapter, we will use this class-based agent representation. The technical architecture of simulator codes is still the same as before, made of three components: initialization, visualization, and updating functions. Let's work on some examples to see how you can build an ABM in Python.

Schelling's Segregation Model

There is a perfect model for our first ABM exercise. It is called *Schelling's segregation model*, widely known as the very first ABM proposed in the early 1970s by Thomas Schelling, the 2005 Nobel Laureate in Economics [85]. Schelling created this model in order to provide an explanation for why people with different ethnic backgrounds tend to segregate geographically. Therefore, this model was developed in approach B discussed above, reproducing the macro-known by using hypothetical micro-unknowns. The model assumptions Schelling used were the following:

- Two different types of agents are distributed in a finite 2-D space.
- In each iteration, a randomly chosen agent looks around its neighborhood, and if the fraction of agents of the same type among its neighbors is below a threshold, it jumps to another location randomly chosen in the space.

As you can see, the rule of this model is extremely simple. The main question Schelling addressed with this model was how high the threshold had to be in order for segregation to occur. It may sound reasonable to assume that segregation would require highly homophilic agents, so the critical threshold might be relatively high, say 80% or so. But what Schelling actually showed was that the critical threshold can be much lower than one would expect. This means that segregation can occur even if people aren't so homophilic. In the meantime, quite contrary to our intuition, a high level of homophily can actually result in a mixed state of the society because agents keep moving without reaching a stationary state. We can observe these emergent behaviors in simulations.

Back in the early 1970s, Schelling simulated his model on graph paper using pennies and nickels as two types of agents (this was still a perfectly computational simulation!). But here, we can loosen the spatial constraints and simulate this model in a continuous space. Let's design the simulation model step by step, going through the design tasks listed above.

1. Design the data structure to store the attributes of the agents. In this model, each agent has a type attribute as well as a position in the 2-D space. The two types can be represented by 0 and 1, and the spatial position can be anywhere within a unit square. Therefore we can generate each agent as follows:

Code 19.5:

```
class agent:
    pass

ag = agent()
ag.type = randint(2)
ag.x = random()
ag.y = random()
```

To generate a population of agents, we can write something like:

Code 19.6:

```
n = 1000 # number of agents

class agent:
    pass

def initialize():
    global agents
    agents = []
    for i in xrange(n):
        ag = agent()
        ag.type = randint(2)
        ag.x = random()
        ag.y = random()
        agents.append(ag)
```

2. Design the data structure to store the states of the environment, 3. Describe the rules for how the environment behaves on its own, & 4. Describe the rules for how agents interact with the environment. Schelling's model doesn't have a separate environment that interacts with agents, so we can skip these design tasks.

5. Describe the rules for how agents behave on their own. We assume that agents don't do anything by themselves, because their actions (movements) are triggered only by interactions with other agents. So we can ignore this design task too.

6. Describe the rules for how agents interact with each other. Finally, there is something we need to implement. The model assumption says each agent checks who are in its neighborhood, and if the fraction of the other agents of the same type is less than a threshold, it jumps to another randomly selected location. This requires *neighbor detection*, which was easy in CA and networks because the neighborhood relationships were explicitly modeled in those modeling frameworks. But in ABM, neighborhood relationships may be implicit, which is the case for our model. Therefore we need to implement a code that allows each agent to find who is nearby.

There are several computationally efficient algorithms available for neighbor detection, but here we use the simplest possible method: Exhaustive search. You literally check all the agents, one by one, to see if they are close enough to the focal agent. This is not computationally efficient (its computational amount increases quadratically as the number of agents increases), but is very straightforward and extremely easy to implement. You can write such exhaustive neighbor detection using Python's list comprehension, e.g.:

Code 19.7:

```
neighbors = [nb for nb in agents
             if (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < r**2 and nb != ag]
```

Here `ag` is the focal agent whose neighbors are searched for. The `if` part in the list comprehension measures the distance squared between `ag` and `nb`, and if it is less than `r` squared (`r` is the neighborhood radius, which must be defined earlier in the code) `nb` is included in the result. Also note that an additional condition `nb != ag` is given in the if part. This is because if `nb == ag`, the distance is always 0 so `ag` itself would be mistakenly included as a neighbor of `ag`.

Once we obtain `neighbors` for `ag`, we can calculate the fraction of the other agents whose type is the same as `ag`'s, and if it is less than a given threshold, `ag`'s position is randomly reset. Below is the completed simulator code, with the visualization function also implemented using the simple plot function:

Code 19.8: segregation.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 1000 # number of agents
r = 0.1 # neighborhood radius
th = 0.5 # threshold for moving

class agent:
    pass
```

```
def initialize():
    global agents
    agents = []
    for i in xrange(n):
        ag = agent()
        ag.type = randint(2)
        ag.x = random()
        ag.y = random()
        agents.append(ag)
```

```
def observe():
    global agents
    cla()
    white = [ag for ag in agents if ag.type == 0]
    black = [ag for ag in agents if ag.type == 1]
    plot([ag.x for ag in white], [ag.y for ag in white], 'wo')
    plot([ag.x for ag in black], [ag.y for ag in black], 'ko')
    axis('image')
    axis([0, 1, 0, 1])
```

```
def update():
    global agents
    ag = agents[randint(n)]
    neighbors = [nb for nb in agents
                 if (ag.x - nb.x)**2 + (ag.y - nb.y)**2 < r**2 and nb != ag]
    if len(neighbors) > 0:
        q = len([nb for nb in neighbors if nb.type == ag.type]) \
            / float(len(neighbors))
        if q < th:
            ag.x, ag.y = random(), random()

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

When you run this code, you should set the step size to 50 under the "Settings" tab to speed up the simulations.

Figure 19.2.1 shows the result with the neighborhood radius `r=0.1` and the threshold for moving `th = 0.5`. It is clearly observed that the agents self-organize from an initially random distribution to a patchy pattern where the two types are clearly segregated from each other.

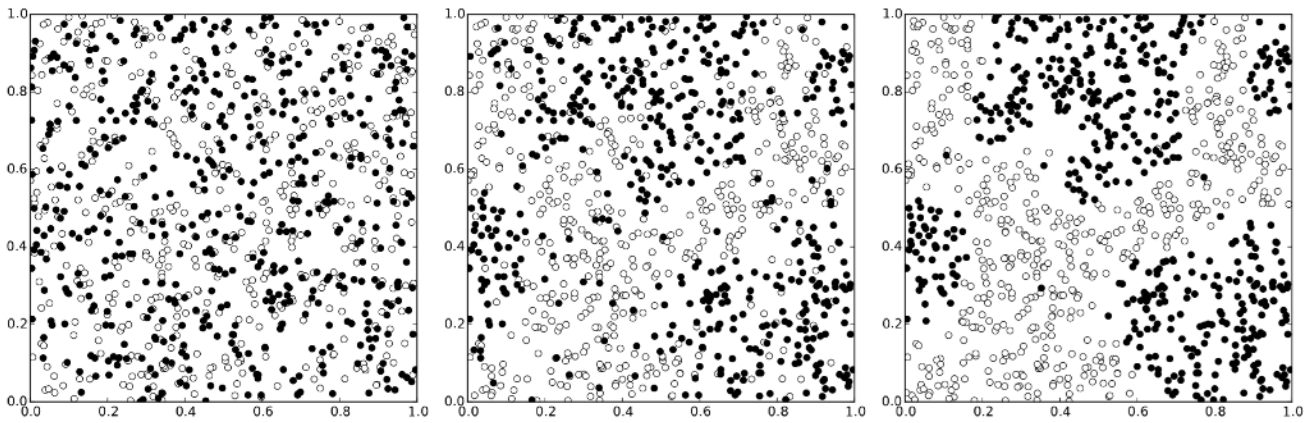


Figure **19.2.1**: Visual output of Code 19.8. Time flows from left to right.

? Exercise 19.2.1

Conduct simulations of Schelling's segregation model with `th` (threshold for moving), `r` (neighborhood radius), and/or `n` (population size = density) varied systematically. Determine the condition in which segregation occurs. Is the transition gradual or sharp?

? Exercise 19.2.2

Develop a metric that characterizes the level of segregation from the positions of the two types of agents. Then plot how this metric changes as parameter values are varied.

Here are some other well-known models that show quite unique emergent patterns or dynamic behaviors. They can be implemented as an ABM by modifying the code for Schelling's segregation model. Have fun!

? Exercise 19.2.3: Diffusion-limited aggregation

Diffusion-limited aggregation (DLA) is a growth process of clusters of aggregated particles driven by their random diffusion. There are two types of particles, like in Schelling's segregation model, but only one of them can move freely. The movable particles diffuse in a 2-D space by random walk, while the immovable particles do nothing; they just remain where they are. If a movable particle "collides" with an immovable particle (i.e., if they come close enough to each other), the movable particle becomes immovable and stays there forever. This is the only rule for the agents' interaction.

Implement the simulator code of the DLA model. Conduct a simulation with all particles initially movable, except for one immovable "seed" particle placed at the center of the space, and observe what kind of spatial pattern emerges. Also carry out the simulations with multiple immovable seeds randomly positioned in the space, and observe how multiple clusters interact with each other at macroscopic scales.

For your information, a completed Python simulator code of the DLA model is available from http://sourceforge.net/projects/pycx/files/, but you should try implementing your own simulator code first.

? Exercise 19.2.4: Boids

This is a fairly advanced, challenging exercise about the collective behavior of animal groups, such as bird flocks, fish schools, and insect swarms, which is a popular subject of research that has been extensively modeled and studied with ABMs. One of the earliest computational models of such collective behaviors was proposed by computer scientist Craig Reynolds in the late 1980s[86]. Reynolds came up with a set of simple behavioral rules for agents moving in a continuous space that can reproduce an amazingly natural-looking flock behavior of birds. His model was called bird-oids, or "Boids" for short. Algorithms used in Boids has been utilized extensively in the computer graphics industry to automatically generate animations of natural movements of animals in flocks (e.g., bats in the Batman movies). Boids' dynamics are generated by the following three essential behavioral rules (Fig. 19.2.1):

- *Cohesion Agents* tend to steer toward the center of mass of local neighbors.
- *Alignment Agents* tend to steer to align their directions with the average velocity of local neighbors.
- *Separation Agents* try to avoid collisions with local neighbors.

Design an ABM of collective behavior with these three rules, and implement its simulator code. Conduct simulations by systematically varying relative strengths of the three rules above, and see how the collective behavior changes.

You can also simulate the collective behavior of a population in which multiple types of agents are mixed together. It is known that interactions among kinetically distinct types of swarming agents can produce various nontrivial dynamic patterns [87].

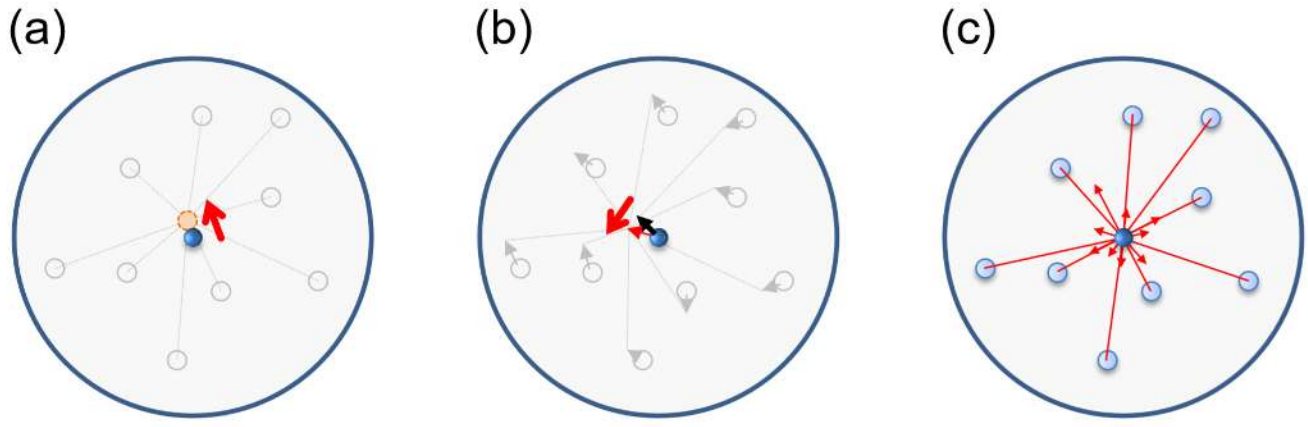


***Figure* 19.2.2** Three essential behavioral rules of Boids. (a) Cohesion. (b) Alignment. (c) Separation.

This page titled 19.2: Building an Agent-Based Model is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

19.3: Agent-Environment Interaction

One important component you should consider adding to your ABM is the interaction between agents and their environment. The environmental state is still part of the system's overall state, but it is defined over space, and not associated with specific agents. The environmental state dynamically changes either spontaneously or by agents' actions (or both). The examples discussed so far (Schelling's segregation model, DLA, Boids) did not include such an environment, but many ABMs explicitly represent environments that agents act on and interact with. The importance of agent-environment interaction is well illustrated by the fact that NetLogo [13], a popular ABM platform, uses "turtles" and "patches" by default, to represent agents and the environment, respectively. We can do the same in Python.

A good example of such agent-environment interaction is in the *Keller-Segel slime mold aggregation model* we discussed in Section 13.4, where slime mold cells behave as agents and interact with an environment made of cAMP molecules. The concentration of cAMP is defined everywhere in the space, and it changes by its own inherent dynamics (natural decay) and by the actions of agents (secretion of cAMP molecules by agents). This model can be implemented as an ABM, designed step by step as follows:

1. Design the data structure to store the attributes of the agents. If the slime mold cells are represented by individual agents, their concentration in the original Keller-Segel model is represented by the density of agents, so they will no longer have any attributes other than spatial position in the 2-D space. Therefore `x` and `y` are the only attributes of agents in this model.

2. Design the data structure to store the states of the environment. The environment in this model is the spatial function that represents the concentration of cAMP molecules at each location. We can represent this environment by discretizing space and assigning a value to each discrete spatial cell, just like we did for numerical simulations of PDEs. We can use the `array` data structure for this purpose.

Here is a sample `initialize` part that sets up the data structures for both agents and the environment. Note that we prepare two arrays, `env` and `nextenv`, for simulating the dynamics of the environment.

Code 19.9:

```
n = 1000 # number of agents
w = 100 # number of rows/columns in spatial array

class agent:
    pass

def initialize():
    global agents, env, nextenv

    agents = []
    for i in xrange(n):
        ag = agent()
        ag.x = randint(w)
        ag.y = randint(w)
        agents.append(ag)
```

```
    env = zeros([w, w])
    nextenv = zeros([w, w])
```

3. Describe the rules for how the environment behaves on its own. The inherent dynamics of the environment in this model are the diffusion and spontaneous decay of the cAMP concentration. These can be modeled by using the discrete version of the Laplacian operator, as well as an exponential decay factor, applied to the environmental states everywhere in the space in every iteration. This is no different from what we did for the numerical simulations of PDEs. We can implement them in the code as follows:

Code 19.10:

```
k = 1 # rate of cAMP decay
Dc = 0.001 # diffusion constant of cAMP
Dh = 0.01 # spatial resolution for cAMP simulation
Dt = 0.01 # time resolution for cAMP simulation

def update():
    global agents, env, nextenv

    # simulating diffusion and evaporation of cAMP
    for x in xrange(w):
        for y in xrange(w):
            C, R, L, U, D = env[x,y], env[(x+1)%w,y], env[(x-1)%w,y], \
                            env[x,(y+1)%w], env[x,(y-1)%w]
            lap = (R + L + U + D - 4 * C)/(Dh**2)
            nextenv[x,y] = env[x,y] + (- k * C + Dc * lap) * Dt
    env, nextenv = nextenv, env
```

Here we adopt periodic boundary conditions for simplicity.

4. Describe the rules for how agents interact with the environment. In this model, agents interact with the environment in two different ways. One way is the secretion of cAMP by the agents, which can be implemented by letting each agent increase the cAMP concentration in a discrete cell where it is located. To do this, we can add the following to the update function:

Code 19.11:

```
    f = 1 # rate of cAMP secretion by an agent

    # simulating secretion of cAMP by agents
    for ag in agents:
        env[ag.x, ag.y] += f * Dt
```

The other way is the chemotaxis, which can be implemented in several different ways. For example, we can have each agent look at a cell randomly chosen from its neighborhood, and move there with a probability determined by the difference in cAMP concentration (Δc) between the neighbor cell and the cell where the agent is currently located. A *sigmoid function*

$$P(\Delta c) = \frac{e^{\Delta c/c_0}}{1 + e^{\Delta c/c_0}} \tag{19.3.1}$$

would be suitable for this purpose, where c_o is a parameter that determines how sensitive this probability is to Δc. $P(\Delta c)$ approaches 1 with $\Delta c \to \infty$, or 0 with $\Delta c \to -\infty$. This part can be implemented in the update function as follows:

Code 19.12:

```
# simulating chemotaxis of agents
for ag in agents:
    newx, newy = (ag.x + randint(-1, 2)) % w, (ag.y + randint(-1, 2)) % w
    diff = (env[newx, newy] - env[ag.x, ag.y]) / 0.1
    if random() < exp(diff) / (1 + exp(diff)):
        ag.x, ag.y = newx, newy
```

Here diff corresponds to Δc, and we used $c_0 = 0.1$.

5. Describe the rules for how agents behave on their own. All the actions taken by the agents in this model are interactions with the environment, so we can skip this design task.

6. Describe the rules for how agents interact with each other. In this model, agents don't interact with each other directly; all interactions among them are indirect, mediated by environmental variables (cAMP concentration in this case), so there is no need to implement anything for the agents' direct interaction with each other. Indirect agent-agent interaction through informational signals written in the environment is called *stigmergy*, which is an essential coordination mechanism used by many social organisms [88].

By putting everything together, and adding the `observe` function for visualization, the entire simulator code looks like this:

Code 19.13: keller-segel-abm.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
```

```
n = 1000 # number of agents
w = 100 # number of rows/columns in spatial array

k = 1 # rate of cAMP decay
Dc = 0.001 # diffusion constant of cAMP
Dh = 0.01 # spatial resolution for cAMP simulation
Dt = 0.01 # time resolution for cAMP simulation

f = 1 # rate of cAMP secretion by an agent

class agent:
    pass

def initialize():
    global agents, env, nextenv
```

```
    agents = []
    for i in xrange(n):
        ag = agent()
        ag.x = randint(w)
        ag.y = randint(w)
        agents.append(ag)

    env = zeros([w, w])
    nextenv = zeros([w, w])

def observe():
    global agents, env, nextenv
    cla()
    imshow(env, cmap = cm.binary, vmin = 0, vmax = 1)
    axis('image')
    x = [ag.x for ag in agents]
```

```
    y = [ag.y for ag in agents]
    plot(y, x, 'b.') # x and y are swapped to match the orientation of env

def update():
    global agents, env, nextenv
```

```
    # simulating diffusion and evaporation of cAMP
    for x in xrange(w):
        for y in xrange(w):
            C, R, L, U, D = env[x,y], env[(x+1)%w,y], env[(x-1)%w,y], \
                            env[x,(y+1)%w], env[x,(y-1)%w]
            lap = (R + L + U + D - 4 * C)/(Dh**2)
            nextenv[x,y] = env[x,y] + (- k * C + Dc * lap) * Dt
    env, nextenv = nextenv, env

    # simulating secretion of cAMP by agents
    for ag in agents:
        env[ag.x, ag.y] += f * Dt

    # simulating chemotaxis of agents
    for ag in agents:
        newx, newy = (ag.x + randint(-1, 2)) % w, (ag.y + randint(-1, 2)) % w
        diff = (env[newx, newy] - env[ag.x, ag.y]) / 0.1
        if random() < exp(diff) / (1 + exp(diff)):
            ag.x, ag.y = newx, newy

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

As you see, the code is getting longer and more complicated than before, which is a typical consequence when you implement an ABM.

Figure 19.3.1 shows a typical simulation result. The tiny blue dots represent individual cells (agents), while the grayscale shades on the background show the cAMP concentration (environment). As time goes by, the slightly more populated areas produce more cAMP, attracting more cells. Eventually, several distinct peaks of agent populations are spontaneously formed, reproducing self-organizing patterns that are similar to what the PDE-based model produced.

What is unique about this ABM version of the Keller-Segel model, compared to its original PDE-based version, is that the simulation result looks more “natural”—the formation of the population peaks are not simultaneous, but they gradually appear one after another, and the spatial arrangements of those peaks are not regular. In contrast, in the PDE-based version, the spots are formed simultaneously at regular intervals (see

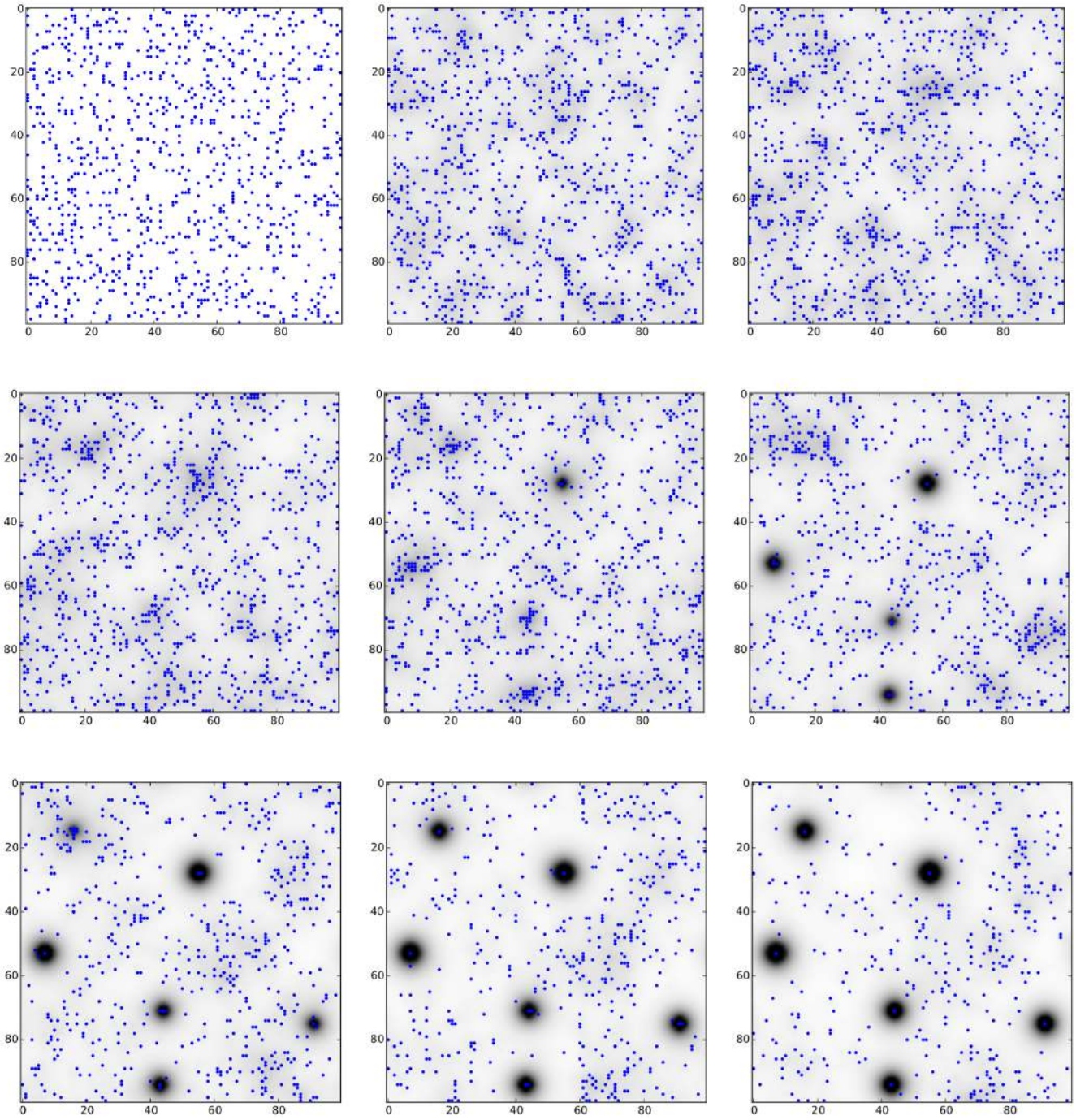


Figure 19.3.1: Visual output of Code 19.13. Time flows from left to right, then from top to bottom.

 Fig. *13.4.3*). Such “natural-lookingness” of the results obtained with ABMs comes from the fact that ABMs are based on the behaviors of discrete individual agents, which often involve a lot of realistic uncertainty.

? Exercise 19.3.1

Conduct simulations of the ABM version of the Keller-Segel model with `k`, `Dc`, and `f` varied systematically. Are the effects of those parameters on pattern formation similar to those in the PDE version of the same model (see Eq. (14.3.40))?

https://math.libretexts.org/@go/page/7887

? Exercise 19.3.2

Implement an additional mechanism into the ABM above to prevent agents from aggregating too densely at a single location, so that the population density doesn't exceed a predetermined upper limit anywhere in the space. Conduct simulations to see what kind of effect this additional model assumption has on pattern formation.

? Exercise 19.3.3

Garbage collection by ants This is another interesting ABM with an agent-environment interaction, which was first presented by Mitchel Resnick in his famous book "Turtles, Termites and Traffic Jams" [89]. Assume there are many tiny pieces of garbage scattered in a 2-D space, where many ants are wandering randomly. When an ant comes to a place where there is some garbage, it behaves according to the following very simple rules:

1. If the ant is holding a piece of garbage, it drops the piece.
2. If the ant isn't holding any garbage, it picks up a piece of garbage.

What would result from these rules? Are the garbage pieces going to be scattered more and more due to these brainless insects? Implement an ABM of this model and conduct simulations to find out what kind of collective behavior emerges.

If you implement the model right, you will see that these very simple behavioral rules let the ants spontaneously collect and pile up garbage and clear up the space in the long run. This model tells us how such emergent behavior of the collective is sometimes counter to our natural intuition.

This page titled 19.3: Agent-Environment Interaction is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

19.4: Ecological and Evolutionary Models

In this very final section of this textbook, we will discuss ABMs of ecological and evolutionary dynamics. Such ABMs are different from the other examples discussed so far in this chapter regarding one important aspect: Agents can be born and can die during a simulation. This means that the number of state variables involved in a system can change dynamically overtime, so the traditional concepts of dynamical systems don't apply easily to those systems. You may remember that we saw a similar challenge when we made a transition from dynamics on networks to dynamics of networks in Chapter 16. A dynamic increase or decrease in the number of system components violates the assumption that the system's behavior can be represented as a trajectory within a static phase space. In order to study the behaviors of such systems, the most general, practical approach would probably be to conduct explicit simulations on computers.

Simulating an ABM with a varying number of agents requires special care for simulating births and deaths of agents. Because agents can be added to or removed from the system at any time during an updating process, it is a little tricky to implement synchronous updating of agents (though it isn't impossible). It is much simpler and more straightforward to update the system's state in an asynchronous manner, by randomly choosing an agent to update its state and, if needed, directly remove it from the system (simulation of death) or add a new agent to the system (simulation of birth). In what follows, we will adopt this asynchronous updating scheme for the simulation of ecological ABMs.

An illustrative example of ecological ABMs is the predator-prey ecosystem, which we already discussed in Section 4.6 and on several other occasions. The basic idea of this model is still simple: Prey naturally grow but get eaten by predators, while the predators grow if they get prey but otherwise naturally die off. When we are to implement this model as an ABM, these ecological dynamics should be described at an individual agent level, not at an aggregated population level. Let's design this ABM step by step, again going through the six design tasks.

1. Design the data structure to store the attributes of the agents. The predator-prey ecosystem is obviously made of two types of agents: prey and predators. So the information about agent type must be represented in the data structure. Also, if we are to simulate their interactions in a space, the information about their spatial location is also needed. Note that these attributes are identical to those of the agents in Schelling's segregation model, so we can use the same agent design, as follows:

Here, we call prey "rabbits" and predators "foxes" in the code, so we can denote them by `r` and `f`, respectively (as both prey and predators begin with "pre"!). Also, we use `r_init` and `f_init` to represent the initial population of each species. The `for` loop iterates `r_init + f_init` times, and in the first `r_init` iteration, the prey agents are generated, while the predator agents are generated for the rest.

2. Design the data structure to store the states of the environment, 3. Describe the rules for how the environment behaves on its own, & 4. Describe the rules for how agents interact with the environment. This ABM doesn't involve an environment explicitly, so we can ignore these design tasks.

5. Describe the rules for how agents behave on their own, & 6. Describe the rules for how agents interact with each other. In this model, the agents' inherent behaviors and interactions are some what intertwined, so we will discuss these two design tasks together. Different rules are to be designed for prey and predator agents, as follows.

For prey agents, each individual agent reproduces at a certain reproduction rate. In equation-based models, it was possible to allow a population to grow exponentially, but in ABMs, exponential growth means exponential increase of memory use because each agent physically takes a certain amount of memory space in your computer. Therefore we need to prevent such growth of memory use by applying a logistic-type growth restriction. In the meantime, if a prey agent meets a predator agent, it dies with some probability because of predation. Death can be implemented simply as the removal of the agent from the `agents` list.

For predator agents, the rules are somewhat opposite. If a predator agent can't find any prey agent nearby, it dies with some probability because of the lack of food. But if it can consume prey, it can also reproduce at a certain reproduction rate.

Finally, both types of agents diffuse in space by random walk. The rates of diffusion can be different between the two species, so let's assume that predators can diffuse a little faster than prey.

The assumptions designed above can be implemented altogether in the update function as follows. As you can see, the code is getting a bit longer than before, which reflects the increased complexity of the agents' behavioral rules:

Here, `ag` is the agent randomly selected for asynchronous updating. Python's `copy` module is used to create a copy of each agent as offspring when it reproduces. Note that the logistic-type growth restriction is implemented for prey agents by multiplying the reproduction probability by $(1 - x/n_r)$, where x is the current population of prey and n_r is the carrying capacity. Also note that at the very beginning of this function, it is checked whether there is any agent in the `agents` list. This is because there is a possibility for all agents to die out in ecological ABMs.

Now, I would like to bring up one subtle issue that arises in ABMs with a varying number of agents that are simulated asynchronously. When the number of agents was fixed and constant, the length of elapsed time in the simulated world was linearly proportional to the number of executions of the asynchronous update function (e.g., in Schelling's segregation model), so we didn't have to do anything special to handle the flow of time. However, when the number of agents varies, an execution of the asynchronous update function on one randomly selected agent doesn't always represent the same amount of elapsed time in the simulated world. To better understand this issue, imagine two different situations: Simulating 10 agents and simulating 1,000 agents. In the former situation, each agent is updated once, on average, when the update function is executed 10 times. However, in the latter situation, 10 times of execution of the function means only about 1% of the agents being updated. But in the simulated world, agents should be behaving concurrently in parallel, so each agent should be updated once, on average, in one unit length of simulated time. This implies that the elapsed time per each asynchronous updating should depend on the size of the agent population. How can we cope with this additional complication of the simulation?

A quick and easy solution to this issue is to assume that, in each asynchronous updating, $1/n$ of a unit length of time passes by, where n is the size of the agent population at the time of updating. This method can naturally handle situations where the size of the agent population changes rapidly, and it (almost) guarantees that each agent is updated once, on average, in each unit time length. To implement a simulation for one unit length of time, we can write the following "wrapper" function to make sure that the time in the simulated world elapses by one unit length:
This trick realizes that the progress of time appears steady in the simulation, even if the number of agents changes over time.

Okay, now we are basically done. Putting everything together and adding the visualization function, the entire simulator code looks like this:

A typical simulation result is shown in Figure 19.5. The tiny blue dots represent prey individuals, while the larger red circles represent predators. As you see in the figure, the interplay between prey and predators produces very dynamic spatial patterns, which somewhat resemble the patterns seen in the host-pathogen CA model discussed in Section 11.5. The prey population grows to form clusters (clouds of tiny blue dots), but if they are infested by predators, the predators also grow rapidly to consume the prey, leaving a deserted empty space behind. As a result, the system as a whole begins to show dynamic waves of prey followed by predators. It is clear that spatial distributions of those species are highly heterogeneous. While we observe the periodic wax and wane of these species, as predicted in the equation-based predator-prey models, the ABM version generates far more complex dynamics that involves patial locality and stochasticity. One could argue that the results obtained from this ABM version would be more realistic than those obtained from purely equation-based models.

We can further modify the simulator code of the predator-prey ABM so that it outputs the time series plot of prey and predator populations next to the visualization of the agents' positions. This can be done in a fairly simple revision: We just need to create lists to store time series of prey and predator populations, and then revise the observe function to count them, append the results to the lists, and visualize the lists as time series plots. Here are the updated `initialize` and `observe` functions:

A typical simulation result with this revised code is shown in Fig. 19.6. You can see that the populations of the two species are definitely showing oscillatory dynamics, yet they are nothing like the regular, cyclic ones predicted by equation-based models (e.g., Fig. 4.8). Instead, there are significant fluctuations and the period of the oscillations is not regular either. Spatial extension, discreteness of individual agents, and stochasticity in their behaviors all contribute in making the results of agent-based simulations more dynamic and "realistic."

? Exercise 19.4.1

In the current model settings, the predator-prey ABM occasionally shows extinction of predator agents (or both types of agents). How can you make the coexistence of two species more robust and sustainable? Develop your own strategy (e.g.,

adjusting model parameters, revising agents' behavioral rules, adding structures to space, etc.), implement it in your simulator code, and test how effective it is. Find what kind of strategy works best for the conservation of both species.

? Exercise 19.4.2

Revise the predator-prey ABM so that it also includes a spatially distributed food resource (e.g., grass) that the prey need to consume for their survival and reproduction. This resource can spontaneously grow and diffuse over space, and decrease when eaten by the prey. Conduct simulations of this revised model to see how the introduction of this third species affects the dynamics of the simulated ecosystem.

The final addition we will make to the model is to introduce the evolution of agents. Evolution is a simple yet very powerful dynamical process by which a population of organisms may spontaneously optimize their attributes for their own survival. It consists of the following three components:

Three components of evolution

Inheritance Organisms reproduce offspring whose attributes are similar to their own.
Variation There is some diversity of organisms' attributes within the population, which primarily arises from imperfect inheritance (e.g., mutation).
Selection Different attributes causes different survivability of organisms (fitness).

When all of these three components are present, evolution can occur in any kind of systems, not limited to biological ones but also social, cultural, and informational ones too. In our predator-prey ABM, the selection is already there (i.e., death of agents). Therefore, what we will need to do is to implement inheritance and variation in the model. For example, we can let the diffusion rates of prey and predators evolve spontaneously over time. To make them evolvable, we need to represent them as heritable traits. This can be accomplished by adding

to the `initialize` function, and then replacing

with

in the `update` function. These changes make the magnitude of movement a heritable attribute of individual agents. Finally, to introduce variation of this attribute, we should add some small random number to it when an offspring is born. This can be implemented by replacing

with something like:

There are two places in the code where this replacement is needed, one for prey and another for predators. Note that, with the mutations implemented above, the diffusion rates of agents `m` could become arbitrarily large or small (they could even become negative). This is fine for our purpose, because `m` is used in the `update` function in the form of `uniform(-m, m)` (which works whether `m` is positive or negative). But in general, you should carefully check your code to make sure the agents' evolvable attributes stay within meaningful bounds.

For completeness, here is the revised code for the evolutionary predator-prey ABM (with the revised parts indicated by ###):

You can conduct simulations with this revised code to see how the prey's and predators' mobilities evolve over time. Is the result consistent with or counter to your initial prediction? I hope you find some interesting discoveries.

? Exercise 19.4.1

Revise the `observe` function of the evolutionary predator-prey ABM developed above so that you can see the distributions of m among prey and predators. Observe how the agents' mobilities spontaneously evolve in a simulation. Are they converging to certain values or continuously changing?

? Exercise 19.4.2

Conduct systematic simulations using both the original and evolutionary predator-prey ABMs to quantitatively assess whether the introduction of evolution into the model has made the coexistence of two species more robust and sustainable or not.

? Exercise 19.4.3

Make the reproduction rates of prey and predator agents also evolvable, in addition to the magnitude of their movements. Conduct simulations to see how the reproduction rates evolve over time.

Believe it or not, we are now at the end of our over-450-page-long journey. Thanks for exploring the world of complex systems with me. I hope you enjoyed it. Needless to say, this is just the beginning for you to dive into a much larger, unexplored territory of complex systems science. I believe there are tons of new properties of various complex systems still to be discovered, and it would be my utmost pleasure if this textbook was a bit of a help for you to make such discoveries in the coming years. Bon voyage!

This page titled 19.4: Ecological and Evolutionary Models is shared under a CC BY-NC-SA 3.0 license and was authored, remixed, and/or curated by Hiroki Sayama (OpenSUNY) via source content that was edited to the style and standards of the LibreTexts platform.

Index

A

Adjacency list
15.2: Terminologies of Graph Theory

Adjacency matrix
15.2: Terminologies of Graph Theory

Alignment Agents
19.2: Building an Agent-Based Model

Asymptotic Behavior
5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems

Asynchronous cellular automata
11.4: Extensions of Cellular Automata

attractor
3.3: What Can We Learn?

B

basin of attraction
3.3: What Can We Learn?

bifurcation
8: Bifurcations

bifurcation diagram
8.2: Bifurcations in 1-D Continuous-Time Models

boids
19.2: Building an Agent-Based Model

butterfly effect
9.1: Chaos in Discrete-Time Models

C

chaos
9: Chaos

Cohesion Agents
19.2: Building an Agent-Based Model

Complex systems
1.1: Complex Systems in a Nutshell

Continuous Field Models
13.1: Continuous Field Models with Partial Differential Equations
13.5: Simulation of Continuous Field Models
14: Continuous Field Models II - Analysis
14.3: Linear Stability Analysis of Continuous Field Models

contour map
13.2: Fundamentals of Vector Calculus

D

diffusion equation
13.4: Modeling Spatial Movement

Diffusion on Networks
18.2: Diffusion on Networks

Dynamical Networks
16: Dynamical Networks I - Modeling

Dynamical systems theory
3.1: What are Dynamical Systems?

E

Ecological Models
19.4: Ecological and Evolutionary Models

eigenfunction
14.3: Linear Stability Analysis of Continuous Field Models

Emergence
1.1: Complex Systems in a Nutshell

epidemic model
11.5: Examples of Biological Cellular Automata Models
16.2: Simulating Dynamics on Networks

equilibrium points
5.1: Finding Equilibrium Points
7.1: Finding Equilibrium Points

equilibrium states
14.1: Finding Equilibrium States

Erdős–Rényi random graph models
16.3: Simulating Dynamics of Networks

Euler forward method
6.4: Simulating Continuous-Time Models

evolution
19.4: Ecological and Evolutionary Models

Evolutionary Models
19.4: Ecological and Evolutionary Models

F

Fibonacci sequence
4.2: Classifications of Model Equations

Fick's first law of diffusion
13.4: Modeling Spatial Movement

forest fire model
11.5: Examples of Biological Cellular Automata Models

G

game of life
11.3: Simulating Cellular Automata

game theory
1.2: Topical Clusters

gradient vector
13.2: Fundamentals of Vector Calculus

graph theory
15.2: Terminologies of Graph Theory

H

Hopfield network
16.2: Simulating Dynamics on Networks

Hopf bifurcation
8.3: Hopf Bifurcations in 2-D Continuous-Time Models

I

iterative map
3.1: What are Dynamical Systems?
5.1: Finding Equilibrium Points

L

limit cycle
8.3: Hopf Bifurcations in 2-D Continuous-Time Models

linear stability analysis
14.3: Linear Stability Analysis of Continuous Field Models

logistic map
5.5: Variable Rescaling of Discrete-Time Models

Lorenz attractor
9.4: Chaos in Continuous-Time Model

Lorenz equations
9.4: Chaos in Continuous-Time Model

Lyapunov Exponent
9.3: Lyapunov Exponent

M

Majority rule
11.2: Examples of Simple Binary Cellular Automata Rules

N

network
15: Basics of Networks

O

Oregonator
13.6: Reaction-Diffusion Systems

P

Parity rule
11.2: Examples of Simple Binary Cellular Automata Rules

phase space
3.2: Phase Space
3.3: What Can We Learn?
5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model
12.1: Sizes of Rule Space and Phase Space

pitchfork bifurcation
8.2: Bifurcations in 1-D Continuous-Time Models

R

random edge rewiring
16.3: Simulating Dynamics of Networks

Rule Space
12.1: Sizes of Rule Space and Phase Space

S

saddle−node bifurcation
8.2: Bifurcations in 1-D Continuous-Time Models

scalar field
13.3: Visualizing Two-Dimensional Scalar and Vector Field

Separation Agents
19.2: Building an Agent-Based Model

Stochastic cellular automata
11.4: Extensions of Cellular Automata

STRETCHING AND FOLDING PHASE SPACE
9.2: Characteristics of Chaos

subcritical pitchfork bifurcation
8.2: Bifurcations in 1-D Continuous-Time Models

Synchronizability
18.3: Synchronizability

T
time series
4.1: Discrete-Time Models with Difference Equations
transcritical bifurcation
8.2: Bifurcations in 1-D Continuous-Time Models
transport equation
13.4: Modeling Spatial Movement
Turing instability
13.6: Reaction-Diffusion Systems
Turing patterns
11.5: Examples of Biological Cellular Automata Models
V
van der Pol oscillator
8.3: Hopf Bifurcations in 2-D Continuous-Time Models
Variable Rescaling
5.5: Variable Rescaling of Discrete-Time Models
7.3: Variable Rescaling of Continuous-Time Models
14.2: Variable Rescaling
vector fields
13.3: Visualizing Two-Dimensional Scalar and Vector Field
voter model
16.2: Simulating Dynamics on Networks

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Introduction to the Modeling and Analysis of Complex Systems (Sayama)

Webpages: 125

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC-SA 3.0: 84.8% (106 pages)
- Undeclared: 14.4% (18 pages)
- CC BY-NC-SA 4.0: 0.8% (1 page)

By Page

- Introduction to the Modeling and Analysis of Complex Systems (Sayama) - *CC BY-NC-SA 3.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Introduction to Modeling and Analysis - *CC BY-NC-SA 3.0*
    - 1.1: Complex Systems in a Nutshell - *CC BY-NC-SA 3.0*
    - 1.2: Topical Clusters - *CC BY-NC-SA 3.0*
  - 2: Fundamentals of Modeling - *CC BY-NC-SA 3.0*
    - 2.1: Models in Science and Engineering - *CC BY-NC-SA 3.0*
    - 2.2: How to Create a Model - *CC BY-NC-SA 3.0*
    - 2.3: Modeling Complex Systems - *CC BY-NC-SA 3.0*
    - 2.4: What Are Good Models? - *CC BY-NC-SA 3.0*
    - 2.5: A Historical Perspective - *CC BY-NC-SA 3.0*
  - 3: Basics of Dynamical Systems - *CC BY-NC-SA 3.0*
    - 3.1: What are Dynamical Systems? - *Undeclared*
    - 3.2: Phase Space - *CC BY-NC-SA 3.0*
    - 3.3: What Can We Learn? - *Undeclared*
  - 4: Discrete-Time Models I - Modeling - *CC BY-NC-SA 3.0*
    - 4.1: Discrete-Time Models with Difference Equations - *CC BY-NC-SA 3.0*
    - 4.2: Classifications of Model Equations - *CC BY-NC-SA 3.0*
    - 4.3: Simulating Discrete-Time Models with One Variable - *CC BY-NC-SA 3.0*
    - 4.4: Simulating Discrete-Time Models with Multiple Variables - *CC BY-NC-SA 3.0*
    - 4.5: Building Your Own Model Equation - *CC BY-NC-SA 3.0*
    - 4.6: Building Your Own Model Equations with Multiple Variables - *CC BY-NC-SA 3.0*
  - 5: Discrete-Time Models II - Analysis - *CC BY-NC-SA 3.0*
    - 5.1: Finding Equilibrium Points - *CC BY-NC-SA 3.0*
    - 5.2: 5.2 Phase Space Visualization of Continuous-State Discrete-Time Models - *CC BY-NC-SA 3.0*
    - 5.3: 5.3 Cobweb Plots for One-Dimensional Iterative Maps - *CC BY-NC-SA 3.0*
    - 5.4: 5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Model - *CC BY-NC-SA 3.0*
    - 5.5: Variable Rescaling of Discrete-Time Models - *CC BY-NC-SA 3.0*
    - 5.6: Asymptotic Behavior of Discrete-Time Linear Dynamical Systems - *CC BY-NC-SA 3.0*
    - 5.7: 5.7 Linear Stability Analysis of Discrete-Time Nonlinear Dynamical Systems - *CC BY-NC-SA 3.0*
  - 6: Continuous-Time Models I - Modeling - *CC BY-NC-SA 3.0*
    - 6.1: Continuous-Time Models with Differential Equations - *CC BY-NC-SA 3.0*
    - 6.2: Classifications of Model Equation - *CC BY-NC-SA 3.0*
    - 6.3: Connecting Continuous - Time Models with DiscreteTime Models - *CC BY-NC-SA 3.0*
    - 6.4: Simulating Continuous-Time Models - *CC BY-NC-SA 3.0*
    - 6.5: Building Your Own Model Equation - *CC BY-NC-SA 3.0*
  - 7: Continuous-Time Models II - Analysis - *CC BY-NC-SA 3.0*

- 7.1: Finding Equilibrium Points - *CC BY-NC-SA 3.0*
- 7.2: Phase Space Visualization - *CC BY-NC-SA 3.0*
- 7.3: Variable Rescaling of Continuous-Time Models - *CC BY-NC-SA 3.0*
- 7.4: Asymptotic Behavior of Continuous-Time Linear Dynamical Systems - *CC BY-NC-SA 3.0*
- 7.5: Linear Stability Analysis of Nonlinear Dynamical Systems - *CC BY-NC-SA 3.0*

8: Bifurcations - *CC BY-NC-SA 3.0*
- 8.1: 8.1 What are Bifurcations? - *Undeclared*
- 8.2: Bifurcations in 1-D Continuous-Time Models - *CC BY-NC-SA 3.0*
- 8.3: Hopf Bifurcations in 2-D Continuous-Time Models - *CC BY-NC-SA 3.0*
- 8.4: Bifurcations in Discrete-Time Models - *CC BY-NC-SA 3.0*

9: Chaos - *CC BY-NC-SA 3.0*
- 9.1: Chaos in Discrete-Time Models - *CC BY-NC-SA 3.0*
- 9.2: Characteristics of Chaos - *CC BY-NC-SA 3.0*
- 9.3: Lyapunov Exponent - *CC BY-NC-SA 3.0*
- 9.4: Chaos in Continuous-Time Model - *CC BY-NC-SA 3.0*

10: Interactive Simulations of Complex Systems - *CC BY-NC-SA 3.0*
- 10.1: Simulation of Systems with a Large Number of Variables - *CC BY-NC-SA 3.0*
- 10.2: Interactive Simulation with PyCX - *CC BY-NC-SA 3.0*
- 10.3: Interactive Parameter Control in PyCX - *CC BY-NC-SA 3.0*
- 10.4: Simulation without PyCX - *CC BY-NC-SA 3.0*

11: Cellular Automata I - Modeling - *CC BY-NC-SA 3.0*
- 11.1: Definition of Cellular Automata - *CC BY-NC-SA 3.0*
- 11.2: Examples of Simple Binary Cellular Automata Rules - *CC BY-NC-SA 3.0*
- 11.3: Simulating Cellular Automata - *CC BY-NC-SA 3.0*
- 11.4: Extensions of Cellular Automata - *CC BY-NC-SA 3.0*
- 11.5: Examples of Biological Cellular Automata Models - *CC BY-NC-SA 3.0*

12: Cellular Automata II - Analysis - *CC BY-NC-SA 3.0*
- 12.1: Sizes of Rule Space and Phase Space - *CC BY-NC-SA 3.0*
- 12.2: Phase Space Visualization - *CC BY-NC-SA 3.0*
- 12.3: Mean-Field Approximation - *CC BY-NC-SA 3.0*
- 12.4: Renormalization Group Analysis to Predict Percolation Thresholds - *CC BY-NC-SA 3.0*

13: Continuous Field Models I - Modeling - *CC BY-NC-SA 3.0*
- 13.1: Continuous Field Models with Partial Differential Equations - *CC BY-NC-SA 3.0*
- 13.2: Fundamentals of Vector Calculus - *CC BY-NC-SA 3.0*
- 13.3: Visualizing Two-Dimensional Scalar and Vector Field - *CC BY-NC-SA 3.0*
- 13.4: Modeling Spatial Movement - *CC BY-NC-SA 3.0*
- 13.5: Simulation of Continuous Field Models - *CC BY-NC-SA 3.0*
- 13.6: Reaction-Diffusion Systems - *CC BY-NC-SA 3.0*

14: Continuous Field Models II - Analysis - *CC BY-NC-SA 3.0*
- Front Matter - *Undeclared*
  - TitlePage - *Undeclared*
  - InfoPage - *Undeclared*
- 14.1: Finding Equilibrium States - *CC BY-NC-SA 3.0*
- 14.2: Variable Rescaling - *CC BY-NC-SA 3.0*
- 14.3: Linear Stability Analysis of Continuous Field Models - *CC BY-NC-SA 3.0*
- 14.4: Linear Stability Analysis of Reaction-Diffusion Systems - *CC BY-NC-SA 3.0*
- Back Matter - *Undeclared*
  - Index - *Undeclared*

15: Basics of Networks - *CC BY-NC-SA 3.0*
- 15.1: Network Models - *CC BY-NC-SA 3.0*
- 15.2: Terminologies of Graph Theory - *CC BY-NC-SA 3.0*
- 15.3: Constructing Network Models with NetworkX - *CC BY-NC-SA 3.0*
- 15.4: Visualizing Networks with NetworkX - *CC BY-NC-SA 3.0*
- 15.5: Importing/Exporting Network Data - *CC BY-NC-SA 3.0*
- 15.6: Generating Random Graphs - *CC BY-NC-SA 3.0*

16: Dynamical Networks I - Modeling - *CC BY-NC-SA 3.0*
- 16.1: Dynamical Network Models - *CC BY-NC-SA 3.0*
- 16.2: Simulating Dynamics on Networks - *CC BY-NC-SA 3.0*
- 16.3: Simulating Dynamics of Networks - *CC BY-NC-SA 3.0*

- 16.4: Simulating Adaptive Networks - *CC BY-NC-SA 3.0*

- 17: Dynamical Networks II - Analysis of Network Topologies - *CC BY-NC-SA 3.0*
  - 17.1: Network Size, Density, and Percolation - *CC BY-NC-SA 3.0*
  - 17.2: Shortest Path Length - *CC BY-NC-SA 3.0*
  - 17.3: Centralities and Coreness - *CC BY-NC-SA 3.0*
  - 17.4: Clustering - *CC BY-NC-SA 3.0*
  - 17.5: Degree Distribution - *CC BY-NC-SA 3.0*
  - 17.6: Assortativity - *CC BY-NC-SA 3.0*
  - 17.7: Community Structure and Modularity - *CC BY-NC-SA 4.0*
- 18: Dynamical Networks III - Analysis of Network Dynamics - *CC BY-NC-SA 3.0*
  - 18.1: Dynamics of Continuous-State Networks - *CC BY-NC-SA 3.0*
  - 18.2: Diffusion on Networks - *CC BY-NC-SA 3.0*
  - 18.3: Synchronizability - *CC BY-NC-SA 3.0*
  - 18.4: Mean-Field Approximation of Discrete-State Networks - *CC BY-NC-SA 3.0*
  - 18.5: Mean-Field Approximation on Random Networks - *CC BY-NC-SA 3.0*
  - 18.6: Mean-Field Approximation on Scale-Free Networks - *CC BY-NC-SA 3.0*
- 19: Agent-Based Models - *CC BY-NC-SA 3.0*
  - 19.1: What Are Agent-Based Models? - *Undeclared*
  - 19.2: Building an Agent-Based Model - *CC BY-NC-SA 3.0*
  - 19.3: Agent-Environment Interaction - *CC BY-NC-SA 3.0*
  - 19.4: Ecological and Evolutionary Models - *CC BY-NC-SA 3.0*
- Back Matter - *Undeclared*
  - Index - *Undeclared*
  - Glossary - *Undeclared*
  - Detailed Licensing - *Undeclared*