Chapter 4

Discrete-Time Models I: Modeling

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 *difference equations*, because you can rewrite any first-order discrete-time dynamical system with a state variable x (Eq. (3.1)), i.e.,

$$x_t = F(x_{t-1}, t) \tag{4.1}$$

into a "difference" form

$$\Delta x = x_t - x_{t-1} = F(x_{t-1}, t) - x_{t-1}, \tag{4.2}$$

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 Eq. (4.1) can also be written as

$$x_{t+1} = F(x_t, t), \tag{4.3}$$

which is mathematically equivalent to Eq. (4.1) 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:

$$\{x_0, x_1, x_2, x_3, \ldots\} \tag{4.4}$$

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* 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 4.2* 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?

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 4.3 Decide whether each of the following examples is (1) linear or non-linear, (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. $x_t = ax_{t-1} + bxt - 2^2 + c\sqrt{x_{t-1}x_{t-3}}$
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

$$x_t = x_{t-1} + x_{t-2} \tag{4.5}$$

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

$$y_t = x_{t-1} \tag{4.6}$$

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

$$x_t = x_{t-1} + y_{t-1} \tag{4.7}$$

$$y_t = x_{t-1} \tag{4.8}$$

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

$$x_t = x_{t-1} + t \tag{4.9}$$

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

$$z_t = z_{t-1} + 1, \quad z_0 = 1 \tag{4.10}$$

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

$$x_t = x_{t-1} + z_{t-1}, \tag{4.11}$$

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

1. $x_t = x_{t-1}(1 - x_{t-1}) \sin t$
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.

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:

$$x_t = ax_{t-1} \tag{4.12}$$

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 in

your 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]:

[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.

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. 4.1. 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, Eq. (4.12) is a typical mathematical model of exponential growth or exponential decay. You can obtain several distinct behaviors by varying the value of a.

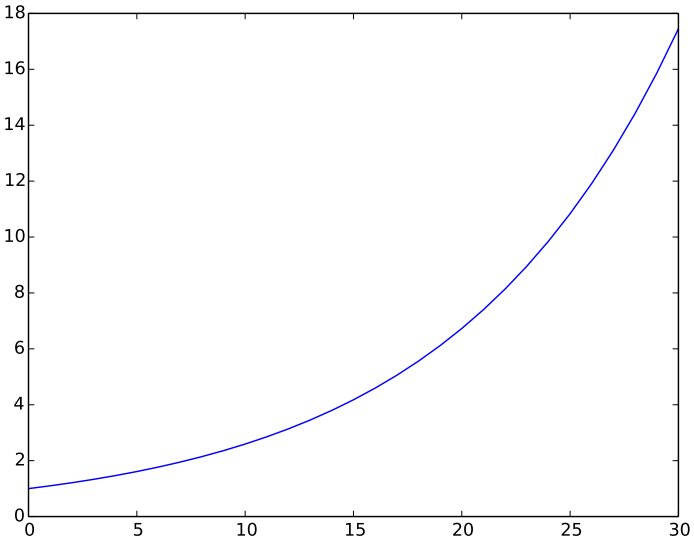


Figure 4.1: Visual output of Code 4.9.

Exercise 4.5 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 4.6 Implement a simulation code of the following difference equation:

$$x_t = ax_{t-1} + b, \quad x_0 = 1 \tag{4.13}$$

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.

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:

$$x_t = 0.5x_{t-1} + y_{t-1} \tag{4.14}$$
$$y_t = -0.5x_{t-1} + y_{t-1} \tag{4.15}$$
$$x_0 = 1, \quad y_0 = 1 \tag{4.16}$$

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

$$\begin{pmatrix} x \\ y \end{pmatrix}_t = \begin{pmatrix} 0.5 & 1 \\ -0.5 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}_{t-1} \tag{4.17}$$

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 `yresult` in a green dashed curve. If you run this code, it produces a decent result (Fig. 4.2), 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:

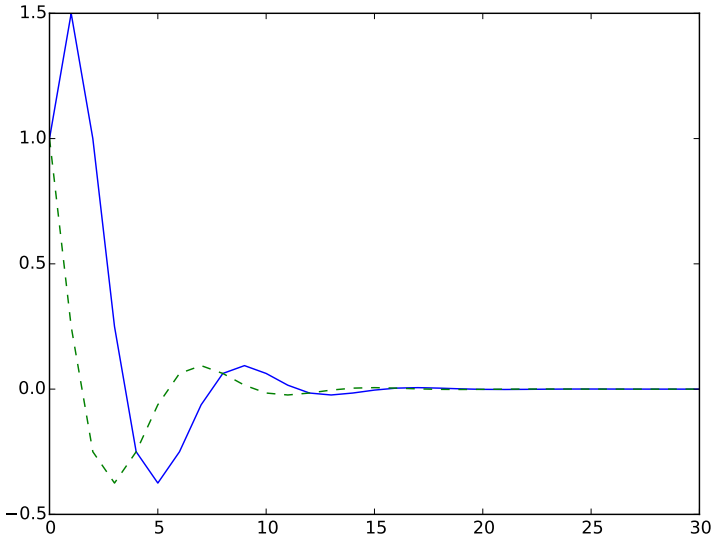


Figure 4.2: Visual output of Code 4.11. This result is actually wrong.

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. 4.3).

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

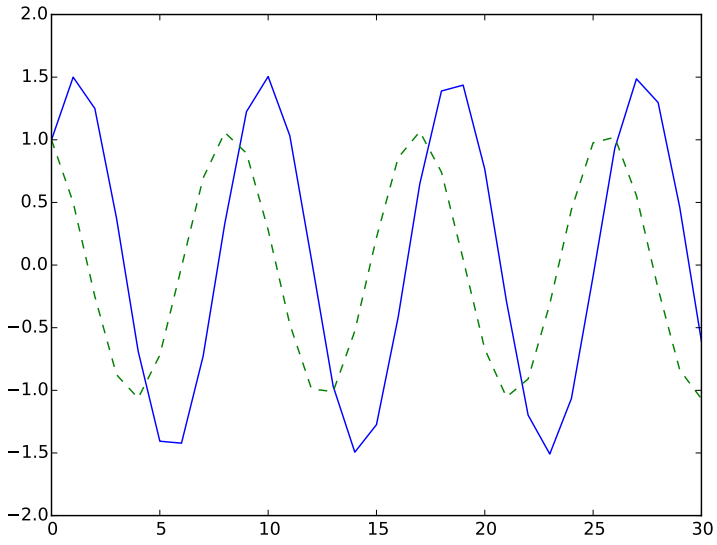


Figure 4.3: Visual output of the simulation result based on a correctly implemented code with Code 4.13.

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), which is a typical signature of a linear system.

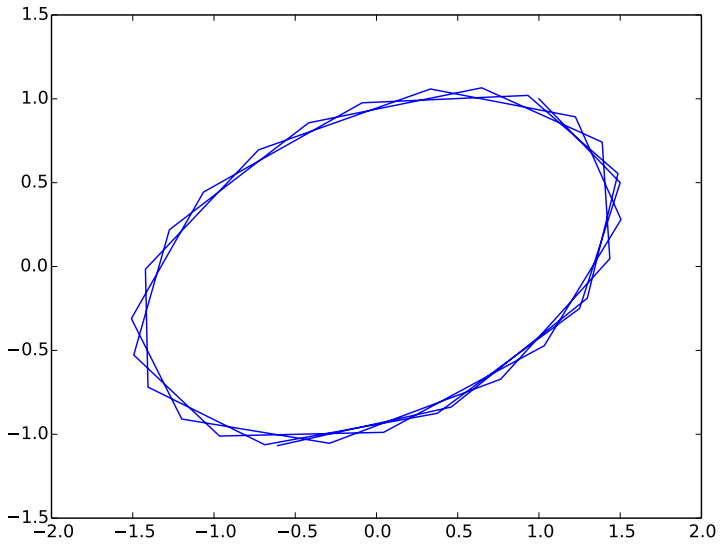


Figure 4.4: Visualization of the simulation result in a phase space using Code 4.15.

Exercise 4.7 Simulate the above two-variable system using several different coefficients in the equations and see what kind of behaviors can arise.

Exercise 4.8 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.

$$x_t = x_{t-1} + x_{t-2}, \quad x_0 = 1, \quad x_1 = 1 \tag{4.18}$$

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.

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:

$$x_t = ax_{t-1} \tag{4.19}$$

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:

$$x_t = f(x_{t-1})x_{t-1} \tag{4.20}$$

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), which is given by

$$f(x) = -\frac{a-1}{K}x + a. \tag{4.21}$$

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

$$x_t = \left(-\frac{a-1}{K}x_{t-1} + a\right) x_{t-1} \tag{4.22}$$

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 $x_{t-1} = 0$? In

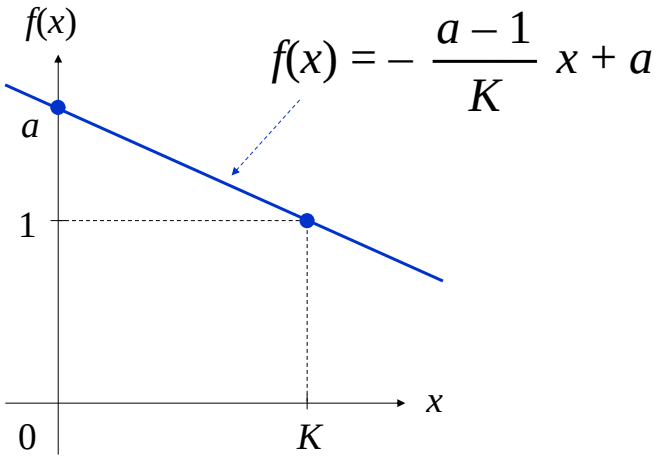


Figure 4.5: The simplest example of how the growth ratio $a = f(x)$ should behave as a function of population size x.

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 4.9 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.

$$x_t = \left(-\frac{a-1}{K} x_{t-1} + a \right) x_{t-1} \tag{4.23}$$

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:

$$x_t = \left(-\frac{a-1}{K}x_{t-1} + a\right) x_{t-1} \tag{4.24}$$

$$= \left(-\frac{r}{K}x_{t-1} + r + 1\right) x_{t-1} \tag{4.25}$$

$$= x_{t-1} + rx_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{4.26}$$

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

$$x_t \approx x_{t-1} + rx_{t-1}. \tag{4.27}$$

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 4.10* 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.

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.

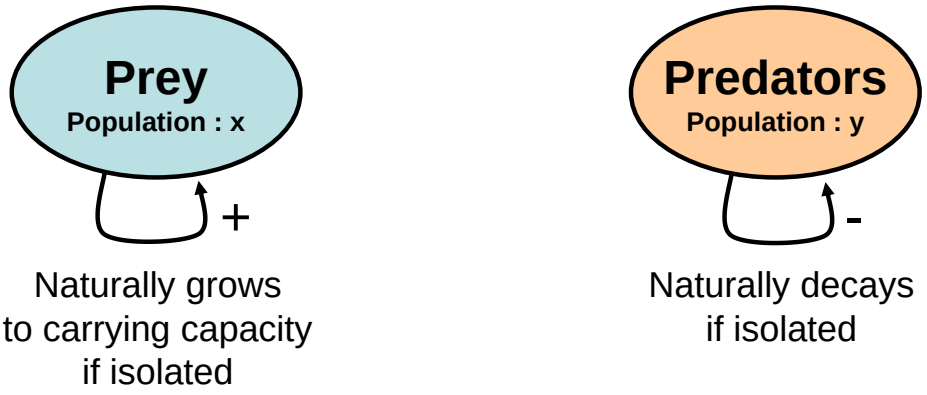


Figure 4.6: Inherent dynamics of each of the prey and predator populations illustrated in a causal loop diagram.

- Predators die if there are no prey.

These assumptions can be diagrammatically illustrated in Fig. 4.6.

This type of diagram is 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. 4.7).

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).

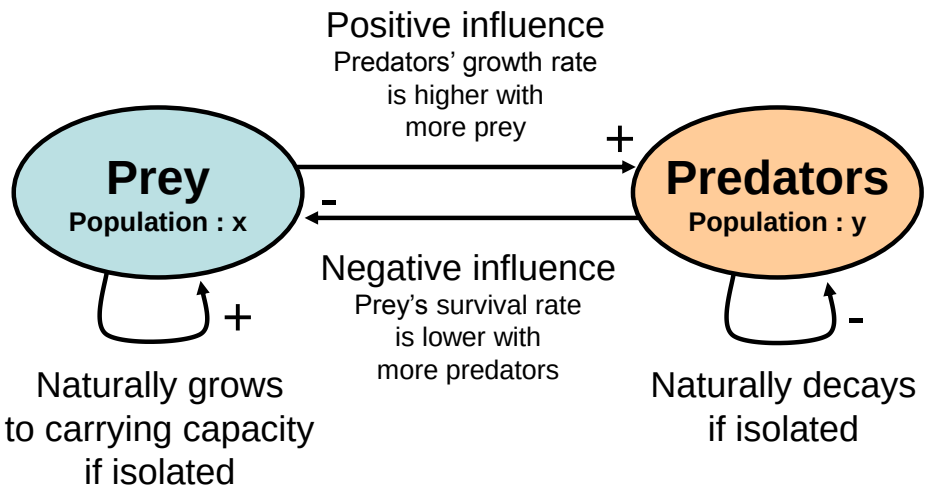


Figure 4.7: Interactions between the prey and predator populations illustrated in a causal loop diagram.

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:

$$x_t = x_{t-1} + r_x x_{t-1}(1 - x_{t-1}/K) \tag{4.28}$$

$$y_t = y_{t-1} - d_y y_{t-1} \tag{4.29}$$

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 ($0 < d_y < 1$).

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:

$$x_t = x_{t-1} + r x_{t-1}(1 - x_{t-1}/K) - d_x(y_{t-1})x_{t-1} \tag{4.30}$$

$$y_t = y_{t-1} - d y_{t-1} + r_y(x_{t-1})y_{t-1} \tag{4.31}$$

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

$$d_x(y) = 1 - \frac{1}{by+1}, \tag{4.32}$$

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

$$r_y(x) = cx, \tag{4.33}$$

where c determines how quickly r_y increases as x increases.

Let's put these functions back into the equations. We obtain the following:

$$x_t = x_{t-1} + rx_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) - \left(1 - \frac{1}{by_{t-1}+1}\right)x_{t-1} \tag{4.34}$$

$$y_t = y_{t-1} - dy_{t-1} + cx_{t-1}y_{t-1} \tag{4.35}$$

Exercise 4.11 Test the equations above by assuming extreme values for x and y, and make sure the model behaves as we intended.

Exercise 4.12 Implement a simulation code for the equations above, and observe the model behavior for various parameter settings.

Figure 4.8 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. 4.8 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.

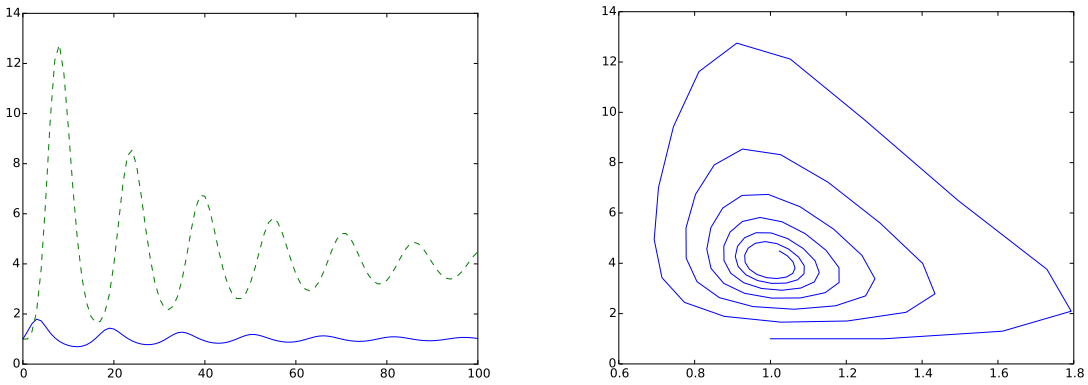


Figure 4.8: Simulation results of the predator-prey model. Left: State variables plotted over time. Blue (solid) = prey, green (dashed) = predators. Right: Phase space visualization of the same result.

Exercise 4.13 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 4.14 Develop a discrete-time mathematical model of two species competing for the same resource, and simulate its behavior.

Exercise 4.15 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_l, respectively ($0 \leq p_c \leq 1$; $0 \leq p_l \leq 1$; $0 \leq p_c + p_l \leq 1$). 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 their relative 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.

Exercise 4.16 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.

Chapter 5

Discrete-Time Models II: Analysis

5.1 Finding Equilibrium Points

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

$$x_t = F(x_{t-1}), \tag{5.1}$$

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

$$x_{\mathrm{eq}} = F(x_{\mathrm{eq}}), \tag{5.2}$$

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

$$x_t = x_{t-1} + r x_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{5.3}$$

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

$$x_{\text{eq}} = x_{\text{eq}} + r x_{\text{eq}} \left(1 - \frac{x_{\text{eq}}}{K}\right) \tag{5.4}$$

$$0 = r x_{\text{eq}} \left(1 - \frac{x_{\text{eq}}}{K}\right) \tag{5.5}$$

$$x_{\text{eq}} = 0, \quad K \tag{5.6}$$

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

Exercise 5.1 Obtain the equilibrium point(s) of the following difference equation:

$$x_t = 2x_{t-1} - x_{t-1}^2 \tag{5.7}$$

Exercise 5.2 Obtain the equilibrium point(s) of the following two-dimensional difference equation model:

$$x_t = x_{t-1} y_{t-1} \tag{5.8}$$

$$y_t = y_{t-1}(x_{t-1} - 1) \tag{5.9}$$

Exercise 5.3 Obtain the equilibrium point(s) of the following difference equation:

$$x_t = x_{t-1} - x_{t-2}^2 + 1 \tag{5.10}$$

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).

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. 4.3. 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. 5.1, which clearly shows that the phase space of this system is made of many concentric trajectories around the origin.

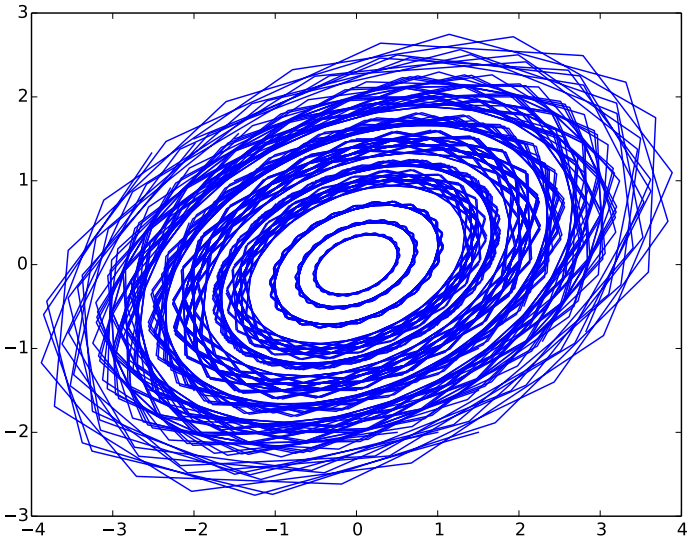


Figure 5.1: Phase space drawn using Code 5.1.

Exercise 5.4 Draw a phase space of the following two-dimensional difference equation model in Python:

$$x_t = x_{t-1} + 0.1(x_{t-1} - x_{t-1}y_{t-1}) \tag{5.11}$$

$$y_t = y_{t-1} + 0.1(y_{t-1} - x_{t-1}y_{t-1}) \tag{5.12}$$

$$(x > 0, \quad y > 0) \tag{5.13}$$

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

$$x_t = 0.5x + y \tag{5.14}$$

$$y_t = -0.5x + y \tag{5.15}$$

$$z_t = -x - y + z \tag{5.16}$$

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.

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. 5.3.

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.

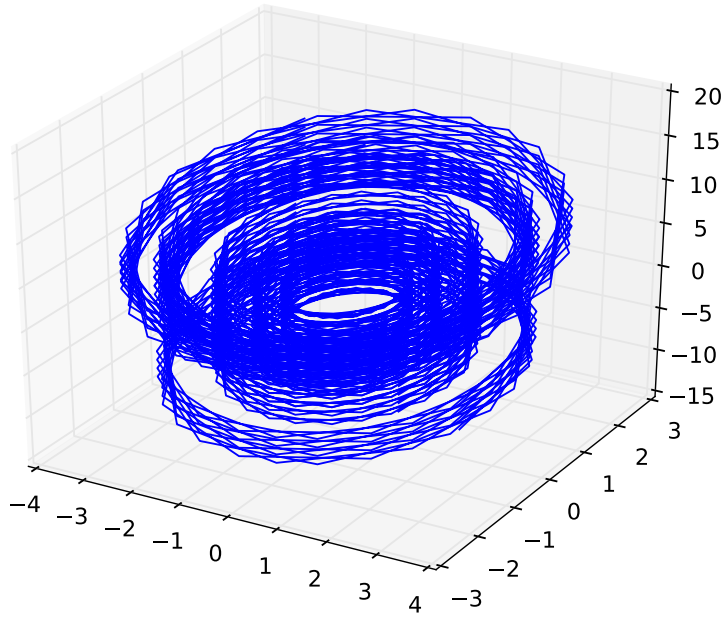


Figure 5.2: Three-dimensional phase space drawn with Code 5.2. If you are drawing this from an interactive environment, such as Anaconda Spyder or Enthought Canopy, you can rotate the 3-D plot interactively by clicking and dragging.

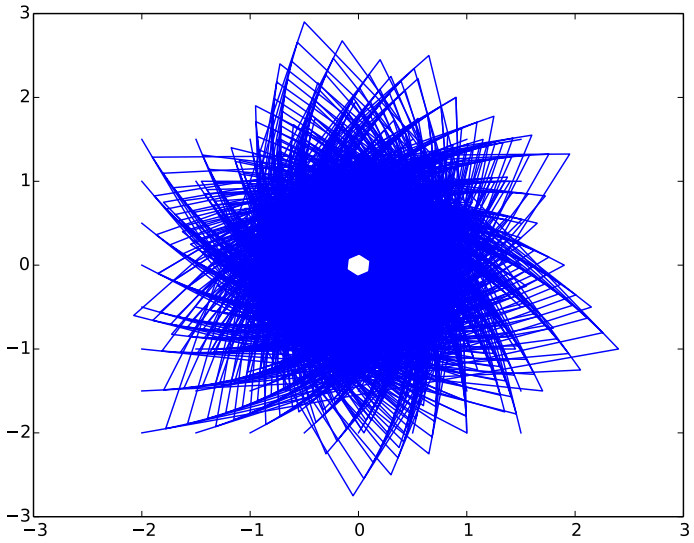


Figure 5.3: Phase space drawn with Code 5.3.

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 four- or 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. 5.4).

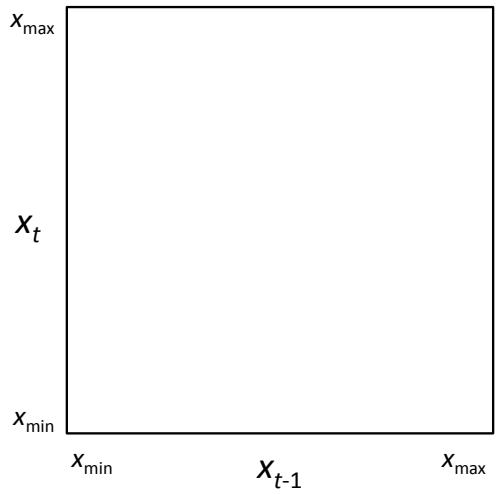


Figure 5.4: Drawing a cobweb plot (1).

2. Draw a curve $x_t = f(x_{t-1})$ and a diagonal line $x_t = x_{t-1}$ within the square (Fig. 5.5). 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. 5.6). Start from a current state value on the bottom axis (initially, this is the initial value x_0, as shown in Fig. 5.6), 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. 5.6). 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. 5.7). This completes one step of the "manual simulation" on the cobweb plot.

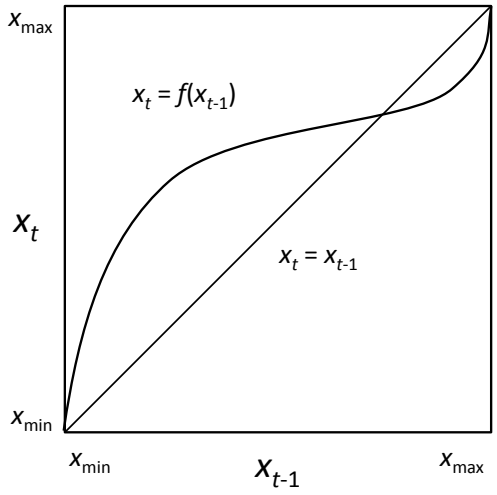


Figure 5.5: Drawing a cobweb plot (2).

5. Repeat the steps above to see where the system eventually goes (Fig. 5.8).

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. 5.9)—*move vertically to the curve, horizontally to the line, and repeat.*

Exercise 5.5 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 5.6 Draw a cobweb plot of the following logistic growth model with $r = 1$, $K = 1$, $N_0 = 0.1$:

$$N_t = N_{t-1} + rN_{t-1}(1 - N_{t-1}/K) \tag{5.17}$$

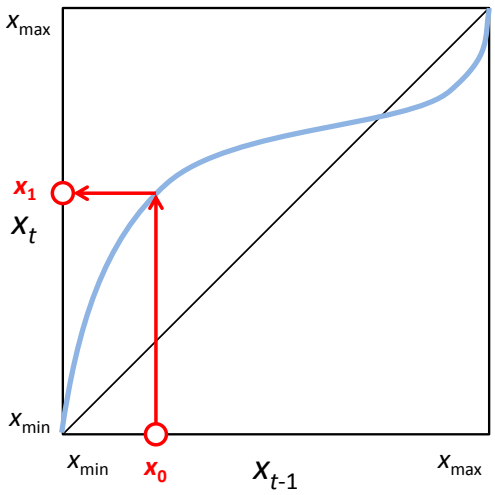


Figure 5.6: Drawing a cobweb plot (3).

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. 5.10.

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)
```

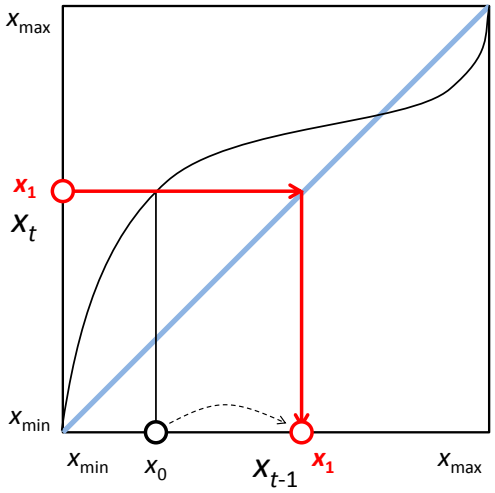


Figure 5.7: Drawing a cobweb plot (4).

```
    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.)
```

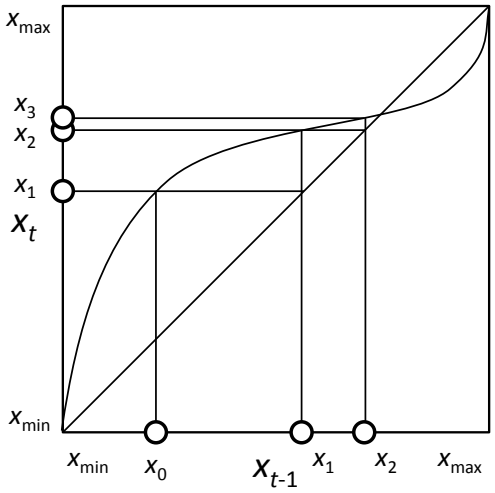


Figure 5.8: Drawing a cobweb plot (5).

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

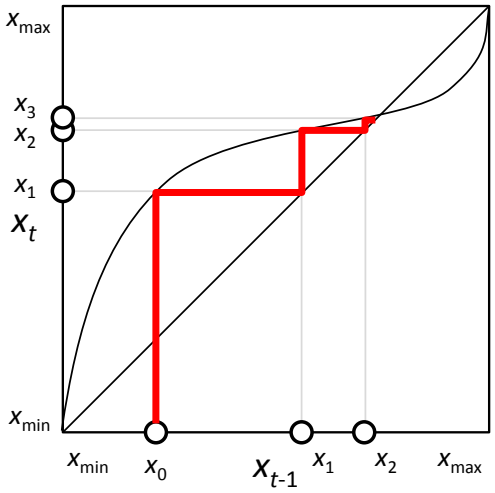


Figure 5.9: Drawing a cobweb plot (6).

5.4 Graph-Based Phase Space Visualization of Discrete-State Discrete-Time Models

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 *if 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:

$$x_t = x_{t-1}x_{t-2} \mod 6 \tag{5.18}$$

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

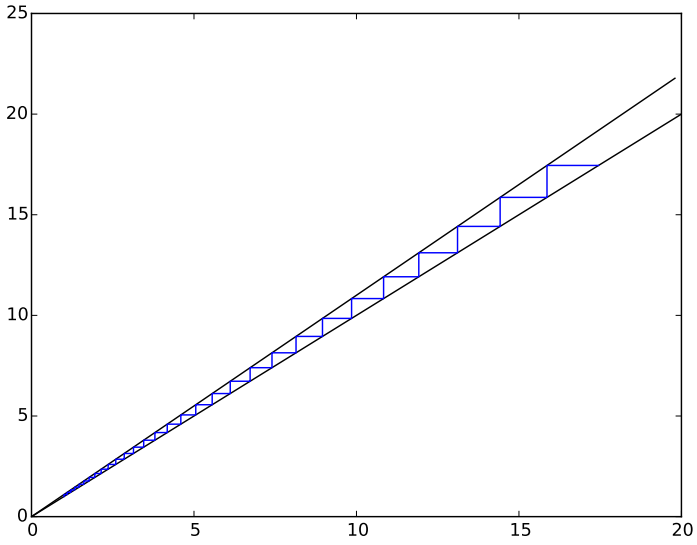


Figure 5.10: Visual output of Code 5.4. This is a cobweb plot of the exponential growth model simulated in Code 4.9.

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):
```

[1] If you are using Anaconda, NetworkX is already installed. If you are using Enthought Canopy, you can easily install it using its Package Manager.

```
        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 `connected_components` function, and then arranged them in a grid of plots using pylab's `subplot` feature. The result is shown in Fig. 5.11. 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 5.8 Draw a phase space of the following difference equation within the range $0 \leq x < 100$ by modifying Code 5.5:

$$x_t = x_{t-1}^{x_{t-1}} \mod 100 \tag{5.19}$$

5.5 Variable Rescaling

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

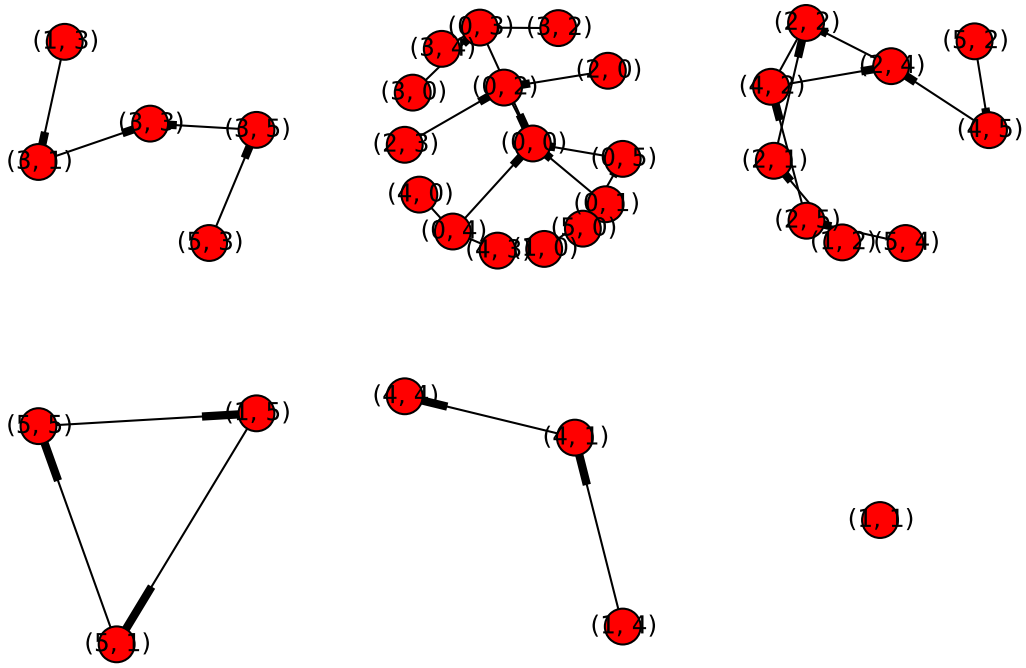


Figure 5.11: Graph-based phase space of Eq. (5.18) drawn with Code 5.5.

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 shouldn't 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:

$$x_t = x_{t-1} + rx_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{5.20}$$

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:

$$x \to \alpha x' \tag{5.21}$$

With this replacement, the model equation becomes

$$\alpha x'_t = \alpha x'_{t-1} + r\alpha x'_{t-1}\left(1 - \frac{\alpha x'_{t-1}}{K}\right). \tag{5.22}$$

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, the equation can be further simplified, for example, like

$$x'_t = x'_{t-1} + rx'_{t-1}\left(1 - \frac{\alpha x'_{t-1}}{K}\right) \tag{5.23}$$

$$= x'_{t-1}\left(1 + r\left(1 - \frac{\alpha x'_{t-1}}{K}\right)\right) \tag{5.24}$$

$$= x'_{t-1}\left(1 + r - \frac{r\alpha x'_{t-1}}{K}\right) \tag{5.25}$$

$$= (1+r)x'_{t-1}\left(1 - \frac{r\alpha x'_{t-1}}{K(1+r)}\right). \tag{5.26}$$

Here, the most convenient choice of α would be $\alpha = K(1+r)/r$, with which the equation above becomes

$$x'_t = (1+r)x'_{t-1}(1 - x'_{t-1}). \tag{5.27}$$

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:

$$x'_t = r'x'_{t-1}(1 - x'_{t-1}) \tag{5.28}$$

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, $\alpha = K(1+r)/r$ 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,

$$x_t = rx_{t-1}(1 - x_{t-1}), \tag{5.29}$$

has a designated name; it is called the *logistic map*. It 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:

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 5.9 Simplify the following difference equation by variable rescaling:

$$x_t = \frac{a}{x_{t-1} + b} \tag{5.30}$$

Exercise 5.10 Simplify the following two-dimensional predator-prey difference equation model by variable rescaling:

$$x_t = x_{t-1} + r x_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) - \left(1 - \frac{1}{b y_{t-1} + 1}\right) x_{t-1} \tag{5.31}$$

$$y_t = y_{t-1} - d y_{t-1} + c x_{t-1} y_{t-1} \tag{5.32}$$

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

$$x_t = Ax_{t-1}, \tag{5.33}$$

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

$$x_t = Ax_{t-1} + a, \tag{5.34}$$

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

$$y_t = \left(\begin{array}{c} x_t \\ \hline 1 \end{array}\right) = \left(\begin{array}{c|c} A & a \\ \hline 0 & 1 \end{array}\right) \left(\begin{array}{c} x_{t-1} \\ \hline 1 \end{array}\right) = By_{t-1}. \tag{5.35}$$

Therefore, we can be assured that the constant-free form of Eq. (5.33) covers all possible behaviors of linear difference equations.

Obviously, Eq. (5.33) has the following closed-form solution:

$$x_t = A^t x_0 \tag{5.36}$$

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. (5.36) 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

$$Av_i = \lambda_i v_i. \tag{5.37}$$

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

$$A^t v_i = A^{t-1}\lambda_i v_i = A^{t-2}\lambda_i^2 v_i = \ldots = \lambda_i^t v_i. \tag{5.38}$$

This looks promising. Now, we just need to apply the above simplification to Eq. (5.36). To do so, we need to represent the initial state x_0 by using A's eigenvectors as the basis vectors, i.e.,

$$x_0 = b_1 v_1 + b_2 v_2 + \ldots + b_n v_n, \tag{5.39}$$

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 Eq. (5.36), we get the following:

$$x_t = A^t(b_1v_1 + b_2v_2 + \ldots + b_nv_n) \tag{5.40}$$
$$= b_1A^tv_1 + b_2A^tv_2 + \ldots + b_nA^tv_n \tag{5.41}$$
$$= b_1\lambda_1^tv_1 + b_2\lambda_2^tv_2 + \ldots + b_n\lambda_n^tv_n \tag{5.42}$$

This result clearly shows that the asymptotic behavior of x_t 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

$$x_t = \lambda_1^t \left(b_1v_1 + b_2 \left(\frac{\lambda_2}{\lambda_1}\right)^t v_2 + \ldots + b_n \left(\frac{\lambda_n}{\lambda_1}\right)^t v_n \right), \tag{5.43}$$

$$\lim_{t\to\infty} x_t \approx \lambda_1^t b_1 v_1. \tag{5.44}$$

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:

$$x_t = x_{t-1} + x_{t-2} \tag{5.45}$$

[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 with such 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].

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

$$x_t = x_{t-1} + y_{t-1} \tag{5.46}$$

$$y_t = x_{t-1} \tag{5.47}$$

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

$$x_t = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} x_{t-1}. \tag{5.48}$$

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:

$$\det(A - \lambda I) = 0 \tag{5.49}$$

Here, $\det(X)$ is the *determinant* of matrix X. For this Fibonacci sequence example, this equation is

$$\det \begin{pmatrix} 1-\lambda & 1 \\ 1 & -\lambda \end{pmatrix} = -(1-\lambda)\lambda - 1 = \lambda^2 - \lambda - 1 = 0, \tag{5.50}$$

which gives

$$\lambda = \frac{1 \pm \sqrt{5}}{2} \tag{5.51}$$

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:

$$x_t \approx 1.61803399\, x_{t-1} \tag{5.52}$$

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:

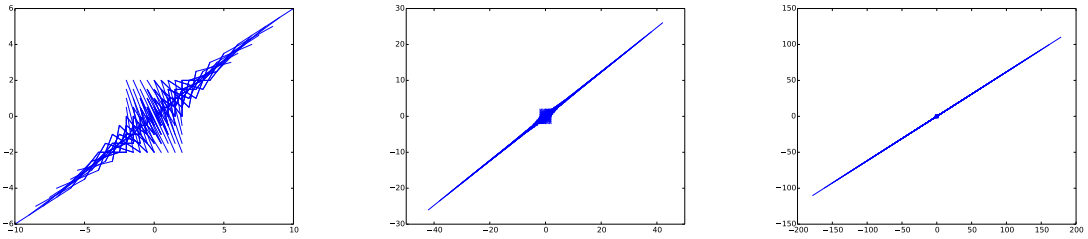
- $|\lambda_d| > 1$: The system is *unstable*, diverging to infinity.
- $|\lambda_d| < 1$: The system is *stable*, converging to the origin.
- $|\lambda_d| = 1$: The system is *stable*, but the dominant eigenvector component is conserved, and therefore the system may converge to a non-zero equilibrium 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 5.11 Visualize the phase space of Eq. (5.48).

The results are shown in Fig. 5.12, 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.



$t = 0 - 3$ $t = 0 - 6$ $t = 0 - 9$

Figure 5.12: Phase space visualizations of Eq. (5.48) for three different simulation lengths.

Figure 5.13 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:

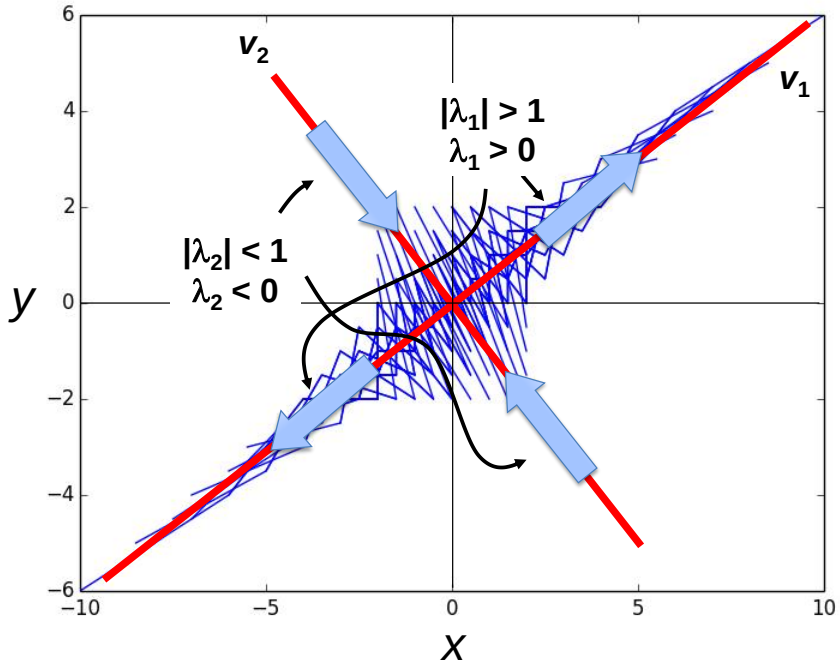


Figure 5.13: Relationships among eigenvalues, eigenvectors, and the phase space of a discrete-time dynamical system.

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:

```
[0.9999999999999989, 0.9999999999999989]
```

This means that $|\lambda|$ is essentially 1, indicating that the system shows sustained oscillation, as seen in Fig. 4.3.

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

$$x_t = x_{t-1} - y_{t-1} \tag{5.53}$$
$$y_t = -x_{t-1} - 3y_{t-1} + z_{t-1} \tag{5.54}$$
$$z_t = y_{t-1} + z_{t-1} \tag{5.55}$$

Exercise 5.13 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 5.14 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?

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

$$x_t = F(x_{t-1}) \tag{5.56}$$

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

$$x_{\text{eq}} = F(x_{\text{eq}}). \tag{5.57}$$

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, $\Delta x_t = x_t - x_{\text{eq}}$. Specifically, we apply the following replacement

$$x_t \Rightarrow x_{\text{eq}} + \Delta x_t \tag{5.58}$$

to Eq. (5.56), to obtain

$$x_{\text{eq}} + \Delta x_t = F(x_{\text{eq}} + \Delta x_{t-1}). \tag{5.59}$$

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:

$$F(x_{\text{eq}} + \Delta x_{t-1}) = F(x_{\text{eq}}) + F'(x_{\text{eq}})\Delta x_{t-1} + \frac{F''(x_{\text{eq}})}{2!}\Delta x_{t-1}^2 + \frac{F'''(x_{\text{eq}})}{3!}\Delta x_{t-1}^3 + \dots \tag{5.60}$$

$$\approx F(x_{\text{eq}}) + F'(x_{\text{eq}})\Delta x_{t-1} \tag{5.61}$$

This means that, for a scalar function F, $F(x_{\text{eq}} + \Delta x)$ 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 Eq. (5.57), Eq. (5.59) becomes the following very simple linear difference equation:

$$\Delta x_t \approx F'(x_{\text{eq}})\Delta x_{t-1} \tag{5.62}$$

This means that, if $|F'(x_{\text{eq}})| > 1$, Δx grows exponentially, and thus the equilibrium point x_{eq} is unstable. Or if $|F'(x_{\text{eq}})| < 1$, Δx shrinks exponentially, and thus x_{eq} is stable. Interestingly, this conclusion has some connection to the cobweb plot we discussed before. $|F'(x_{\text{eq}})|$ 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:

$$x_{1,t} = F_1(x_{1,t-1}, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.63}$$
$$x_{2,t} = F_2(x_{1,t-1}, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.64}$$
$$\vdots$$
$$x_{n,t} = F_n(x_{1,t-1}, x_{2,t-1}, \ldots, x_{n,t-1}) \tag{5.65}$$

Using variable replacement similar to Eq. (5.58), these equations are rewritten as follows:

$$x_{1,\mathrm{eq}} + \Delta x_{1,t} = F_1(x_{1,\mathrm{eq}} + \Delta x_{1,t-1}, x_{2,\mathrm{eq}} + \Delta x_{2,t-1} \ldots, x_{n,\mathrm{eq}} + \Delta x_{n,t-1}) \tag{5.66}$$
$$x_{2,\mathrm{eq}} + \Delta x_{2,t} = F_2(x_{1,\mathrm{eq}} + \Delta x_{1,t-1}, x_{2,\mathrm{eq}} + \Delta x_{2,t-1} \ldots, x_{n,\mathrm{eq}} + \Delta x_{n,t-1}) \tag{5.67}$$
$$\vdots$$
$$x_{n,\mathrm{eq}} + \Delta x_{n,t} = F_n(x_{1,\mathrm{eq}} + \Delta x_{1,t-1}, x_{2,\mathrm{eq}} + \Delta x_{2,t-1} \ldots, x_{n,\mathrm{eq}} + \Delta x_{n,t-1}) \tag{5.68}$$

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 in to an infinitesimally small area near the equilibrium point, each F_i looks like a completely flat "plane" in a multidimensional space (Fig. 5.14) 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. (5.61), as

$$\begin{aligned} &F_i(x_{1,\mathrm{eq}} + \Delta x_{1,t-1}, x_{2,\mathrm{eq}} + \Delta x_{2,t-1} \ldots, x_{n,\mathrm{eq}} + \Delta x_{n,t-1}) \\ &\quad \approx F_i(x_{\mathrm{eq}}) + \left.\frac{\partial F_i}{\partial x_1}\right|_{x_{\mathrm{eq}}} \Delta x_{1,t-1} + \left.\frac{\partial F_i}{\partial x_2}\right|_{x_{\mathrm{eq}}} \Delta x_{2,t-1} + \ldots + \left.\frac{\partial F_i}{\partial x_n}\right|_{x_{\mathrm{eq}}} \Delta x_{n,t-1}. \end{aligned} \tag{5.69}$$

This linear approximation allows us to rewrite Eqs. (5.66)–(5.68) into the following, very concise linear equation:

$$x_{\mathrm{eq}} + \Delta x_t \approx F(x_{\mathrm{eq}}) + \left.\begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix}\right|_{x=x_{\mathrm{eq}}} \Delta x_{t-1} \tag{5.70}$$

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

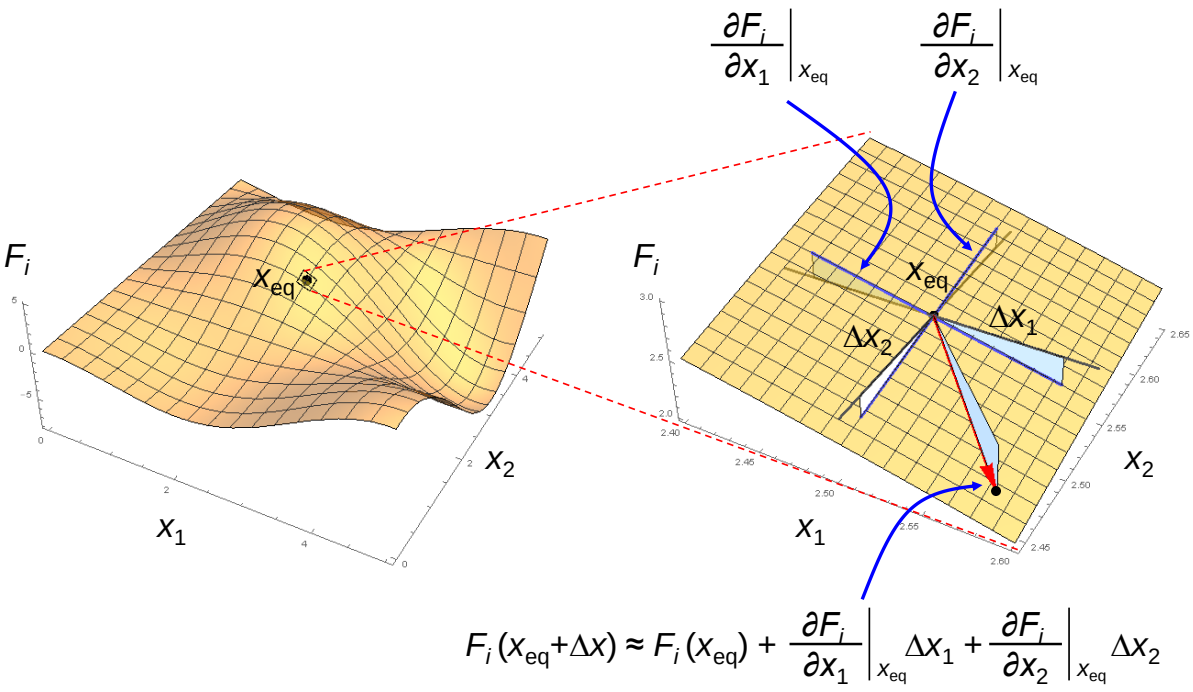


Figure 5.14: Linearization of a multidimensional nonlinear function around an equilibrium point.

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

By combining the result above with Eq. (5.57), we obtain

$$\Delta x_t \approx J \Delta x_{t-1}, \tag{5.71}$$

where J is the Jacobian matrix of F at $x = x_{\rm eq}$. 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 $x_{\rm eq}$ as the new origin.

Now we can calculate the eigenvalues of J to see if this system is stable or not, around $x_{\rm eq}$. If the absolute value of the dominant eigenvalue λ_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 $|\lambda_d| > 1$, 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 $|\lambda_d| = 1$, 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 5.15 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 $\Rightarrow$ The equilibrium point is unstable.

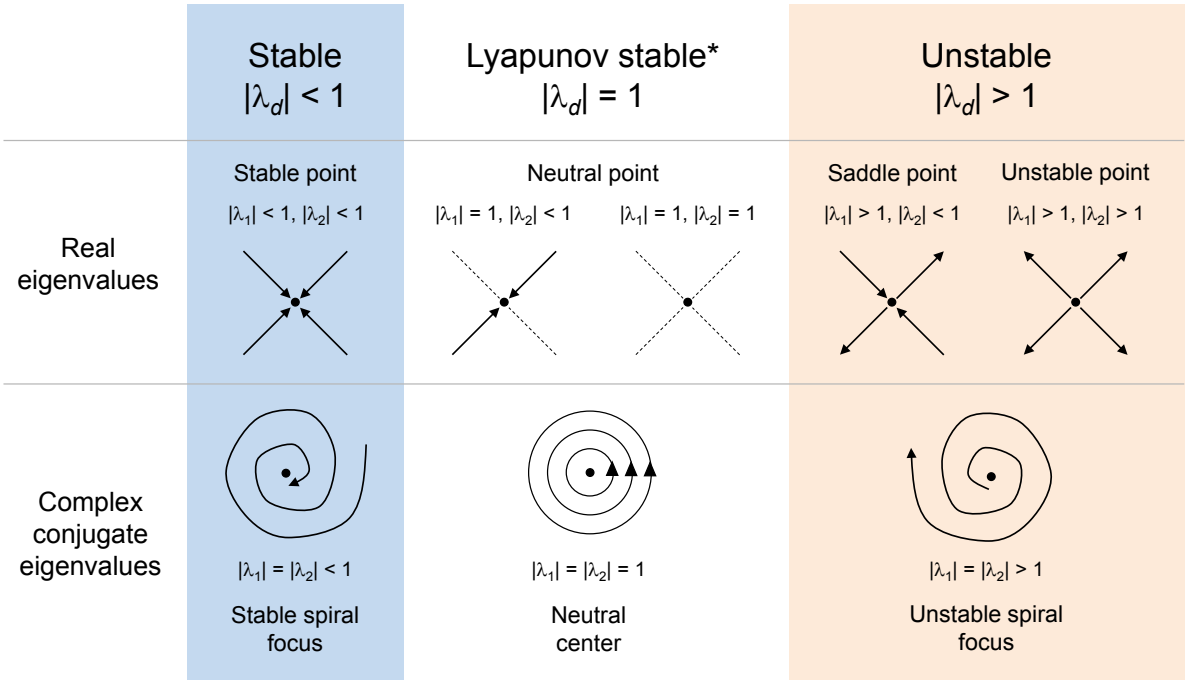


Figure 5.15: Schematic illustrations of several different types of equilibrium points and their nearby trajectories in 2-D discrete-time dynamical systems. The absolute value of the dominant eigenvalue $|\lambda_d|$ determines the overall stability of the equilibrium point (top), although it would generally 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 $\Rightarrow$ The equilibrium point is stable.
- Equal to 1 $\Rightarrow$ 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).

Exercise 5.15 Consider the following iterative map ($a > 0$, $b > 0$):

$$x_t = x_{t-1} + a\sin(bx_{t-1}) \tag{5.72}$$

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

Exercise 5.16 Consider the following two-dimensional difference equation model:

$$x_t = x_{t-1} + 2x_{t-1}(1 - x_{t-1}) - x_{t-1}y_{t-1} \tag{5.73}$$
$$y_t = y_{t-1} + 2y_{t-1}(1 - y_{t-1}) - x_{t-1}y_{t-1} \tag{5.74}$$

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 5.17 Consider the following two-dimensional difference equation model:

$$x_t = x_{t-1}y_{t-1} \tag{5.75}$$
$$y_t = y_{t-1}(x_{t-1} - 1) \tag{5.76}$$

1. Find all equilibrium points (which you may have done already in Exercise 5.2).
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.

Chapter 6

Continuous-Time Models I: Modeling

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:

$$\frac{dx}{dt} = F(x, t) \tag{6.1}$$

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

$$\frac{dx}{dt} = \lim_{\delta t \to 0} \frac{x(t + \delta t) - x(t)}{\delta t}. \tag{6.2}$$

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.

6.2 Classifications of Model Equations

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:

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

Here is an example:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta \tag{6.3}$$

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:

$$\omega = \frac{d\theta}{dt} \tag{6.4}$$

Using this, the left hand side of Eq. (6.3) can be written as $d\omega/dt$, and therefore, the equation can be turned into the following first-order form:

$$\frac{d\theta}{dt} = \omega \tag{6.5}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta \tag{6.6}$$

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. This time it is a non-autonomous equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta + k\sin(2\pi f t + \phi) \tag{6.7}$$

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:

$$\frac{d\theta}{dt} = \omega \tag{6.8}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta + k\sin(2\pi f t + \phi) \tag{6.9}$$

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:

$$\frac{d\tau}{dt} = 1, \quad \tau(0) = 0 \tag{6.10}$$

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

$$\frac{d\theta}{dt} = \omega \tag{6.11}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta + k\sin(2\pi f \tau + \phi) \tag{6.12}$$

$$\frac{d\tau}{dt} = 1, \quad \tau(0) = 0 \tag{6.13}$$

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 6.1 Convert the following differential equation into first-order form.

$$\frac{d^2x}{dt^2} - x\frac{dx}{dt} + x^2 = 0 \tag{6.14}$$

Exercise 6.2 Convert the following differential equation into an autonomous, first-order form.

$$\frac{d^2x}{dt^2} - a\cos(bt) = 0 \tag{6.15}$$

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.

6.3 Connecting Continuous-Time Models with Discrete-Time 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

$$x_t = F(x_{t-1}), \tag{6.16}$$

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

$$\frac{dx}{dt} = G(x), \tag{6.17}$$

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:

$$\frac{dx}{dt} \approx \frac{\Delta x}{\Delta t} \tag{6.18}$$

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 Eq. (6.17) as

$$\frac{\Delta x}{\Delta t} = \frac{x(t + \Delta t) - x(t)}{\Delta t} \approx G(x(t)), \tag{6.19}$$

$$x(t + \Delta t) \approx x(t) + G(x(t))\Delta t. \tag{6.20}$$

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

$$F(x) \Leftrightarrow x + G(x)\Delta t \tag{6.21}$$

Or, equivalently:

$$G(x) \Leftrightarrow \frac{F(x) - x}{\Delta t} \tag{6.22}$$

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

$$Ax \Leftrightarrow x + Bx\Delta t, \quad \text{i.e.,} \tag{6.23}$$

$$A \Leftrightarrow I + B\Delta t, \quad \text{or} \tag{6.24}$$

$$B \Leftrightarrow \frac{A - I}{\Delta t}. \tag{6.25}$$

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, Eq. (6.20) 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 straightforward, 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 Eq. (6.25) 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 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 Eq. (6.25).

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

$dx/dt = G(x)$:

$$x(t + \Delta t) = x(t) + G(x(t))\Delta t \tag{6.26}$$

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 limit $\Delta t \to 0$, although in practice, this method is less accurate for finite-sized Δt than other more sophisticated methods (see Exercise 6.6). 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 $0 \leq t < 50$ in Python, with $x(0) = 0.1$, $r = 0.2$, $K = 1$ and $\Delta t = 0.01$:

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right) \tag{6.27}$$

The first (and only) thing we need to do is to discretize time in the model above. Using Eq. (6.26), the equation becomes

$$x(t + \Delta t) = x(t) + rx(t)\left(1 - \frac{x(t)}{K}\right)\Delta t, \tag{6.28}$$

which is nothing more than a typical difference equation. So, we can easily revise Code 4.10 to create a simulator of Eq. (6.27):

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()
```

Note that there really isn't much difference between this and what we did previously. This code will produce a nice, smooth curve as a result of the numerical integration, shown in Fig. 6.1. If you choose even smaller values for Δt, the curve will get closer to the true solution of Eq. (6.27).

> *Exercise 6.4* 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

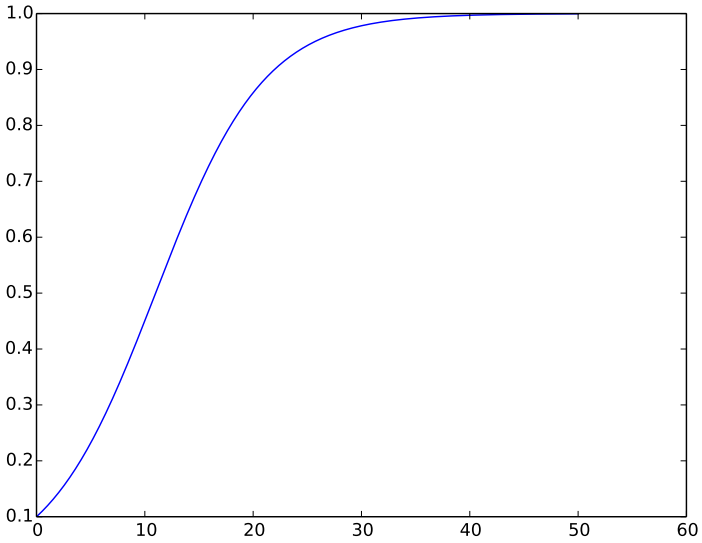


Figure 6.1: Visual output of Code 6.1.

models. I hope the following exercises will help you further investigate those subtleties of the numerical integration of differential equations.

Exercise 6.5 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.

$$\frac{dx}{dt} = ax - bxy \tag{6.29}$$

$$\frac{dy}{dt} = -cy + dxy \tag{6.30}$$

Then try to reduce the value of Δt to even smaller values and see how the simulation results are affected by such changes. Discuss what the overall results imply about the Euler forward method.

Exercise 6.6 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.

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

Exercise 6.8 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:

$$S + E \rightarrow 2E \tag{6.31}$$

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

Exercise 6.9 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's life, and simulate its behavior.

Chapter 7

Continuous-Time Models II: Analysis

7.1 Finding Equilibrium Points

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

$$0 = G(x_{\rm eq}) \tag{7.1}$$

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

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right) \tag{7.2}$$

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

$$0 = rx_{\rm eq}\left(1 - \frac{x_{\rm eq}}{K}\right) \tag{7.3}$$

$$x_{\rm eq} = 0, \quad K \tag{7.4}$$

It turns out that the result is the same as that of its discrete-time counterpart (see Eq. (5.6)).

Exercise 7.1 Find the equilibrium points of the following model:

$$\frac{dx}{dt} = x^2 - rx + 1 \tag{7.5}$$

Exercise 7.2 Find the equilibrium points of the following model of a simple pendulum:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta \tag{7.6}$$

Exercise 7.3 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.

$$\frac{dS}{dt} = -aSI \tag{7.7}$$
$$\frac{dI}{dt} = aSI - bI \tag{7.8}$$
$$\frac{dR}{dt} = bI \tag{7.9}$$

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:

$$\frac{dx}{dt} = x - xy \tag{7.10}$$

$$\frac{dy}{dt} = -y + xy \tag{7.11}$$

The result is shown in Fig. 7.1. 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 7.4 Draw a phase space of the following differential equation (motion of a simple pendulum) in Python:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta \tag{7.12}$$

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

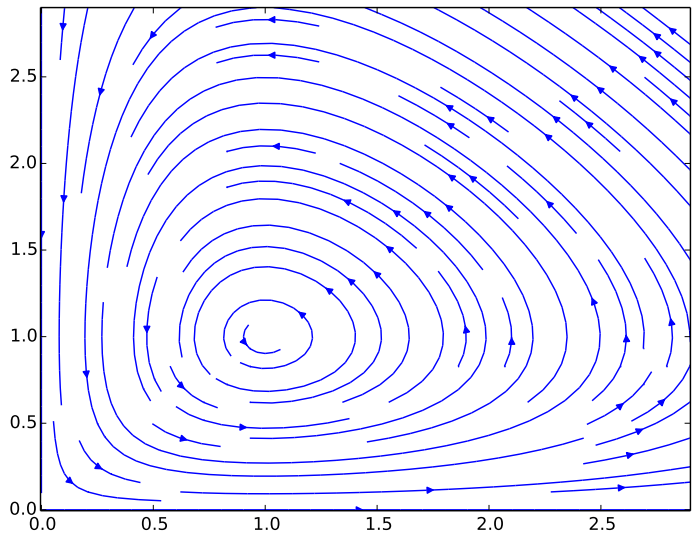


Figure 7.1: Phase space drawn with Code 7.1.

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*:

$$\frac{dx}{dt} = ax - bxy \tag{7.13}$$

$$\frac{dy}{dt} = -cy + dxy \tag{7.14}$$

$$x \geq 0, \quad y \geq 0, \quad a > 0, \quad b > 0, \quad c > 0, \quad d > 0 \tag{7.15}$$

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

$$0 = ax - bxy \tag{7.16}$$

$$0 = -cy + dxy \tag{7.17}$$

The first equation gives

$$x = 0, \quad \text{or} \quad y = \frac{a}{b}. \tag{7.18}$$

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

$$y = 0, \quad \text{or} \quad x = \frac{c}{d}. \tag{7.19}$$

Again, these two lines constitute another set of nullclines for $dy/dt = 0$. These results can be visualized manually as shown in Fig. 7.2. Equilibrium points exist where the two sets of nullclines intersect.

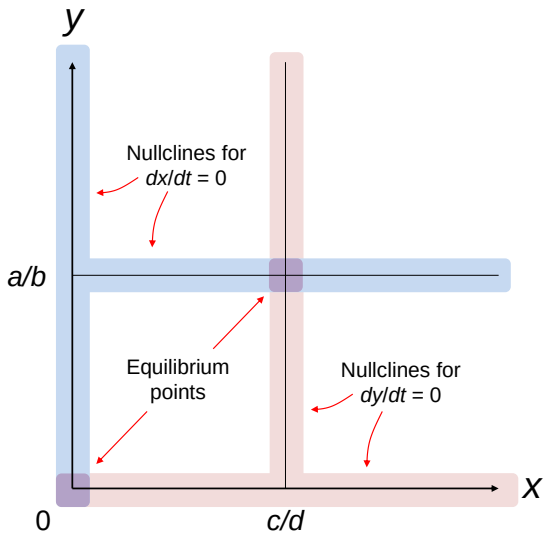


Figure 7.2: Drawing a phase space (1): Adding nullclines.

Everywhere on the first set of nullclines, 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, 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. 7.3).

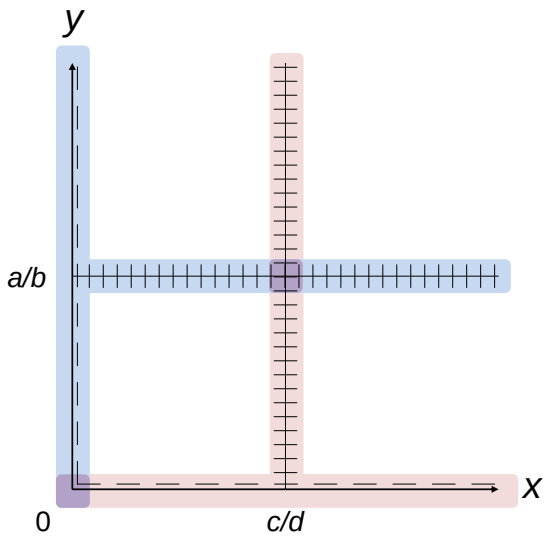


Figure 7.3: Drawing a phase space (2): Adding directions to the nullclines.

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:

- $\frac{dx}{dt} > 0$, $\frac{dy}{dt} > 0$ (to "Northeast")
- $\frac{dx}{dt} < 0$, $\frac{dy}{dt} > 0$ (to "Northwest")
- $\frac{dx}{dt} > 0$, $\frac{dy}{dt} < 0$ (to "Southeast")
- $\frac{dx}{dt} < 0$, $\frac{dy}{dt} < 0$ (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

$$\left.\frac{dx}{dt}\right|_{(x,y)=\left(\frac{2c}{d},\frac{2a}{b}\right)} = a\frac{2c}{d} - b\frac{2c}{d}\frac{2a}{b} = -\frac{2ac}{d} < 0, \tag{7.20}$$

$$\left.\frac{dy}{dt}\right|_{(x,y)=\left(\frac{2c}{d},\frac{2a}{b}\right)} = -c\frac{2a}{b} + d\frac{2c}{d}\frac{2a}{b} = \frac{2ac}{b} > 0. \tag{7.21}$$

Therefore, you can tell that the trajectories are flowing to "Northwest" in that region. If you repeat the same testing for the three other regions, you obtain an outline of the phase space of the model shown in Fig. 7.4, which shows a cyclic behavior caused by the interaction between prey (x) and predator (y) populations.

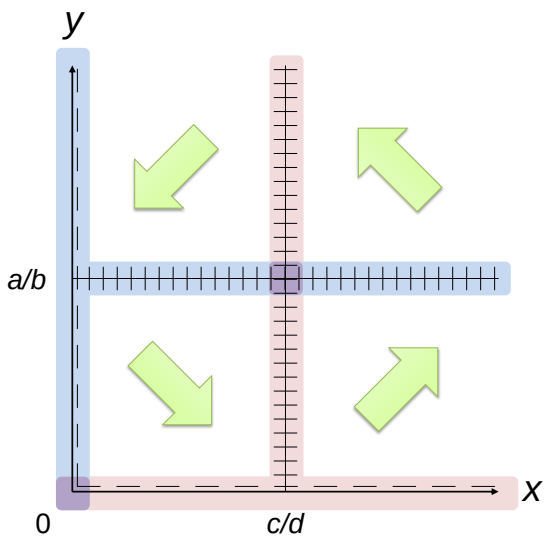


Figure 7.4: Drawing a phase space (3): Adding directions of trajectories in each region.

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 7.5 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.

$$\frac{dS}{dt} = -aSI \tag{7.22}$$

$$\frac{dI}{dt} = aSI - bI \tag{7.23}$$

$$S \geq 0, \quad I \geq 0, \quad a > 0, \quad b > 0 \tag{7.24}$$

Exercise 7.6 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.

$$\frac{d^2x}{dt^2} - x\frac{dx}{dt} + x^2 = 0 \tag{7.25}$$

7.3 Variable Rescaling

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

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

$$x_t = x_{t-1} + rx_{t-1}\left(1 - \frac{x_{t-1}}{K}\right) \tag{7.26}$$

was simplified to the following form:

$$x'_t = r' x'_{t-1}(1 - x'_{t-1}) \tag{7.27}$$

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:

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right) \tag{7.28}$$

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

$$x \to \alpha x' \tag{7.29}$$

$$t \to \beta t' \tag{7.30}$$

With these replacements, the model equation is simplified as

$$\frac{d(\alpha x')}{d(\beta t')} = r\alpha x'\left(1 - \frac{\alpha x'}{K}\right) \tag{7.31}$$

$$\frac{\beta}{\alpha} \cdot \frac{d(\alpha x')}{d(\beta t')} = \frac{\beta}{\alpha} \cdot r\alpha x'\left(1 - \frac{\alpha x'}{K}\right) \tag{7.32}$$

$$\frac{dx'}{dt'} = r\beta x'\left(1 - \frac{\alpha x'}{K}\right) \tag{7.33}$$

$$\frac{dx'}{dt'} = x'(1 - x') \tag{7.34}$$

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 7.7 Simplify the following differential equation by variable rescaling:

$$\frac{dx}{dt} = ax^2 + bx + c \tag{7.35}$$

Exercise 7.8 Simplify the following differential equation by variable rescaling:

$$\frac{dx}{dt} = \frac{a}{x + b} \tag{7.36}$$

$$a > 0, \quad b > 0 \tag{7.37}$$

Exercise 7.9 Simplify the following two-dimensional differential equation model by variable rescaling:

$$\frac{dx}{dt} = ax(1-x) - bxy \tag{7.38}$$

$$\frac{dy}{dt} = cy(1-y) - dxy \tag{7.39}$$

7.4 Asymptotic Behavior of Continuous-Time Linear Dynamical Systems

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

$$\frac{dx}{dt} = Ax, \tag{7.40}$$

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:

$$y = \left(\begin{array}{c} x \\ \hline 1 \end{array}\right) \tag{7.41}$$

$$\frac{dy}{dt} = \left(\begin{array}{c|c} A & a \\ \hline 0 & 0 \end{array}\right) \left(\begin{array}{c} x \\ \hline 1 \end{array}\right) = By \tag{7.42}$$

Note that the last-row, last-column element of the expanded coefficient matrix is now 0, not 1, because of Eq. (6.25). This result guarantees that the constant-free form given in Eq. (7.40) is general enough to represent various dynamics of linear dynamical systems.

Now, what is the asymptotic behavior of Eq. (7.40)? 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:

$$x(t) = e^{At}x(0) \tag{7.43}$$

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

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}, \tag{7.44}$$

with $X^0 = I$. 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 7.10 Confirm that the solution Eq. (7.43) satisfies Eq. (7.40).

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

$$Xv = \lambda v \quad \Rightarrow \quad e^X v = e^\lambda v \tag{7.45}$$

Exercise 7.11 Confirm Eq. (7.45) using Eq. (7.44).

We can use these properties to study the asymptotic behavior of Eq. (7.43). 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

$$x(0) = b_1 v_1 + b_2 v_2 + \ldots + b_n v_n, \tag{7.46}$$

where n is the dimension of the state space and v_i are the eigenvectors of A (and of e^A). Applying this to Eq. (7.43) results in

$$\begin{aligned} x(t) &= e^{At}(b_1 v_1 + b_2 v_2 + \ldots + b_n v_n) && (7.47)\\ &= b_1 e^{At} v_1 + b_2 e^{At} v_2 + \ldots + b_n e^{At} v_n && (7.48)\\ &= b_1 e^{\lambda_1 t} v_1 + b_2 e^{\lambda_2 t} v_2 + \ldots + b_n e^{\lambda_n t} v_n. && (7.49) \end{aligned}$$

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 $|e^{\lambda_i}| = e^{\mathrm{Re}(\lambda_i)}$, 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 ($\mathrm{Re}(\lambda_1) > \mathrm{Re}(\lambda_2), \mathrm{Re}(\lambda_3), \ldots \mathrm{Re}(\lambda_n)$), then

$$\begin{aligned} x(t) &= e^{\lambda_1 t}\left(b_1 v_1 + b_2 e^{(\lambda_2 - \lambda_1)t} v_2 + \ldots + b_n e^{(\lambda_n - \lambda_1)t} v_n\right), && (7.50)\\ \lim_{t\to\infty} x(t) &\approx e^{\lambda_1 t} b_1 v_1. && (7.51) \end{aligned}$$

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:

- $\mathrm{Re}(\lambda) > 0$ means that the component is growing.
- $\mathrm{Re}(\lambda) < 0$ means that the component is shrinking.
- $\mathrm{Re}(\lambda) = 0$ means that the component is conserved.

For continuous-time models, the *real part of the dominant eigenvalue* λ_d determines the stability of the whole system as follows:

- $\mathrm{Re}(\lambda_d) > 0$: The system is *unstable*, diverging to infinity.
- $\mathrm{Re}(\lambda_d) < 0$: The system is *stable*, converging to the origin.
- $\mathrm{Re}(\lambda_d) = 0$: 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]):

$$\frac{dx}{dt} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} x = Ax \tag{7.52}$$

The eigenvalues of the coefficient matrix can be obtained by solving the following equation for λ:

$$\det \begin{pmatrix} a-\lambda & b \\ c & d-\lambda \end{pmatrix} = 0 \tag{7.53}$$

Or:

$$(a-\lambda)(d-\lambda)-bc = \lambda^2-(a+d)\lambda+ad-bc \tag{7.54}$$
$$= \lambda^2-\mathrm{Tr}(A)\lambda+\det(A)=0 \tag{7.55}$$

Here, $\mathrm{Tr}(A)$ is the *trace* of matrix A, i.e., the sum of its diagonal components. The solutions of the equation above are

$$\lambda = \frac{\mathrm{Tr}(A) \pm \sqrt{\mathrm{Tr}(A)^2-4\det(A)}}{2}. \tag{7.56}$$

Between those two eigenvalues, which one is dominant? 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:

$$\mathrm{Re}(\lambda_\mathrm{d}) = \begin{cases} \dfrac{\mathrm{Tr}(A)}{2} & \text{if } \mathrm{Tr}(A)^2 < 4\det(A) \\ \dfrac{\mathrm{Tr}(A)+\sqrt{\mathrm{Tr}(A)^2-4\det(A)}}{2} & \text{if } \mathrm{Tr}(A)^2 \geq 4\det(A) \end{cases} \tag{7.57}$$

If $\mathrm{Tr}(A)^2 < 4\det(A)$, the stability condition is simply

$$\mathrm{Tr}(A) < 0. \tag{7.58}$$

If $\mathrm{Tr}(A)^2 \geq 4\det(A)$, the stability condition is derived as follows:

$$\mathrm{Tr}(A)+\sqrt{\mathrm{Tr}(A)^2-4\det(A)} < 0 \tag{7.59}$$
$$\sqrt{\mathrm{Tr}(A)^2-4\det(A)} < -\mathrm{Tr}(A) \tag{7.60}$$

Since the radical on the left hand side must be non-negative, $\mathrm{Tr}(A)$ must be negative, at least. Also, by squaring both sides, we obtain

$$\mathrm{Tr}(A)^2-4\det(A) < \mathrm{Tr}(A)^2, \tag{7.61}$$
$$-4\det(A) < 0, \tag{7.62}$$
$$\det(A) > 0. \tag{7.63}$$

By combining all the results above, we can summarize how the two-dimensional linear dynamical system's stability depends on $\mathrm{Tr}(A)$ 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.

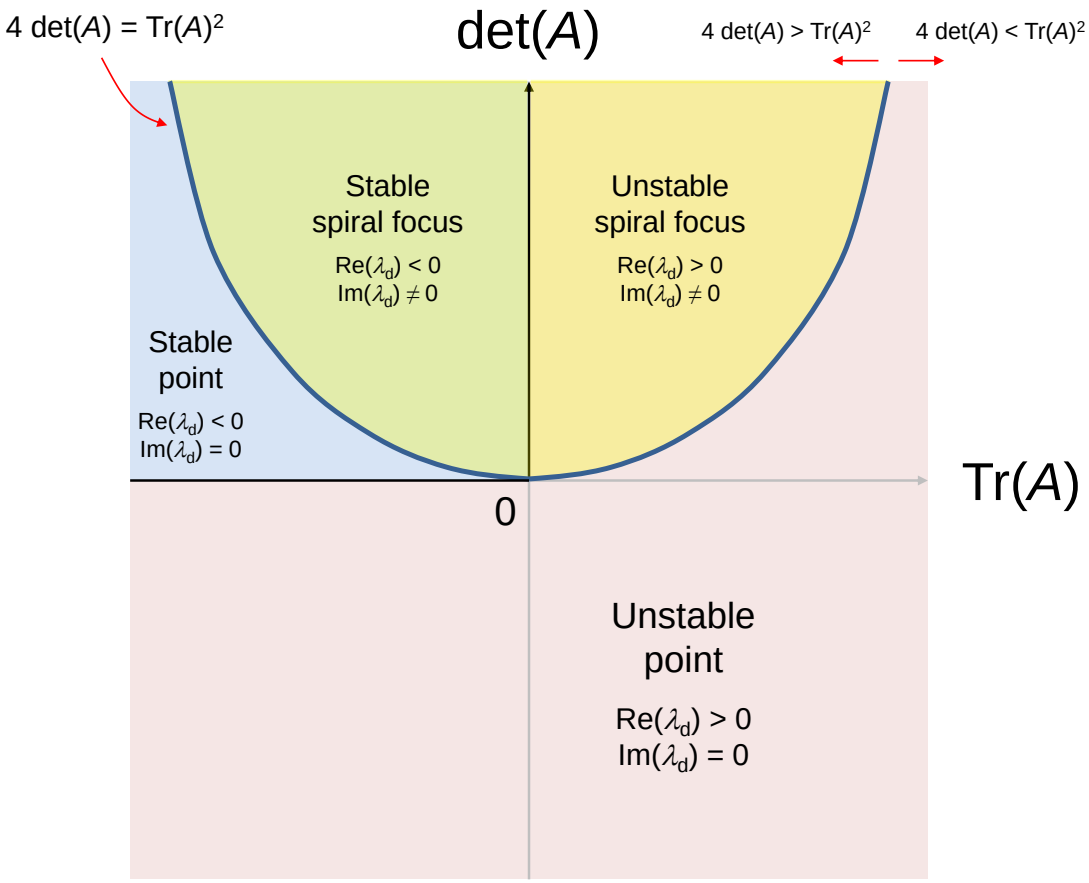


Figure 7.5: Overview of how the stability of a two-dimensional continuous-time linear dynamical system depends on $\mathrm{Tr}(A)$ and $\det(A)$.

Exercise 7.12 Show that the unstable points with $\det(A) < 0$ are saddle points.

Exercise 7.13 Determine the stability of the following linear systems:

- $\frac{dx}{dt} = \begin{pmatrix} -1 & 2 \\ 2 & -2 \end{pmatrix} x$
- $\frac{dx}{dt} = \begin{pmatrix} 0.5 & -1.5 \\ 1 & -1 \end{pmatrix} x$

Exercise 7.14 Confirm the analytical result shown in Fig. 7.5 by conducting numerical simulations in Python and by drawing phase spaces of the system for several samples of A.

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

$$\frac{dx}{dt} = F(x) \tag{7.64}$$

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

$$0 = F(x_{\rm eq}). \tag{7.65}$$

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

$$x(t) \Rightarrow x_{\rm eq} + \Delta x(t) \tag{7.66}$$

to Eq. (7.64), to obtain

$$\frac{d(x_{\rm eq} + \Delta x)}{dt} = \frac{d\Delta x}{dt} = F(x_{\rm eq} + \Delta x). \tag{7.67}$$

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

$$\frac{d\Delta x}{dt} \approx F(x_{\rm eq}) + J\Delta x, \tag{7.68}$$

where J is the Jacobian matrix of F at $x = x_{\rm eq}$ (if you forgot what the Jacobian matrix was, see Eq. (5.70)). By combining the result above with Eq. (7.65), we obtain

$$\frac{d\Delta x}{dt} \approx J\Delta x. \tag{7.69}$$

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 $x_{\rm eq}$. 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 than 0 $\Rightarrow$ The equilibrium point is stable.
   - Equal to 0 $\Rightarrow$ 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).

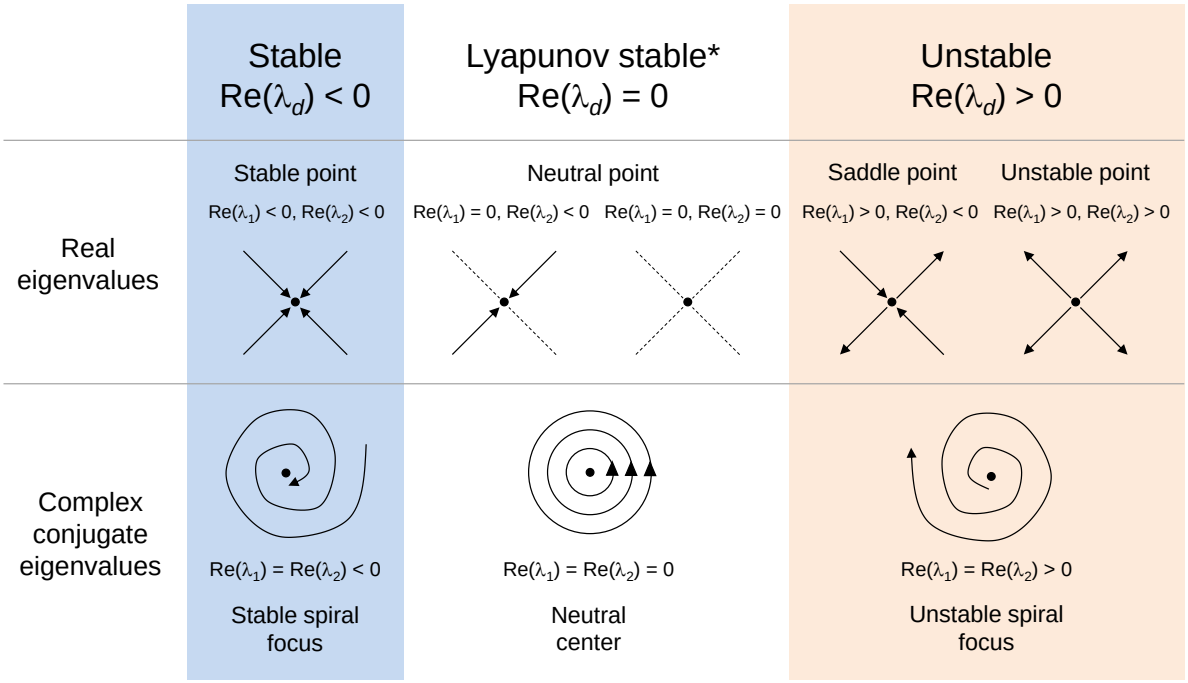


Figure 7.6: Schematic illustrations of several different types of equilibrium points and their nearby trajectories in 2-D continuous-time dynamical systems, shown in the same format as in Fig. 5.15. The real part of the dominant eigenvalue $\mathrm{Re}(\lambda_d)$ determines the overall stability of the equilibrium point (top), although it would generally require nonlinear analysis to show that the equilibrium point is truly neutral/Lyapunov stable (*).

Exercise 7.15 Consider the logistic growth model ($r > 0$, $K > 0$):

$$\frac{dx}{dt} = rx\left(1 - \frac{x}{K}\right) \tag{7.70}$$

Conduct a linear stability analysis to determine whether this model is stable or not at each of its equilibrium points $x_{\mathrm{eq}} = 0, K$.

Exercise 7.16 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):

$$\frac{dx}{dt} = -x + rxy - x^2 \tag{7.71}$$

$$\frac{dy}{dt} = y(1 - y) \tag{7.72}$$

$$x \geq 0, \quad y \geq 0, \quad r > 1 \tag{7.73}$$

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 7.17 Consider the differential equations of the *SIR model*:

$$\frac{dS}{dt} = -aSI \tag{7.74}$$

$$\frac{dI}{dt} = aSI - bI \tag{7.75}$$

$$\frac{dR}{dt} = bI \tag{7.76}$$

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 7.3).
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.