Nonlinearity

Nonlinearity

Wolfram Barfuss | University of Bonn | 2024/2025 **Complex Systems Modeling of Human-Environment Interactions**

Motivation

The issue of climate change

Read the following summary, adapted from the Intergovernmental Panel on Climate Change (IPCC) Third Assessment Report's Summary for Policymakers.

In 2001, the Intergovernmental Panel on Climate Change (IPCC), a scientific panel organized by the United Nations, concluded that carbon dioxide (CO2) and other greenhouse gas emissions were contributing to global warming. The panel stated that "most of the warming observed over the last 50 years is attributable to human activities."
The amount of CO2 in the atmosphere is affected by natural processes and human activity. Anthropogenic CO2 emissions (emissions resulting from human activity, including combustion of fossil fuels and changes in land use, especially deforestation) have been growing since the start of the Industrial Revolution (Fig. A). Natural processes gradually remove CO2 from the atmosphere (for example, as it is used by plant life and dissolves in the ocean). Currently, the net removal of atmospheric CO2 by natural processes is about half of the anthropogenic CO2 emissions. As a result, concentrations of CO2 in the atmosphere have increased from preindustrial levels of about 280 parts per million (ppm) to about 370 ppm today (Fig. B). Increases in the concentrations of greenhouse gases reduce the efficiency with which the Earth's surface radiates energy to space. This results in a positive radiative forcing that tends to warm the lower atmosphere and surface. As shown in Fig. C, global average surface temperatures have increased since the start of the Industrial Revolution.

Adapted from Sterman & Sweeney (2007) *Understanding public complacency about climate change: adults' mental models of climate change violate conservation of matter*

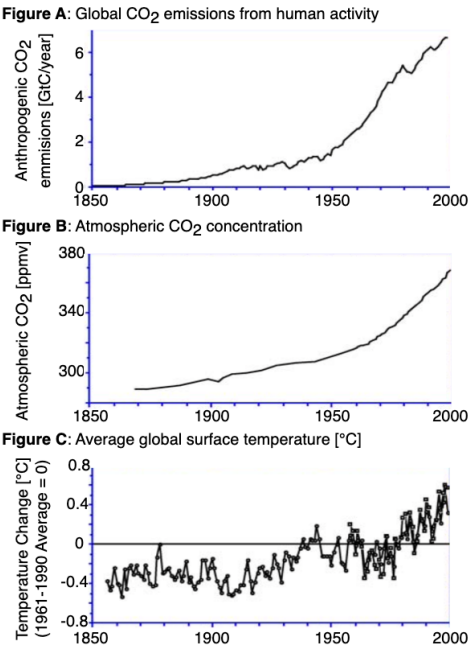


Figure 1: The issue of climate change

Now consider the following questions:

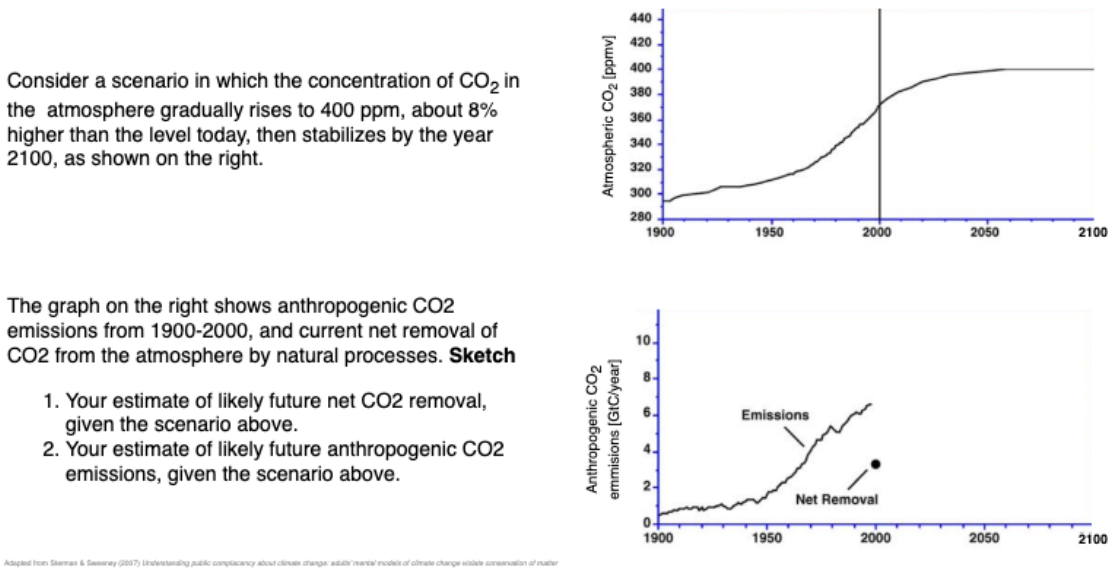


Figure 2: A scenario of climate change

Typical responses. This experiment was conducted with MIT graduate students, a group of highly educated adults (Sterman and Sweeney 2007). Yet, they showed a widespread misunderstanding of fundamental stock and flow relationships. Most subjects believed that atmospheric greenhouse gas (GHG) can be stabilized while emissions into the atmosphere continuously exceed the removal of GHGs from it.

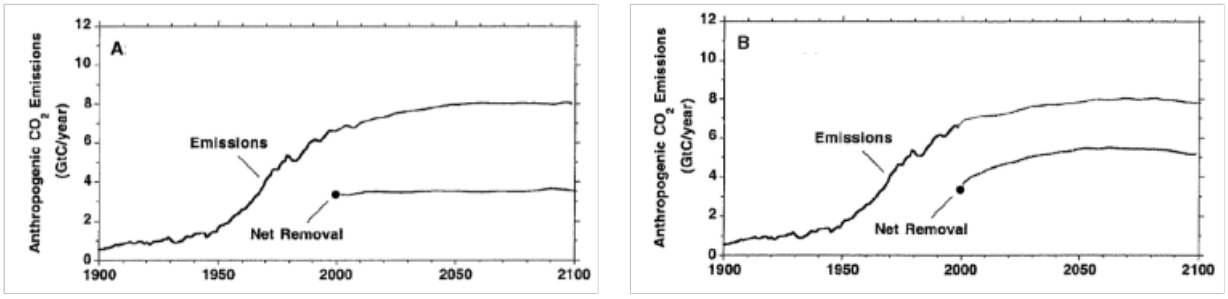


Figure 3: Typical Responses

These beliefs support wait-and-see policies and neglecting the issue climate change as a top priority for the policy agenda.

Carbon bathtub

Knowledge of climatology or calculus is not needed to respond correctly. Think of it like a bathtub: the water level represents the amount of greenhouse gases (GHGs) in the atmosphere. The water flowing into the tub is like the rate of GHG emissions, and the water flowing out is like the rate of GHG removal. If more water is flowing in than out, the water level rises. To keep the water level stable, the inflow and outflow need to be equal. This is similar to stabilizing GHG concentrations - emissions must equal removals. The balance between inflows and outflows determines how GHGs accumulate, not just the level of inflows.

Figure 4: Carbon Bathtub

Learning goals

After this chapter, students will be able to:

- Define and describe the **components of a dynamic system**.
- **Represent** dynamic system models in visual and mathematical form.
- Explain the concepts of **feedback loops** and **delays**.
- Explain two kinds of **non-linearity** and how they are related.
- **Implement** dynamic system models and **visualize model outputs** using Python, to interpret model results.
- **Analyze** the **stability** of equilibrium points in dynamic systems using linear stability analysis.

After motivating this chapter, we make ourselves ready for some computations by importing some Python libraries and setting up the plotting style.

```
import numpy as np
import scipy
import matplotlib.pyplot as plt
from ipywidgets import interact

import matplotlib.style as style; style.use('seaborn-v0_8')
plt.rcParams['figure.figsize'] = (7.8, 2.5); plt.rcParams['figure.dpi'] = 300
color = plt.rcParams['axes.prop_cycle'].by_key()['color'][0]  # get the first color
 ↪  of the default color cycle
plt.rcParams['axes.facecolor'] = 'white'; plt.rcParams['grid.color'] = 'gray';
 ↪  plt.rcParams['grid.linewidth'] = 0.25;
```

Dynamic systems

> A *dynamic system*[1] is a system whose state is uniquely specified by a set of variables and whose behavior is described by predefined rules.

You can think of the **state** of the system as a collection of **stocks** (also known as *state variables*), and the **rules** as the **flows** that change the stocks over time. At the system boundaries, you can imagine **sources** and **sinks**, which represent in- and out-flows to the system we are currently looking at.

For example, in the case of the carbon bathtub, the state of the system is the amount of carbon in the atmosphere. The rules are that emissions increase and net removals decrease the amount of carbon in the atmosphere. The source is the origin of emissions, and the sink where the net removals go (i.e., mostly ocean and biosphere). Both, source and sink, are not explicitly represented in this model - but could be in another model.

Pictorial representation

Graphically, we can compose (often called) stock-and-flow or causal (loop) diagrams via the following building blocks.

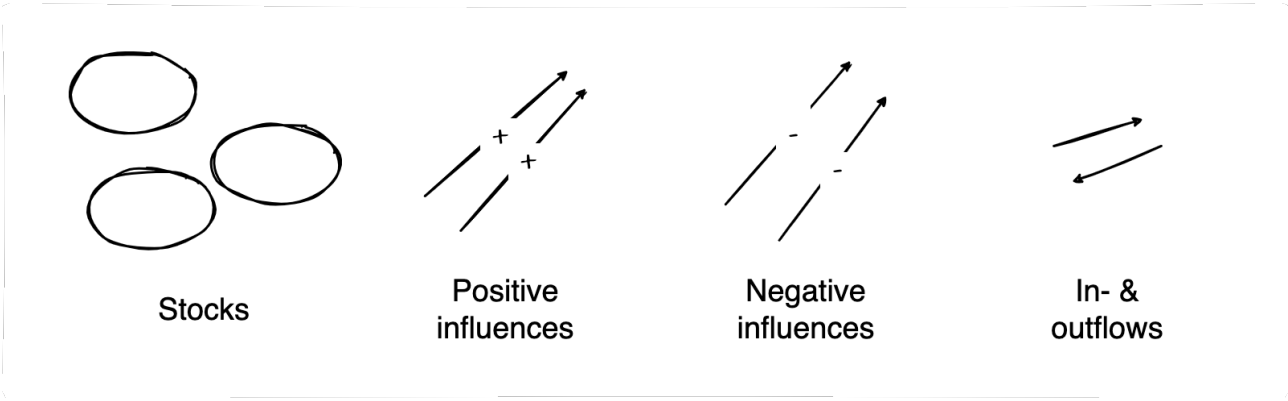


Figure 5: Graphical elements of a dynamic system

For instance, the carbon bathtub could resemble this:

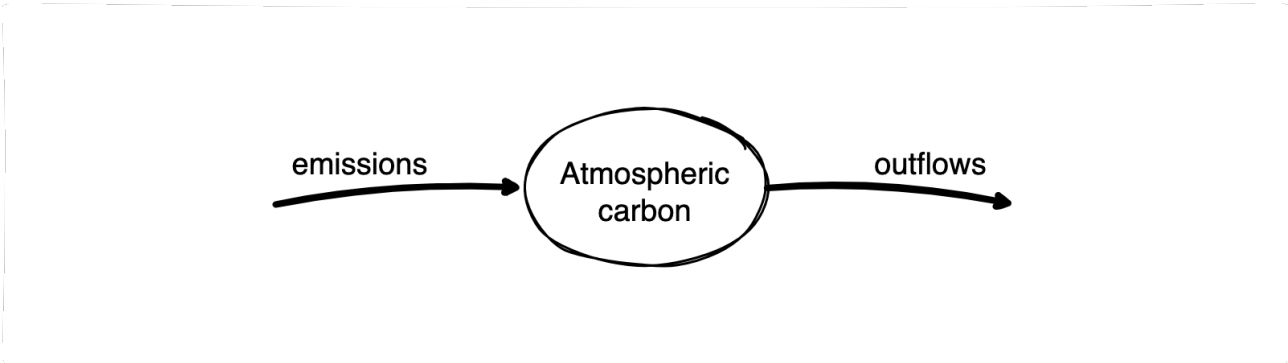


Figure 6: Graphical elements of a dynamic system

[1] In mathematics, the term "dynamical system" is more commonly used. But there is also the related field of "system dynamics", with its own scientific community. I don't want to get into the details of the differences between these two fields. For the purpose of this course, we will use the term "dynamic system" to refer to the overarching ideas.

Such pictorial models are a powerful tool to develop a dynamic systems model and communicate its structure. However, they are limited regarding specifying model details and analysis. For this, a mathematical representation is essential.

Mathematical representation

Mathematically, we use **variables** (such as x, y, and any other letter) as a **placeholder** for the value of a stock. We indicate the value at a **specific time** t by an index (such as x_t, y_t, etc.), assuming that time advances in discrete steps, i.e., $t \in \mathbb{Z}$. To describe the **change of stocks**, we formulate an equation (with +'s and −'s for positive and negative changes). In its most general form, it looks like,

$$x_{t+1} = F(x_t).$$

This means the value of the stock x at time $t+1$ equals the value of the function F, which depends on the value of stock at time t. Note that another common name for dynamic systems in discrete time is *maps*.

For example, the carbon bathtub equations look like,

$$x_{t+1} = x_t + e_t - o_t,$$

where x denotes the atmospheric carbon stock, and $e_t \geq 0$ and $o_t \geq 0$ the amount of emissions and outflow at time t.

This equation shows that the stock at time $t+1$ equals the stock at time x_t plus the inflow of emissions at time e_t minus the outflow at time o_t. Thus, what ultimately determines the stock at time $t+1$ is the difference between the emissions and the outflow at time t. Let $n_t = e_t - o_t$ be the net flow. Then, we can rewrite the equation as

$$x_{t+1} = x_t + n_t.$$

Note that n_t can be positive or negative, depending on the emissions and the outflows.

Sometimes, it can be handy to represent the dynamic system in its **difference form**, directly indicating the **change of stocks**,

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

In analogy to the more common *differential equations* (which work with *continuous time*), we call this form **difference equations**.

For example, the carbon bathtub difference equations look like,

$$\Delta x = e_t - o_t.$$

DeepDive | Why discrete-time models Most dynamical system models consider the continous-time case; but we will focus on discrete time.

Discrete-time models are easy to understand, develop and simulate.

- Computer simulations require time-discretization anyway.
- Experimental data often already discret.
- They can represent abrupt changes.
- They are more expressive using fewer variables than their continuous-time counterparts.

Discrete-time models are a cornerstone in mathematical modeling due to their simplicity and adaptability. They align naturally with computer simulations, as digital systems process time in discrete intervals. This compatibility makes them essential for precise computational analysis. Additionally, experimental data is often recorded at specific intervals, such as daily or monthly, fitting seamlessly with discrete-time models without requiring transformation processes needed for continuous-time models. These models also excel at representing abrupt changes found in real-world phenomena, such as population dynamics or financial markets, capturing these shifts more directly than continuous models. Furthermore, discrete-time models often require fewer variables, enhancing both simplicity and interpretability. This efficiency allows researchers to focus on critical system aspects, making these models powerful tools for theoretical and practical applications alike. In essence, the strengths of discrete-time models lie in their alignment with digital computation, natural fit with discrete data, ability to capture sudden changes, and efficient expressiveness, making them indispensable for scientists and engineers.

Computational representation

There are, in fact, **many ways** to translate the pictorial and mathematical models into a computer model. We start by defining the **function**, $F(x_t)$, from above but give it a more **descriptive name**.

```
def update_stock(stock, inflow, outflow):
    new_stock = stock + inflow - outflow
    return new_stock
```

Now, we are ready to perform our **first model simulation**.

Conceptually, we need to define the **initial value** of the stock. Let's assume we start at 280 parts per million (ppm).

```
stock = 280
```

Technically, we must define a container to store the simulation output. We create a Python list for this purpose and store the stock's initial value inside.

```
time_series = [stock]
```

Conceptually, before we can start the simulation, we must decide **how many time steps** it should run, and on the values of **inflow and outflow**. Let's simulate 150 steps, denoting yearly updates from 1850 to 2000, and assume a **constant flow** with an inflow of 75.6 ppm and an outflow of 75 ppm.

Technically, we loop over the `range` from 1851 to 2001, calling the `update_stock` function with the flow parameter values and appending the new `stock` value to the `time_series` list.

```
for t in range(1851, 2001):
    stock = update_stock(stock, 75.6, 75);
    time_series.append(stock)
```

Finally, we can graphically investigate the output time series of our model simulation.

```
plt.plot(list(range(1850, 2001)), time_series, '.-');
plt.xlabel('time [years]'); plt.ylabel('stock [ppm]');
```

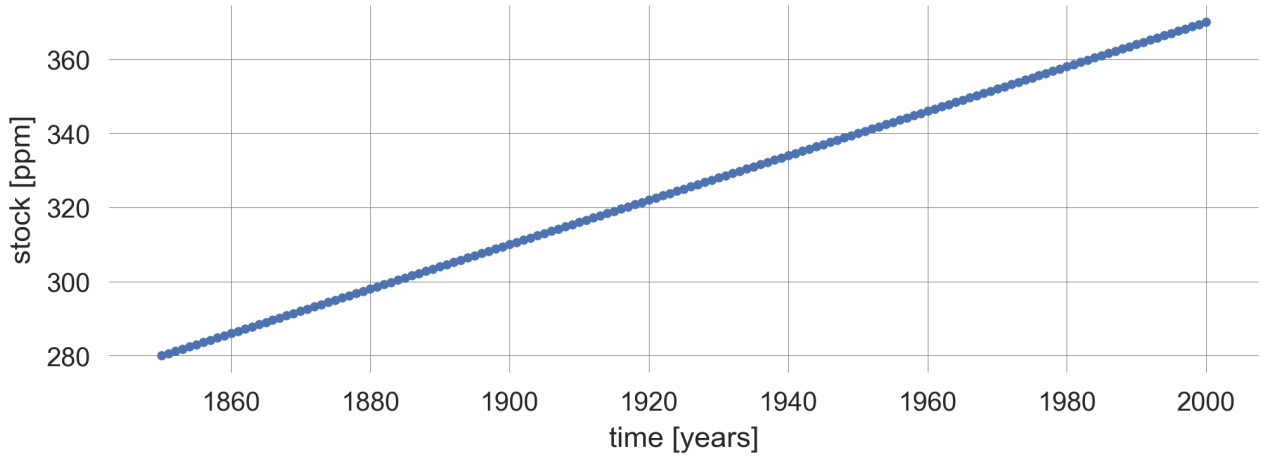


Figure 7: Linear growth

The above code cell plots the time_series data with dots at each data point and lines connecting them.

- `plt` is an alias for `matplotlib.pyplot`, a popular plotting library in Python.
- `plot` is a function that creates a 2D line plot.
- `list(range(1850, 2001))` represents the values to be plotted along the x-axis.
- `time_series` is the data being plotted. It is expected to be a sequence of values (e.g., a list or a NumPy array).
- `'.-'` is a format string that specifies the style of the plot:
  - `'.'` indicates that the data points should be marked with dots.
  - `'-'` indicates that the data points should be connected with lines.
- The second line equips the plot with an x- and a y-label.
  - The `;` at the end of each statement allows for multiple statements in one line.

We observe a rise in CO2 concentration from 280 ppm to 370 ppm, as in the observation data.

However, the shape of the curve is different. Here, we observe just a **linear trend**. The change of stock equals the net flow of inflows minus outflows. If they are constant, the stock evolution is linear.

Linear here means that the change in stock is proportional to the in and outflows. If they are constant, the rate of change is constant and the stock evolution is linear.

A general modeling framework

The carbon bathtub example is perhaps the simplest dynamic model one can imagine. Still, it illustrates the importance of differentiating between a stock and a flow, which is changing that stock.

However, the real power of dynamic system models comes from their **generality** and the possibility to include **feedbacks** in the change of stocks.

Dynamic systems are a very **general modeling framework**. They can model many more phenomena than the evolution of atmospheric greenhouse gas concentrations, from classic examples, such as the *bouncing of a ball*, the *swing of a pendulum*, and the *motions of celestial bodies*, to more advanced dynamics, such as the *evolution of populations*, the *weather and climate*, *neural networks* in the brain, or the *behavior of agents*.

For example, the `update_stock` function we defined above is entirely **agnostic** regarding which kind of stock it models. The Python function does not refer to the stock of atmospheric greenhouse gas concentration. It can **model any system** with one stock where the stock change is independent of the current level of the stock. Or, in other words, systems **without feedback**.

Feedback

In dynamic systems, feedback means that **stock changes depend on current stock levels**.

Positive feedback loops

Consider, for example, the following system, pictorially,

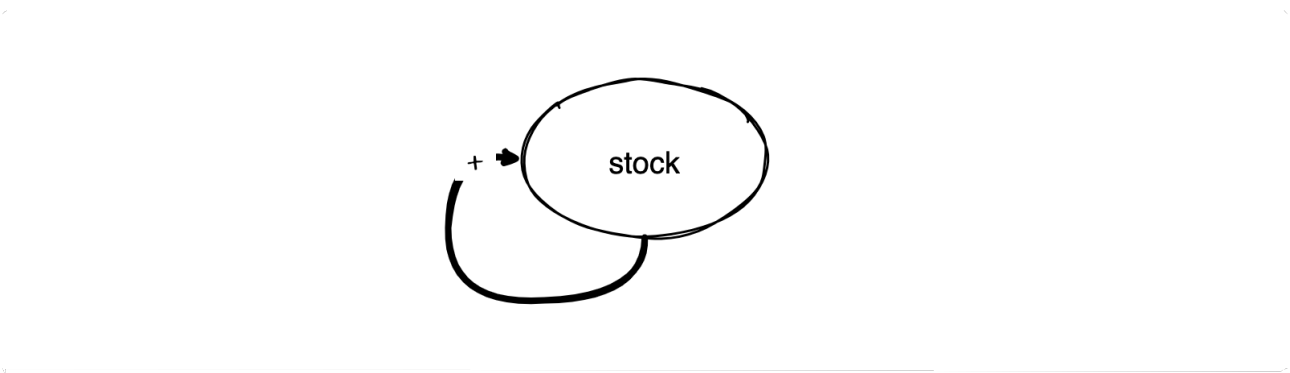


Figure 8: Positive Feedback Loop

or mathematically,

$$x_{t+1} = x_t + rx_t \quad \Leftrightarrow \quad \Delta x = rx$$

with $r > 0$.

Guess what will happen, given a positive initial stock value.

1) The stock will grow in a straight line.
2) The stock will grow faster, i.e., on an upward-bending curve.
3) The stock will grow slower, i.e., on a downward-bending curve.

We will find out.

Let's first define our new `update_stock` function.

```
def update_stock(stock, rate):
    new_stock = stock + rate*stock
    return new_stock
```

To run the model, i.e., to iterate the `update_stock` function, we define an `iterate_model` function.

```
def iterate_model(nr_timesteps, initial_value, update_func, **update_params):
    stock = initial_value
    time_series = [stock]
    for t in range(nr_timesteps):
        stock = update_func(stock, **update_params)
        time_series.append(stock)
    return np.array(time_series)
```

This function takes, the number of time steps, the initial stock value as input arguments. Furthermore, it takes an `update_func` function and flexible `**update_params` as arguments, which allows us to use different stock update functions. It returns a `numpy` array of stock values over time.

The stock value starts at `initial_value`. The list `time_series` is initialized with the starting stock value. This will store the stock value at each timestep.

Then, a loop runs `nr_timesteps` times. The `update_func` function is called in each iteration with the current stock value and the `update_parameters`. The result is assigned to `stock`, updating its value. The new `stock` value is appended to the `time_series` list, which tracks the stock's value at each timestep.

For convenience, we will also define a `plot_stock_evolution` function, plotting the stock evolution.

```
def plot_stock_evolution(nr_timesteps, initial_value, update_func,
                         **update_parameters):
    time_series = iterate_model(nr_timesteps, initial_value,
                                update_func, **update_parameters)
    plt.plot(time_series, '.-', label=str(update_parameters)[1:-1]);
    plt.xlabel("Time steps"); plt.ylabel("Stock value");
    return time_series
```

The `plot_stock_evolution` function calls the `iterate_model` function with the given parameters and plots the `time_series` on the axis, with the format `'.-'` (dots connected by lines) and a legend label indicating the parameters used. The `str(update_parameters)[1:-1]` converts the `update_parameters` dictionary into a string and, with `[1:-1]`, removes the first and last characters (`{` and `}` in the case of a dictionary).

Finally, the function returns the `time_series` list, which contains the stock values at each timestep.

For example, let's consider the phenomenon of CO2 emissions.

Our stock will be the annual CO2 emissions. We assume that we start with 0.01 gigatons of CO2 emissions around 1750 and assume a constant growth rate of 3.3% per year. Where are we 250 years later?

```
plot_stock_evolution(250, 0.01, update_func=update_stock, rate=0.033);
plt.xlabel("Time steps [years]"); plt.ylabel("Anthropogenic\nCO2 emissions
 ↪  [GtC/year]"); plt.legend();
```

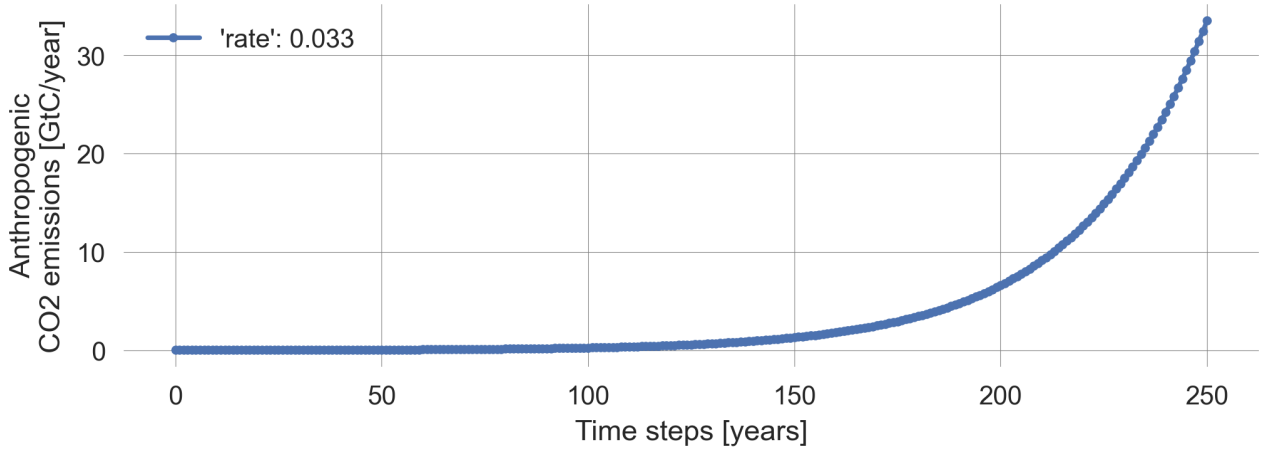
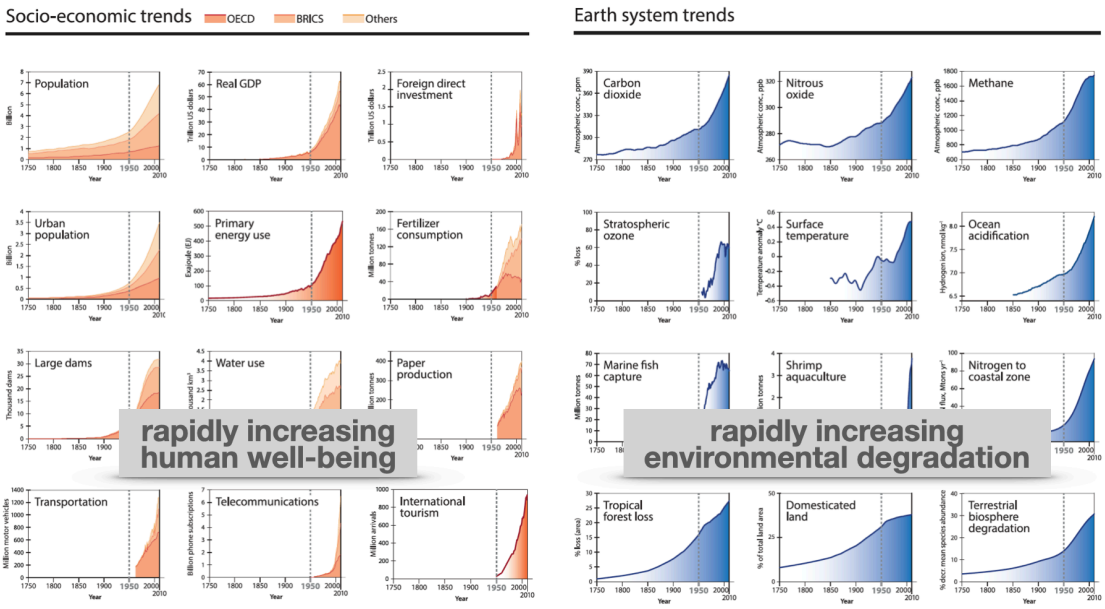


Figure 9: Exponential growth

We reach a **level** of annual CO2 emissions that resembles the empirical observation; additionally, the **trajectory** aligns more closely with the empirical data than the linear growth above (Figure 7).

With the generality of dynamic system models in mind, we can regard the simple positive feedback loop as the **meta-level systems structure of the great acceleration**. The great acceleration (Steffen et al. 2015) refers to the hockey-stick-like growth of many socio-economic and environmental indicators since the mid-20th century (Figure 10).



Steffen et al. (2015) The trajectory of the Anthropocene: The Great Acceleration

Figure 10: The great acceleration

As these developments are not necessarily positive in a normative sense, positive feedback loops are better called **reinforcing feedback loops**.

The worth of a formal model lies in enabling us to conduct **"experiments" safely and at low cost**. By adjusting the input parameters, we see how the output shifts. Overall, our goal is to gain deeper insights into how the output depends on the inputs.

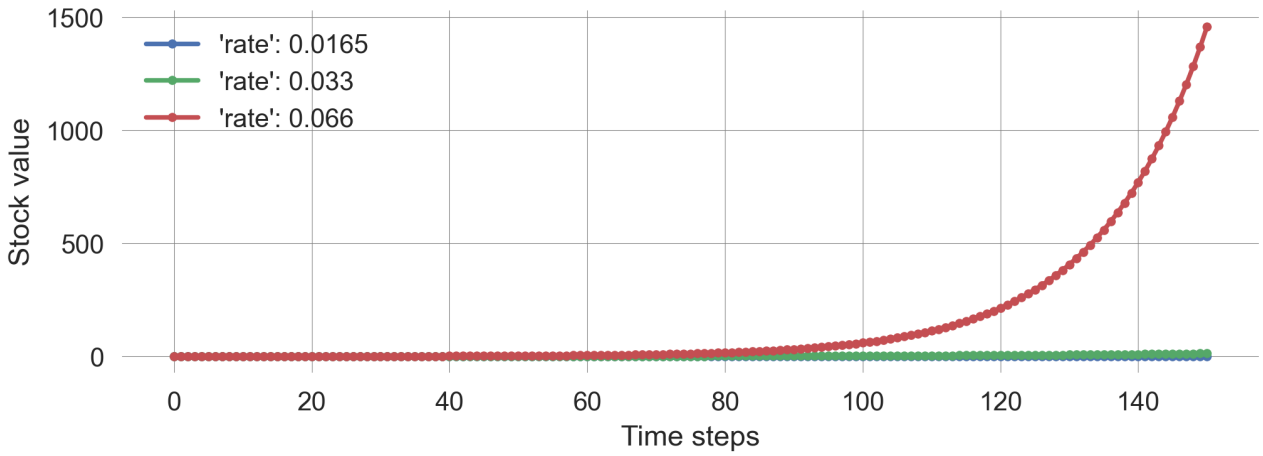
For example, how does the output change if we cut the **rate** of change **in half**? How does the output change if we **double** the rate?

What do you think?

1) A halved rate leads to about a **quarter of the stock** at the end, and a doubled rate leads to about **four times the stock** at the end.
2) A halved rate leads to about a **16th of the stock** at the end, and a doubled rate leads to about **four times the stock** at the end.
3) A halved rate leads to about a **quarter of the stock** at the end, and a doubled rate leads to about **16 times of the stock** at the end.
4) A halved rate leads to about a **10th of the stock** at the end, and a doubled rate leads to about **10 times the stock** at the end.
5) A halved rate leads to about a **10th of the stock** at the end, and a doubled rate leads to about **100 times of the stock** at the end.
6) A halved rate leads to about a **100th of the stock** at the end, and a doubled rate leads to about **10 times of the stock** at the end.

Let's find out.

```
ts_half = plot_stock_evolution(150, 0.1, update_stock, rate=0.033/2)
ts_norm = plot_stock_evolution(150, 0.1, update_stock, rate=0.033)
ts_doub = plot_stock_evolution(150, 0.1, update_stock, rate=0.033*2)
plt.legend();
```



Thus, a halved rate leads to about a **10th of the stock** at the end:

```
ts_norm[-1]/ts_half[-1]
```

```
11.192852530716676
```

And a doubled rate leads to about **100 times of the stock** at the end:

```
ts_doub[-1]/ts_norm[-1]
```

```
111.82334406335414
```

> "*The greatest shortcoming of the human race is our inability to understand the exponential function*" - Albert Allen Bartlett

In summary, in positive or reinforcing feedback loops, positive stock values cause the stock to **increase** proportionally to the stock level, leading to **exponential growth**.

Negative feedback loops

At first glance, negative feedback loops appear quite similar to positive ones.

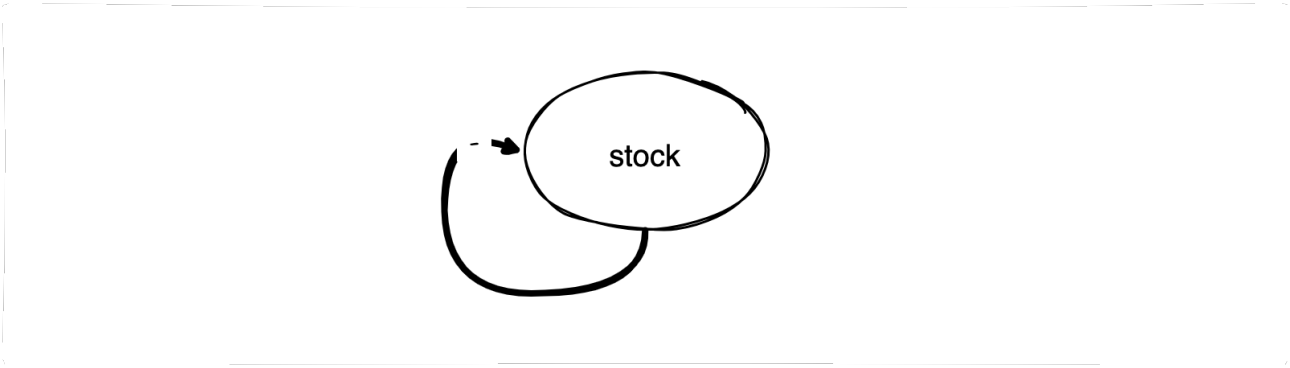


Figure 11: Negative Feedback Loop

However, here, positive stock values cause the stock to **decrease** proportionally to the stock value.

Mathematically, we write

$$x_{t+1} = x_t - rx_t \quad \Leftrightarrow \quad \Delta x = -rx$$

with $r > 0$.

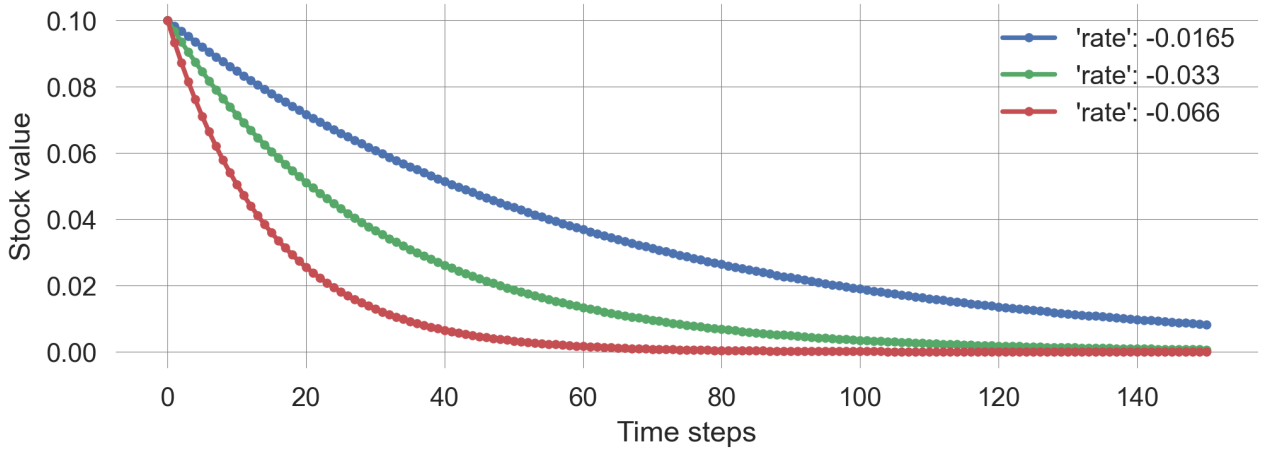
Guess what will happen, given a positive initial stock value.

1) The stock will shrink in a straight line to $-\infty$.
2) The stock will shrink in a straight line to 0.
3) The stock will shrink faster, i.e., on a downward-bending curve to $-\infty$.
4) The stock will shrink faster, i.e., on a downward-bending curve to 0.
5) The stock will shrink slower, i.e., on an upward-bending curve to $-\infty$.
6) The stock will shrink slower, i.e., on an upward-bending curve to 0.

Let's find out.

Fortunately, there's no need to create a new Python function. We can just insert negative growth rates into our current functions.

```
ts_half = plot_stock_evolution(150, 0.1, update_func=update_stock, rate=-0.033/2)
ts_norm = plot_stock_evolution(150, 0.1, update_func=update_stock, rate=-0.033)
ts_doub = plot_stock_evolution(150, 0.1, update_func=update_stock, rate=-0.033*2)
plt.legend();
```



Thus, the stocks shrink **faster than linearly** toward zero, with more negative growth rates causing faster decay. This process is also called **exponential decay**.

Since decay isn't inherently negative in a normative sense—consider environmental degradation—it's more accurate to refer to negative feedback loops as **balancing feedback loops**.

Delays

So far, we have considered only **instantaneous feedback**. The stock change at the next step was caused directly by the current stock level. This is not always the case.

Delays are a common and crucial feature in many dynamic systems. They result from the time it takes for a signal to travel to a stock or, vice versa, for a stock to react to a signal.

How can we **model** the concept of delays in a dynamic system?

In short, we consider **systems with multiple stocks**.

Example | Economy-Innovation interactions

For example, let's consider the phenomenon of **economic growth**. The simplest model explanation is that economic development (e.g., measured by GDP) directly causes more economic development. A slightly refined model explanation might be that economic development causes more innovation (e.g., measured by number of patents), which in turn causes more economic development.

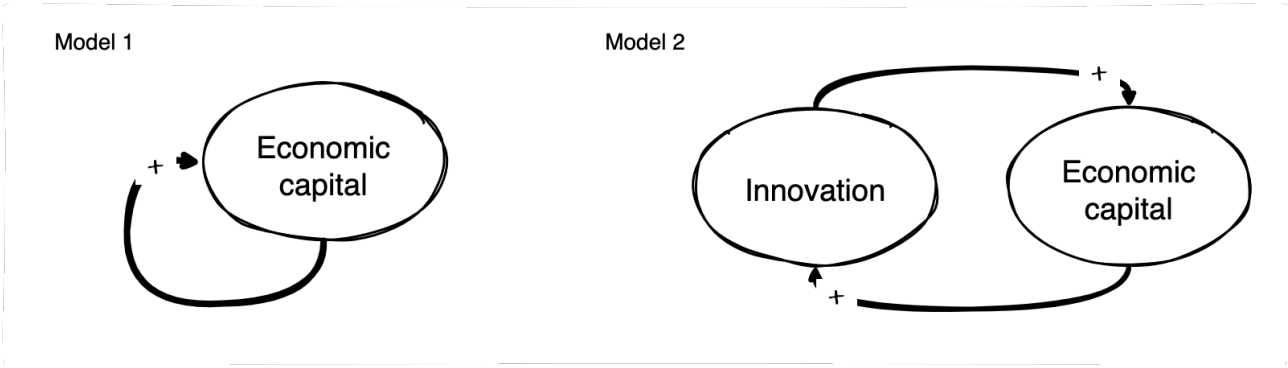


Figure 12: Two models of economic growth

Both models show a **reinforcing feedback loop**, so we should **expect exponential growth** again. But how do the rates of change relate to each other?

Let's first define a **mathematical model**. Let x_t be the level of economic development and y_t the level of innovations at time t.

Model 1:

$$x_{t+1} = x_t + rx_t$$

where $r > 0$ denotes a positive growth rate.

Model 2:

$$x_{t+1} = x_t + ay_t \tag{1}$$
$$y_{t+1} = y_t + bx_t \tag{2}$$

where $a > 0$ denotes the rate of converting innovations to economic development and $b > 0$ denotes the rate of converting economic development to innovations.

Now, we convert the mathematical model into a **computational model**.

First, we define the `update_model` functions.

```
def update_model1(x, r):
    x_ = x + r*x
    return x_
```

```
def update_model2(z, a, b):
    x, y = z
    x_ = x + a*y
    y_ = y + b*x
    return x_, y_
```

Second, we define two `plot_evolution` functions, detailing the plotting of the economic development and innovation levels.

```
def plot_model_evolution1(initial_value, nr_timesteps, **update_parameters):
    time_series = iterate_model(nr_timesteps, initial_value, update_model1,
                                **update_parameters)
    plt.plot(time_series, '.--', label="Economy1 | " + str(update_parameters)[1:-1],
             color='purple');
    return time_series
```

```
def plot_model_evolution2(initial_value_x, initial_value_y, nr_timesteps,
                          **update_parameters):
    z = [initial_value_x, initial_value_y]
    time_series = iterate_model(nr_timesteps, z, update_model2,
                                **update_parameters)
    plt.plot(time_series[:, 0], '.-', label="Economy2 |
 ↪  "+str(update_parameters)[1:-1],
             color='red');
    plt.plot(time_series[:, 1], '.-', #label="Innovation2 |
 ↪  "+str(update_parameters)[1:-1],
             color='blue');
```

```
    return time_series
```

Last, we define a `compare_model` function, which sets the initial levels of the economies as identical and has descriptive parameters for easy interpretation of the model results.

```
def compare_models(economy=1.0, innovation=1.0, timesteps=20, selfrate=0.2,
 ↪  innoTOecon=0.01, econTOinno=4.0,
                    ymax=40):
    plot_model_evolution2(economy, innovation, timesteps, a=innoTOecon,
 ↪  b=econTOinno);
    plot_model_evolution1(economy, timesteps, r=selfrate);
    plt.ylim(0, ymax)
    plt.legend()
```

We set the default parameter so that in *model 1*, there is a default growth rate of 0.2. In *model 2*, we assume that innovations take a long time to result in economic development (i.e., the rate from innovation to the economy is small, a 20th compared to the default rate of *model 1*, to be precise). Even if the rate of converting economic development to innovation is 20 times larger than the default rate, economic growth in *model 2* is slower than in *model 1*.

```
compare_models()
```

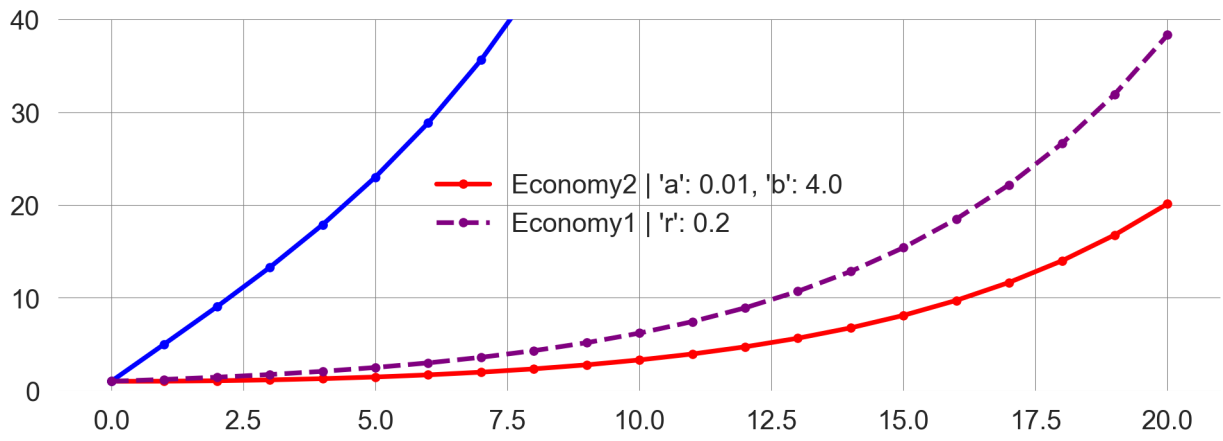


Figure 13: Default Economy-Innovation interactions

How could be boost growth in *model 2*?

One way could be to increase the number of innovations.

It would require a 20-fold increase in the number of innovations to match the growth rate of *model 1*. This is also intuitive: if it takes 20 times longer for innovations to result in economic development, we need 20 times more innovations to achieve the same growth rate. Compare Figure 14 with Figure 13.

```
compare_models(innovation=20)
```

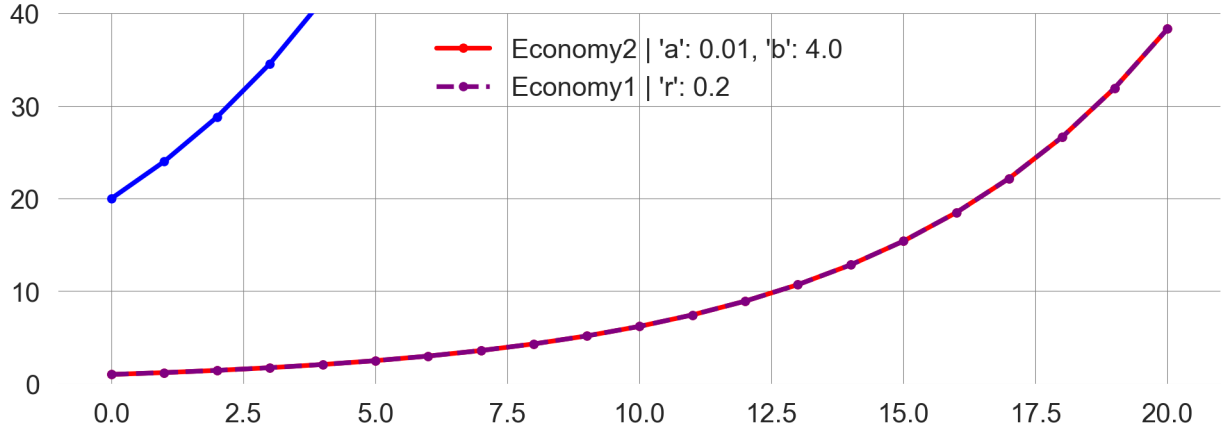


Figure 14: Economy-Innovation interactions with a 20-fold increase in innovations

Is there another way?

How much would the rate of converting economic development into innovations increase to match economic growth in *model 1*?

How much would we need to change the rate of converting innovations to economic development in *model 2* to outperform the economic growth of *model 1*?

We need to increase either the innovation-to-economy or the economy-to-innovation rate by a factor of approx. 1.45 to match the sizes of the economies after 20 time periods (Figure 15).

```
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(121) # LEFT PLOT
compare_models(econTOinno=4.0*1.45)
ax2 = fig.add_subplot(122) # RIGHT PLOT
compare_models(innoTOecon=0.01*1.45)
```

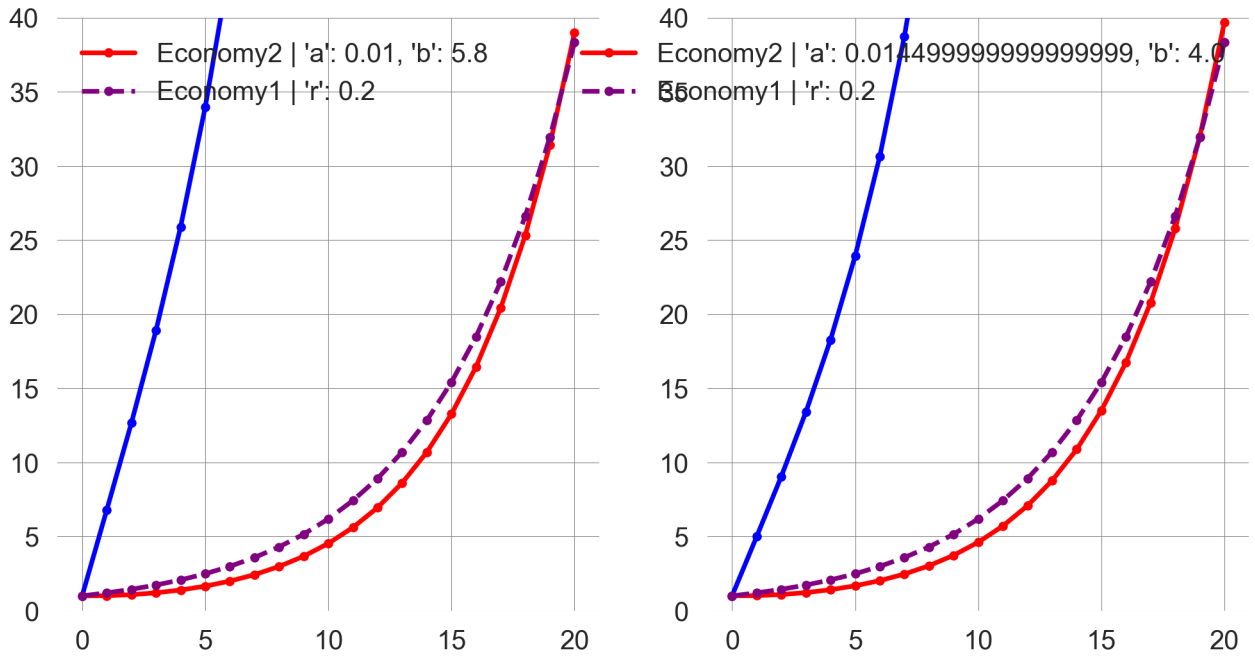


Figure 15: Economy-Innovation interactions with a 1.45-fold increase of a single conversion rate

Since the innovation-to-economy rate is minimal initially, this might be the more accessible lever to pull.

> A key insight for policy interventions from dynamic system models is that it can be much more effective to intervene in the systems' dynamics than in the systems' state variables.

Example | Economy-Nature interactions

One of the defining themes of this course is that we are embedded in the biosphere. Economic growth depends on an intact natural environment, whereas current economic practices negatively impact the state of nature. Let's assume the following feedback structure.

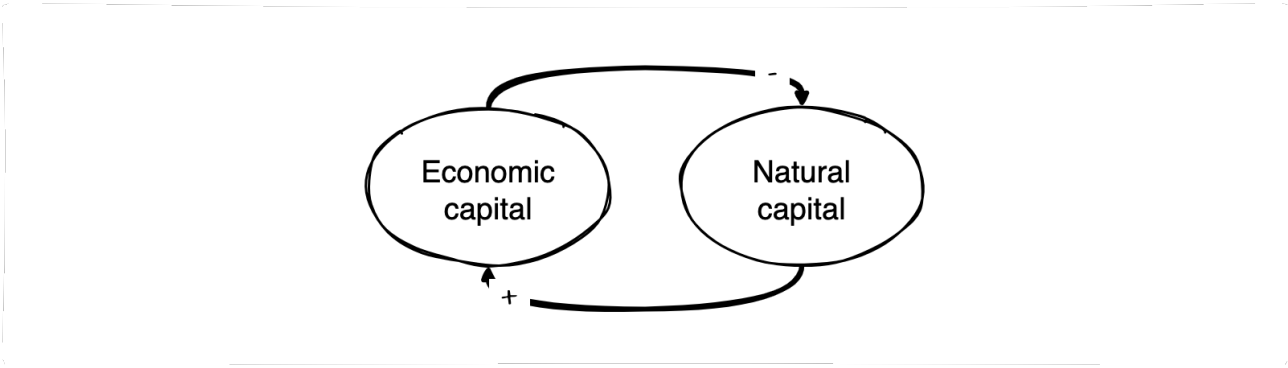


Figure 16: Economy-Nature interactions

Assuming the feedback structure defined in Figure 16, we can reuse our code block from above.

Let's assume that economic and natural capital start at a base level of 1. Economic growth depends positively on the state of natural capital (assuming a base rate of 0.1). In contrast, natural capital changes depend negatively on economic capital but on a slower timescale (let's take a rate of 0.005). Of course, these parameters serve mainly illustrative purposes.

```
def plot_EconomyNature(economy=1.0, nature=1.0, timesteps=100, natTOecon=0.1,
 ↪  econTOnat=-0.005):
    plot_model_evolution2(economy, nature, timesteps, a=natTOecon, b=econTOnat);
    plt.legend()
```

The economy starts growing linearly while nature degrades. At around 60 periods, the economy reaches a maximum and enters a recession while nature continues to degrade ({#fig-EconomyNature100}).

```
plot_EconomyNature()
```

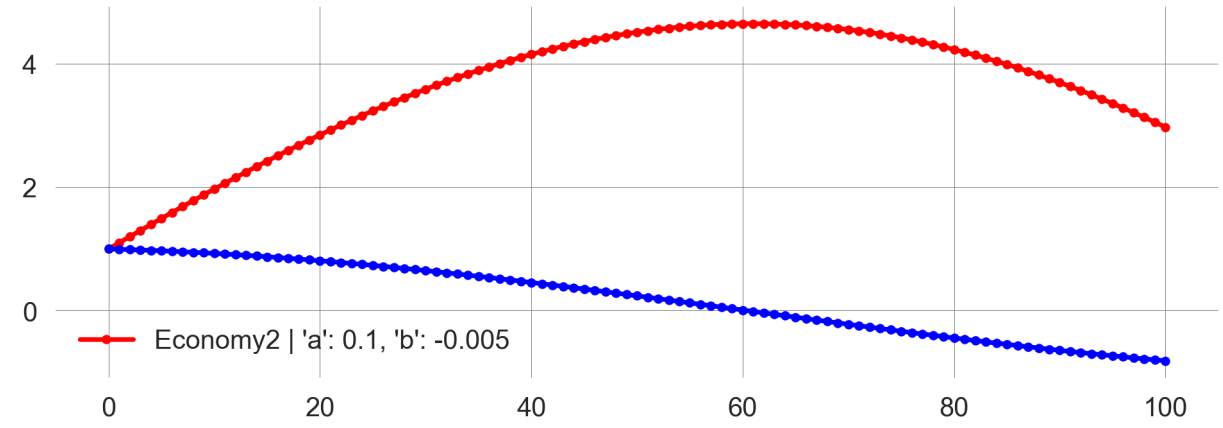


Figure 17

What happens if we continue the simulation for 1000 periods?

```
plot_EconomyNature(timesteps=1000)
```

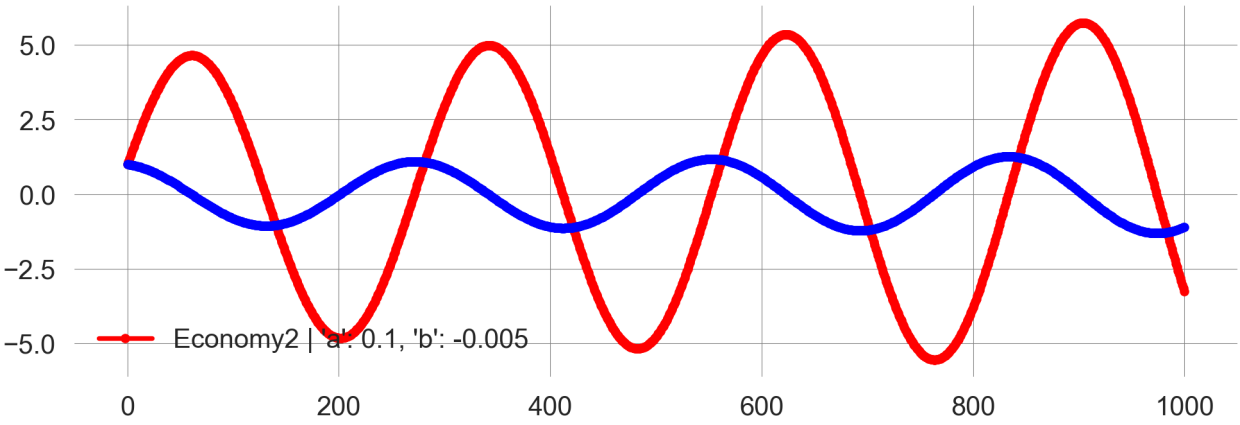


Figure 18

We observe ossilations in the levels of economic and natural capital. The economy grows, but the environment degrades, leading to an economic recession. The environment recovers, and the economy starts growing again, leading to another recession. This cycle repeats indefinitely.

Mathematically, this behavior can be explained by the fact, that both economic and natural capital can have negative values. It is not straitforward to interpret negative values in this context. In effect, a negative value of economic capital results in a net positive effect on natural capital, and a negative value of natural capital results in a net negative effect on economic capital, i.e., when enviornmental damages are high, the economy is likely to suffer.

How special is this ossilatory behavior? Is it due to the specific parameters we chose? Or is it a general feature of the model structure?

To study the possible behavior of a sytem a bit of theory is useful.

DeepDive | Autonomous first-order systems are all you need

Here, we make sure that we do note forget to analyze some system structures. We show that so-called autonomous first-order systems are all we need to model any dynamic system.

In this DeepDive, we answer the question of why we did not consider dynamic equations which take into account the system state of longer than just one time step ago and why we did not model systems which depend explicitly on time. The short answer is, we do not need to. These modifications will not give rise to fundamentally different behavior. We can always represent these modification by introducing additional state variables.

Let us first introduce some terminology.

A system is called *first-order* if it depends only on the state of the system at the previous time step.

First-order system: A difference equation whose rules involve state variables of the immediate past only,

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

Higher-order system: Anything else,

$$x_t = F(x_{t-1}, x_{t-2}, x_{t-3}, \dots).$$

A system is called *autonomous* if it does not depend explicitly on time.

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

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

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

$$x_t = F(x_{t-1}, t).$$

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*,

$$x_t = x_{t-1} + x_{t-2} \quad \text{aka the Fibonacci sequence}$$

can be converted into a first-order form by introducing a "memory" variable,

$$y_t = x_{t-1} \Leftrightarrow y_{t-1} = x_{t-2}$$

Thus, the equation can be rewritten as follows

$$x_t = x_{t-1} + y_{t-1} \tag{3}$$
$$y_t = x_{t-1} \tag{4}$$

This conversion technique works for any higher-order equations as long as the historical dependency is finite.

Similarly, a *non-autonomous equation*

$$x_t = x_{t-1} + t$$

can be converted into an autonomous form by introducing a "clock" variable,

$$z_t = z_{t-1} + 1, z_0 = 0$$

Then,

$$x_t = x_{t-1} + z_{t-1}$$

Take-home message. Autonomous first-order equations can cover all the dynamics of any non-autonomous, higher-order equations. We can safely focus on *autonomous first-order equations* without missing anything fundamental.

Matrix representation

In this section, we will see how to represent a system of difference equations in matrix form. This representation is useful for analyzing the system's behavior, especially when the system has multiple stocks.

Let's consider a general model with two stock variables,

$$x_{t+1} = ax_t + by_t, \tag{5}$$
$$y_{t+1} = dy_t + cx_t. \tag{6}$$

We can rewrite these equations with a matrix multiplication,

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_t \\ y_t \end{pmatrix}$$

This idea **generalizes to any number of stock variables**. Consider a system with n stocks, denoted by x^1, x^2, ..., x^n and influence coefficients a_{ij}, for $i, j \in \{1, 2, \dots, n\}$, denoting the influence stock x_t^2 at time t has on the stock x_{t+1}^1 at time $t+1$. We can convert this logic into the following system of update equations,

$$x_{t+1}^1 = a_{11}x_t^1 + a_{12}x_t^2 + \cdots + a_{1n}x_t^n \tag{7}$$
$$x_{t+1}^2 = a_{21}x_t^1 + a_{22}x_t^2 + \cdots + a_{2n}x_t^n \tag{8}$$
$$\vdots \quad = \quad \vdots \qquad\qquad \cdots \qquad\qquad \vdots \tag{9}$$
$$x_{t+1}^n = a_{n1}x_t^1 + a_{n2}x_t^2 + \cdots + a_{nn}x_t^n. \tag{10}$$

Equivalently, we can summarize all stocks x^1, x^2, ..., x^n into a vector $\mathbf{x}$ and all influence coefficients into a matrix $\mathbf{A}$, with

$$\mathbf{x} = \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{pmatrix}, \qquad \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

Doing so simplifies the form of the update equation to

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t.$$

Some definitions.

We call the number of state variables needed to specify the system's state uniquely the **degrees of freedom**.

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

Thus, the degrees of freedom of a dynamic system equals the dimensionality of its phase space.

Long-term behavior and stability analysis

Playing with a computer model is fun, but the range of possibilities becomes enormous too quickly.

In this section, we will obtain a so-called **closed-form solution** for the time evolution of our systems. This means we write down an equation that gives us the system state for each point in time without the need to iterate the difference equation forward. This is particularly useful if we want to understand the very **long-term behavior** of systems, as this would require many simulation steps. These steps are crucial to understanding what it means for a system state to be **stable**.

Closed-form solution for 1D systems

For one-dimensional systems, we can write

$$x_{t+1} = ax_t$$

This means the system state at time $t = 1$ is $x_1 = ax_0$. At time $t = 2$, the system state is $x_2 = ax_1 = aax_0 = a^2x_0$. Thus, generalizing this pattern yields the system state at time t to be

$$x_t = x_0a^t$$

This means, we can directly calculate the system state at any point in time without the need to iterate the difference equation forward.

For example, let's say we want to calculate only each 10th time step,

```
t = np.arange(0, 101, 10)
t
```

```
array([  0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

The system state is,

```
x_0 = 1.2; r = 0.05
x_0 * (1 + r)**t
```

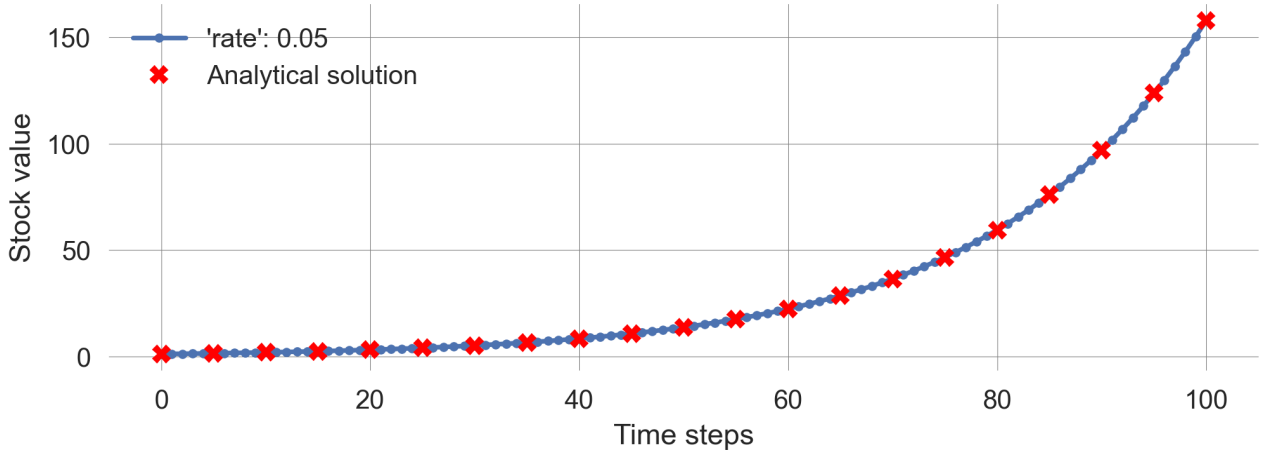
```
array([  1.2       ,   1.95467355,   3.18395725,   5.18633085,
         8.44798645,  13.76087974,  22.41502307,  36.51171064,
        59.47372928,  96.87643806, 157.80150942])
```

Here, we made use of the element-wise exponentiation of NumPy arrays. This means that each element of the array is raised to the power of the corresponding element of the other array. This a very convenient feature of NumPy, as it allows us to perform operations on arrays without the need for explicit loops.

To check, whether our closed-form solution works, we compare it to the simulation results.

```
def compare_solutions(initial_value=1.2, nr_timesteps=100, rate=0.05):
    plot_stock_evolution(initial_value=initial_value, nr_timesteps=nr_timesteps,
                         update_func=update_stock, rate=rate)
    t = np.arange(0, nr_timesteps+1, 5);
    plt.plot(t, initial_value*(1+rate)**t, 'X', color='red', label='Analytical
↪   solution');
    plt.legend()
```

```
compare_solutions()
```



Try it out with different parameter values and observe that the closed-form solution matches the simulation results perfectly.

Cobweb plots

Cobweb plots are a graphical tool to understand the dynamics of one-dimensional systems.

The idea is to plot the system state at time $t + 1$ against the system state at time t. Including the system's udpate function $F(x_t)$, together with the identity line, $y = x$, allows us to see how the system evolves over time. The next system state is obtained by a vertical line from the current state to the system update function. From this point, a horizontal line to the identity line makes the *next* system state, the *current* system state. Thus, the system evolution is represented by horizontal and vertical lines, hence the name, because the resulting picture resembles a cobweb.

```
def cobweb(update_func, initial_value, nr_timesteps=10, ax=None, **update_params):
    x=initial_value; h=[x]; v=[x];  # lists for (h)orizontal and (v)ertical points
    for _ in range(nr_timesteps):  # iterate the dynamical system
        x_ = update_func(x, **update_params)  # get the next system's state
        h.append(x); v.append(x_)  # going vertically (changing v)
        h.append(x_); v.append(x_)  # going horizontially (changing h)
        x = x_  # the new system state becomes the current state

    fix, ax = plt.subplots(1,1) if ax is None else None, ax  # get ax
    ax.plot(h, v, 'k-', alpha=0.5)  # plot on ax
    if np.allclose(h[-2],h[-1]) and np.allclose(v[-1],v[-2]):
        # if last points are close, assume convergence
        ax.plot([h[-1]], [v[-1]], 'o', color='k', alpha=0.7)  # plot dot

    return h, v
```

We study the simple system $x_{t+1} = ax_t$.

```
def Flin(x, a): return a*x
def plotF(a, x0=1.4):
    fix, ax = plt.subplots(1,2, figsize=(12, 3.5));  # axes and limits
    ax[0].set_xlim(-1,2); ax[0].set_ylim(-1,2), ax[1].set_ylim(-1,2)

    xs = np.linspace(-1, 2, 101); # plot F(x) and x
    ax[0].plot(xs, Flin(xs, a), label="F(x)"); ax[0].plot(xs, xs, label="x")
    ax[0].legend(); ax[0].set_xlabel('system state x'); ax[0].set_ylabel('system
 ↪  state x')

    h,v = cobweb(update_func=Flin, initial_value=x0, a=a, nr_timesteps=20,
 ↪  ax=ax[0]);  # include cobweb

    plot_stock_evolution(initial_value=x0, nr_timesteps=20, update_func=Flin, a=a);
    plt.xlabel("Time steps"); plt.tight_layout()  # make axis fit nicely
```

For example, with $a = 0.8$, we observe the cobweb plot in Figure 19.

```
plotF(a=0.8)
```

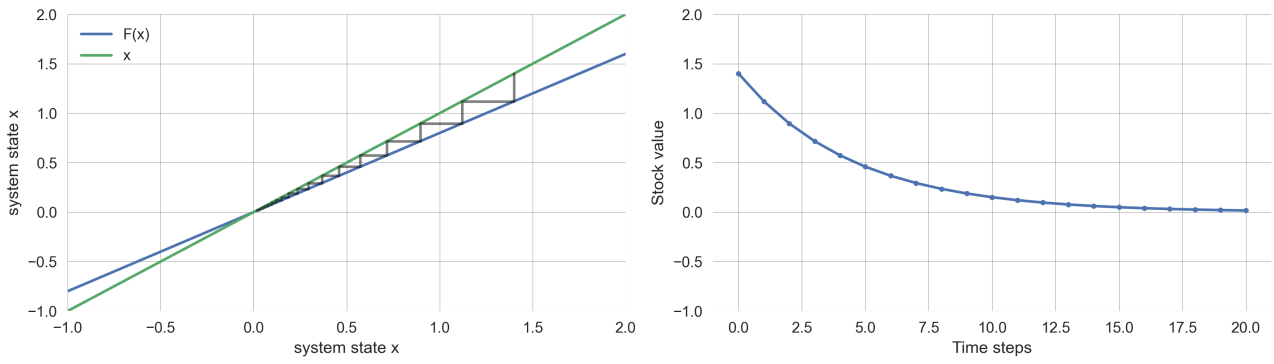


Figure 19

Convince yourself about the following **observations**:

- $1 < a$: Divergences to infinity
- $a = 1$: Conserved behavior
- $0 < a < 1$: Convergence to fixed point
- $-1 < a < 0$: Convergence to fixed point with transient oscillatory behavior
- $a = -1$: Conserved oscillatory behavior
- $a < -1$: Divergend ossiliaroty behavior

We can summarize these observations into **three qualitatively distinct cases** for the asymptotic behavior of linear systems.

1) $|a| < 1$: The system converges to fixed point
2) $|a| > 1$: The system diverges to infinity
3) $|a| = 1$: The system is conserved

How do these observations generalize to multi-dimensional systems?

Multi-dimensional phase space vizualization

Let us first create a general, multi-dimensional update function.

```
def update_general_model(x, A): return A@x
```

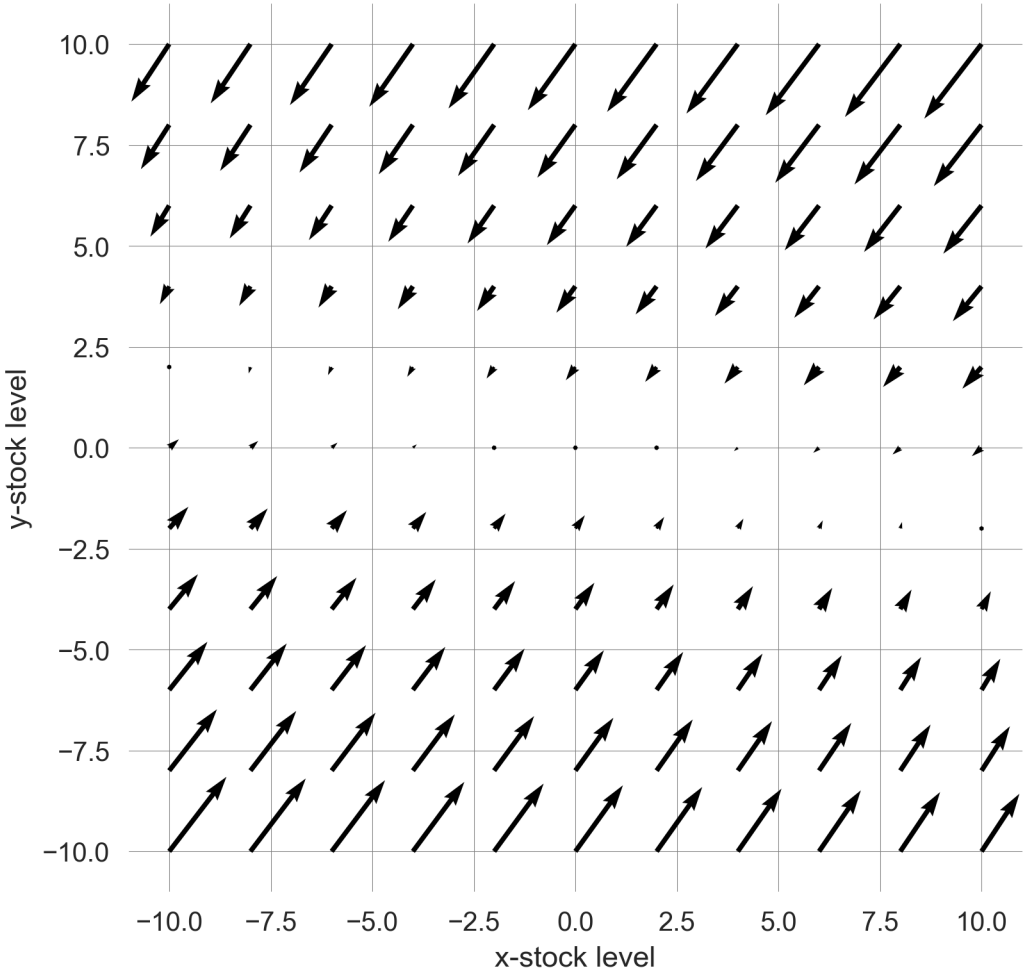
Here, the `@` operator is used for matrix multiplication.

Then, we create a `plot_flow` function, using the `matplotlib.quiver` function.

```
def plot_flow(A, extent=10, nr_points=11, ax=None):
    if ax is None: _, ax = plt.subplots(1,1, figsize=(6,6))

    x = y = np.linspace(-extent, extent, nr_points)  # the x and y grid points
    X, Y = np.meshgrid(x, y)  # transformed into a meshgrid

    dX = np.ones_like(X); dY = np.ones_like(Y)  # containers for the changes
    for i in range(len(x)):  # looping through the x grid points
        for j in range(len(y)):  # looping through the y grid points
            s = np.array([x[i], y[j]]) # the current state
            s_ = update_general_model(s, A) # the next state
            ds = s_ - s # the change in state
            dX[j,i] = ds[0]  # capturing the change along the x-dimension
            dY[j,i] = ds[1]  # captuaring the change along the y-dimension

    q = ax.quiver(X, Y, dX, dY, angles='xy')  # plot the result
    ax.set_xlabel('x-stock level'); ax.set_ylabel('y-stock level')
```

Let's test our `plot_flow` function with a random two-by-two matrix.

```
A = np.random.randn(2,2)
plot_flow(A)
```

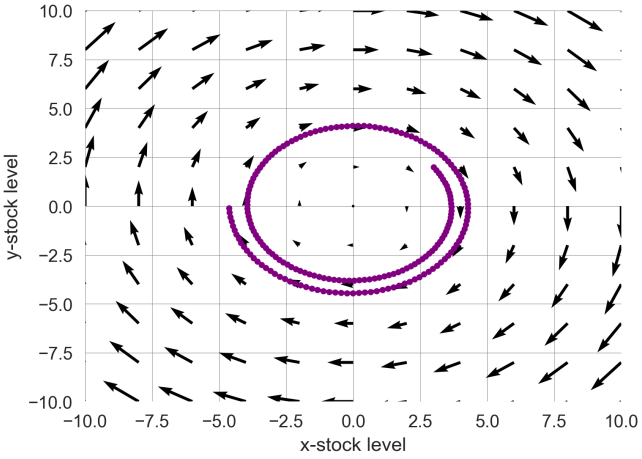


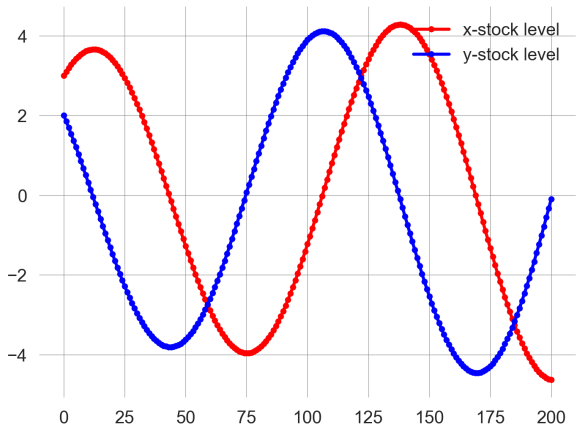
Now, we can visualize the flow of any two-dimensional system, including a trajectory, in the phase space.

We create a `plot_flow_trajectory` function, which plots the flow of a system with a given matrix, and the trajectory of the system over time.

```
def plot_flow_trajectory(a=1,b=0.05,c=-0.05,d=1, nr_timesteps=200):
    fix, ax = plt.subplots(1,2, figsize=(12, 4));  # axes and limits
    # ax[0].set_xlim(-1,2); ax[0].set_ylim(-1,2), ax[1].set_ylim(-1,2)

    A = np.array([[a, b], [c, d]])
    ts = iterate_model(nr_timesteps, [3, 2], update_general_model, A=A)

    plot_flow(A, ax=ax[0])
    ax[0].plot(ts[:,0], ts[:,1], '.-', label='Model trajectory', color='purple')
    ax[0].set_xlim(-10, 10); ax[0].set_ylim(-10, 10);

    ax[1].plot(ts[:,0], '.-', label='x-stock level', color='red')
    ax[1].plot(ts[:,1], '.-', label='y-stock level', color='blue')
    ax[1].legend()
    return ts
```

```
plot_flow_trajectory();
```





Such a phase space visulaization is a powerful tool, connecting the time-evolution of a dynamic systems with a **geometrical representation**.

It allows us to understand the **long-term behavior** of a system, and eventually, how a system's fate **depends on its initial state**.

Closed-form solutions of multi-dimensional systems

Similarly to one-dimensional systems with direct feedback only, a **closed-form solution** to the time evolution of multi-dimensional systems with delays, $\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t$, has the form,

$$\mathbf{x}_k = \mathbf{A}^k \mathbf{x}_0$$

to calculate the system state at time k and study how the system behaves when $k \to \infty$. The only problem is **how to calculate the exponential of a matrix**, $\mathbf{A}^t$.

To study the long-term behavior of multi-dimensional systems with delays, we turn the equation $\mathbf{x}_k = \mathbf{A}^k \mathbf{x}_0$ into a more manageable form. For this purpose, we utilize the **eigenvalues and eigenvectors** of matrix $\mathbf{A}$. To recap, eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of $\mathbf{A}$ are the scalars and vectors satisfying,

$$\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i.$$

Thus, when applying an eigenvector to its matrix effectively turns the matrix into a scalar number (the corresponding eigenvalue). Raising a scalar number to a power is easy. If we repeatedly apply this technique, we get

$$\mathbf{A}^k \mathbf{v}_i = \mathbf{A}^{k-1} \lambda_i \mathbf{v}_i = \mathbf{A}^{k-2} \lambda_i^2 \mathbf{v}_i = \cdots = \lambda_i^k \mathbf{v}_i.$$

Decomposable components. Last, we need to **represent** the **initial system state** $\mathbf{x}_0$ using the **eigenvectors** of matrix $\mathbf{A}$ as the basis vectors, i.e.,

$$\mathbf{x}_0 = c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n,$$

where n is the dimension of the state space and the coefficients $c_1, c_2, \ldots, c_n$ represent the vector $\mathbf{x}_0$ in the eigenvector basis of the $n \times n$ matrix $\mathbf{A}$.

In practice, most $n \times n$ matrices are diagonalizable[2] and thus have n linearly independent eigenvectors. Therefore, we assume we can use them as the basis vectors to represent any initial state $\mathbf{x}_0$. Representing $\mathbf{x}_0$ in the eigenbasis of $\mathbf{A}$ gives us

$$\begin{aligned}
\mathbf{x}_k &= \mathbf{A}^k \mathbf{x}_0 && (11)\\
&= \mathbf{A}^k (c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n) && (12)\\
&= c_1 \mathbf{A}^k \mathbf{v}_1 + c_2 \mathbf{A}^k \mathbf{v}_2 + \cdots + c_n \mathbf{A}^k \mathbf{v}_n && (13)\\
&= c_1 \lambda_1^k \mathbf{v}_1 + c_2 \lambda_2^k \mathbf{v}_2 + \cdots + c_n \lambda_n^k \mathbf{v}_n && (14)\\
& && (15)
\end{aligned}$$

Now, we can clearly see that the system's time evolution, $\mathbf{x}_t$, is described by a summation of multiple exponential terms of λ_i.

> 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.

An eigenvalue tells us whether a particular component of a system's state (given by its corresponding eigenvector) grows or shrinks over time. * When the eigenvalue is greater than 1, the component grows exponentially. * When the eigenvalue is less than 1, the component shrinks exponentially. * When the eigenvalue is equal to 1, the component is conserved.

Dominant components and systems stability. In the long term, the exponential term with the largest absolute eigenvalue $|\lambda_i|$ will eventually dominate the others. Suppose λ_1 has the largest absolute value ($|\lambda_1| > |\lambda_2|, \ldots, |\lambda_n|$), and we factor our λ_1 from the closed-form solution for $\mathbf{x}_t$,

$$\mathbf{x}_t = \lambda_1^t \left(c_1 \mathbf{v}_1 + c_2 \frac{\lambda_2^t}{\lambda_1^t} \mathbf{v}_2 + \cdots + c_n \frac{\lambda_n^t}{\lambda_1^t} \mathbf{v}_n \right).$$

We can see that, eventually, the first term will dominate,

$$\lim_{t \to \infty} \mathbf{x}_t \approx \lambda_1^t c_1 \mathbf{v}_1.$$

The eigenvalue with the largest absolute value is known as the **dominant eigenvalue**, while its related eigenvector is termed the **dominant eigenvector**. This eigenvector determines the asymptotic direction of the system's state. This means if a linear difference equation ($\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t$)'s coefficient matrix, $\mathbf{A}$, has a single dominant eigenvalue, its system state will eventually align with the direction of its corresponding eigenvector, no matter the initial state.

- 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.

[2] This assumption doesn't apply to defective (non-diagonalizable) matrices that don't have n linearly independent eigenvectors. However, such cases are 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 mathematical modeling.

Oscillating behavior. Where does oscillating behavior come from?

In short, when some eigenvalues of a coefficient matrix are complex numbers. Why? The answer lies in **Euler's Formula**, which states that for any real number x,

$$e^i x = \cos(x) + i \sin(x),$$

bridging the world of trigonometric functions (i.e., oscillations) with exponential functions (i.e., the closed-form solutions of linear difference equations). Thus, when some eigenvalues of a coefficient matrix are complex numbers, the resulting system's behavior is rotations around the origin of the system's phase space.

The meaning of the absolute values of those complex eigenvalues is still the same as before:

- if the eigenvalue's absolute value is **larger than one**, $|\lambda| > 1$, we have **instability** in the form of rotations with an expanding amplitude;
- if the eigenvalue's absolute value is **smaller than one**, $|\lambda| < 1$, we have **stability** in the form of rotations with a shrinking amplitude; and
- if the eigenvalue's absolute value **equals one**, $|\lambda| = 1$, we have **conservation** in the form of rotations with a sustained amplitude.

Eigenvalue spectrum.

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.

How to put this into practice/Python?

We use `scipy.linalg.eig` to calculate the eigenvalues and eigenvectors of a matrix.

From the documentation (`scipy.linalg.eig?`) we note that it return two objects, an iterable of eigenvalues `w` and an iterable of eigenvalues `v`. The normalized eigenvector corresponding to the eigenvalue `w[i]` is the column `v[:,i]`.

```
evals, evecs = scipy.linalg.eig(A)
```

For example, the eigenvalues are

```
evals
```

```
array([ 0.90140379+0.j, -1.1914619 +0.j])
```

The eigenvectors are

```
evecs
```

```
array([[ 0.99099361,  0.60104809],
       [-0.13390914,  0.79921285]])
```

For readbility, we store eigenvalues and eigenvectors in new variables.

```
eval1, eval2 = evals
print(eval1)
print(eval2)
```

```
(0.9014037889457369+0j)
(-1.1914619046393624+0j)
```

Since the eigenvectors are return in the format in which they are return we need to transpose them to assign them to two separate variables.

```
evec1, evec2 = evecs.T
# We check that we did not make any mistake:
print("This should be zeros:", evec1 - evecs[:,0])
print("This too:", evec2 - evecs[:,1])
```

```
This should be zeros: [0. 0.]
This too: [0. 0.]
```

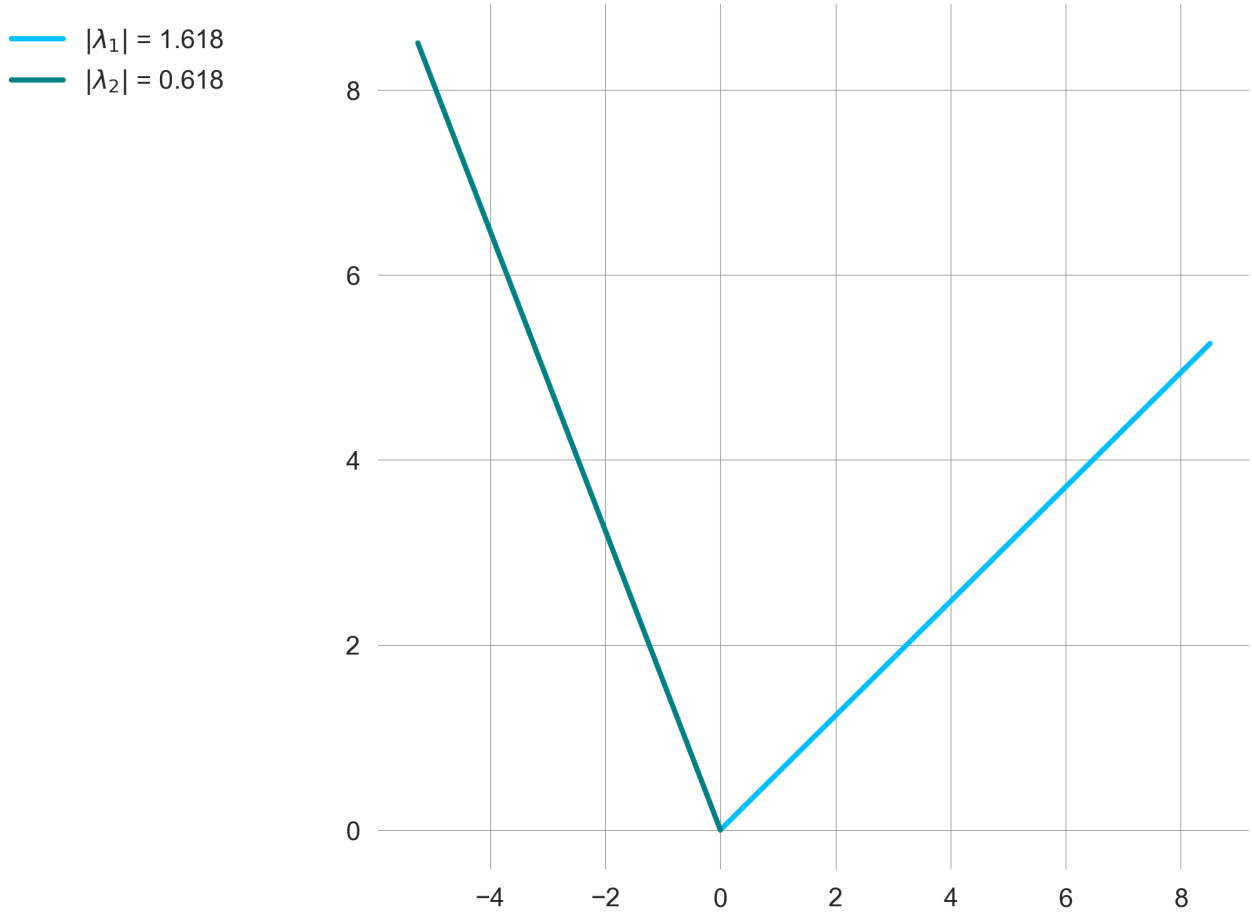
We can autmate such checks with the `assert` statement.

```
assert np.allclose(evec1, evecs[:,0]), "The first eigenvector is not correct"
assert np.allclose(evec2, evecs[:,1]), "The second eigenvector is not correct"
```

Now, we create a `plot_eigenvectors` function.

```
def plot_eigenvectors(a,b,c,d, extent=10, ax=None):
        if ax is None: _, ax = plt.subplots(1,1, figsize=(6,6))

        A = np.array([[a, b], [c, d]])
        evals, evecs = scipy.linalg.eig(A)
        eval1, eval2 = evals
        evec1, evec2 = evecs.T

        # plotting the real part of the eigenvectors
        ax.plot([0, extent*evec1[0].real], [0, extent*evec1[1].real], '-',
                lw=2, color='deepskyblue',
                label='$|\lambda_1|$ = {}'.format(np.abs(eval1).round(4)))
        ax.plot([0, extent*evec2[0].real], [0, extent*evec2[1].real], '-',
                lw=2, color='teal',
                label='$|\lambda_2|$ = {}'.format(np.abs(eval2).round(4)))

        ax.legend(loc='upper right', bbox_to_anchor=(-0.15, 1))
```

```
plot_eigenvectors(a=1, b=1, c=1.0, d=0)
```



Putting it all togehter, we observe how the eigenvector represent the long-term behavior of the system (Figure 20).

```
plot_flow_trajectory_with_ev()
```

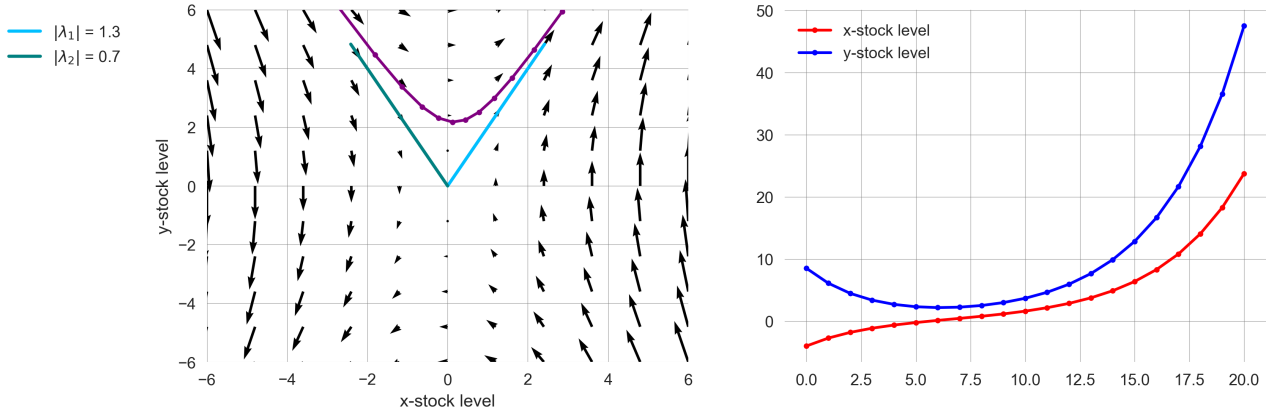


Figure 20

Summary | Systems with linear changes

	k th-component is ...	if λ_k is complex-conjugate, the kth-component is rotating around the origin	if λ_k is dominant
$\|\lambda_k\| > 1$	growing	with an expanding amplitude.	system unstable, diverging to infinity
$\|\lambda_k\| < 1$	shrinking	with a shrinking amplitude.	system stable, converging to the origin.
$\|\lambda_k\| = 1$	conserved	with a sustained amplitude.	system stable, dominant eigenvector component conserved,system may converge to a non-zero equilibrium point

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

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. (Sayama 2023)

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.

Non-linear changes

Systems with non-linear changes, often called just non-linear systems, are defined as systems who are not linear ($\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t$). In other words, they are systems whose rules involve non-linear combinations of state variables.

While **linear systems exhibit relatively simple behavior** (exponential growth/decay, periodic oscillation, stationary states (no change), or their hybrids (e.g., exponentially growing oscillation)), **non-linear systems can exhibit a much wider range of behaviors**, including chaotic dynamics, bifurcations, and limit cycles. As a result, there is no general way to obtain a closed-form solution for non-linear systems, making them much more challenging to analyze and predict than linear systems.

The **logisitc map** is a classic example of a one-dimensional nonlinear system. It is defined as

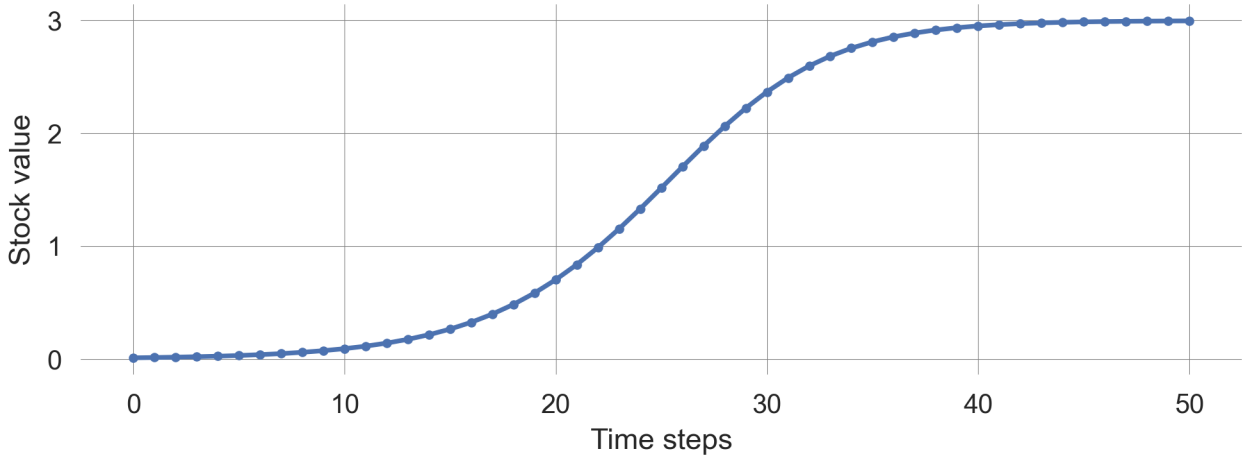
$$x_{t+1} = x_t + rx_t(1 - \frac{x_t}{C}),$$

where r is the growth rate and C the carrying capacity. In Python, it can be implemented as follows:

```
def logistic_map(x, r, C=1): return x + r*x*(1-x/C)
```

Plotting the logistic map for a growth rate of 0.25, and a carrying capacity of 3.0, results in the following time evolution:

```
plot_stock_evolution(50, 0.01, update_func=logistic_map, r=0.25, C=3.0);
```



Finding equilibrium points

An *equilibirum* point x_e (also called fixed points or steady states) is a point in the state space, where the system can stay unchanged over time.

$$x_e = F(x_e)$$

In other words, if the system state is at an equilibrium point, it will remain there indefinitely. Fixed points are **theoretically important** as a meaningful reference when we understand the structure of the phase space. They are of **practical relevance** when we want to sustain the system at a certain desirable state.

To find the equilibrium points of a system, we need to solve the equation $x_e = F(x_e)$ for x_e. This can be done by setting $x_{t+1} = x_t = x_e$ in the system's update function and solving for x_e.

For example, in the logistic map, we obtain,

$$x_e = x_e + r x_e (1 - \frac{x_e}{C}) \tag{16}$$

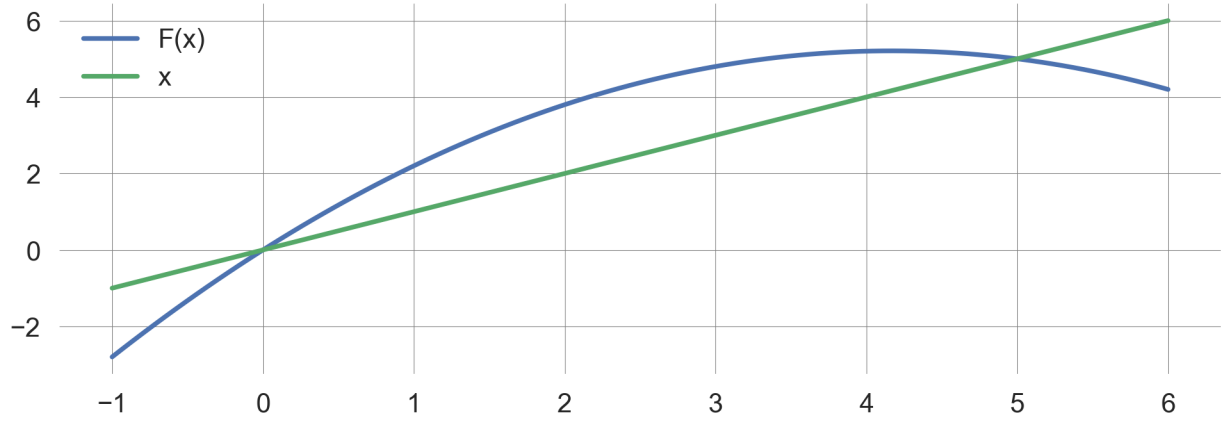
$$0 = 0 + r x_e (1 - \frac{x_e}{C}) \tag{17}$$

$$0 = r x_e (1 - \frac{x_e}{C}) \tag{18}$$

This equation is fulfilled if either $x_e = 0$ or $x_e = C$. Thus, the logistic map has two equilibrium points, $x_e = 0$ and $x_e = C$.

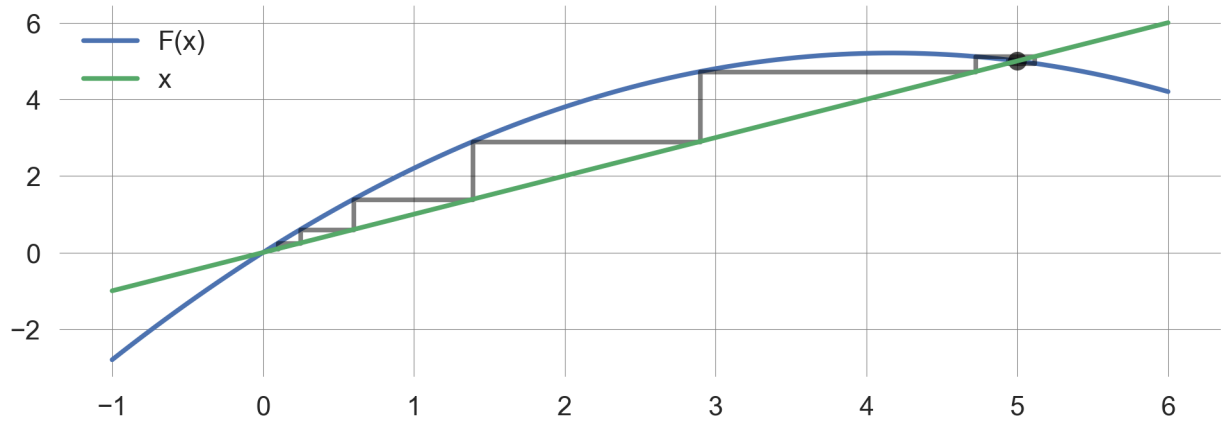
Graphically, fixpoints of an iterative map are the intersections between $F(x)$ and x.

```
xs = np.linspace(-1, 6, 101)  # Resolution of the x axis
plt.plot(xs, logistic_map(xs, 1.5, 5), label="F(x)")  # Plot the map x_=F(x)
plt.plot(xs, xs, label="x")  # Plot the diagonal x_=x
plt.legend();  # Include the legend
```



We can enrich this representation with the cobweb plot, which shows the system's time evolution from an initial state to the equilibrium point.

```
xs = np.linspace(-1, 6, 101)
plt.plot(xs, logistic_map(xs, 1.5, 5), label="F(x)")
h,v = cobweb(logistic_map, initial_value=0.1, r=1.5, C=5,
             nr_timesteps=100, ax=plt.gca());
plt.plot(xs, xs, label="x")
plt.legend();
```



Linear stability in nonlinear systems

Unfortunately, it is impossible to forcast the asymptotic behaviors of nonlinear systems in the same way as for linerar systems.

However, the concept of stability in linear systems can be applied to equilibirum points of non-linear systems. > 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.

Consider the system $x_{t+1} = F(x_t)$ with steady state x_e.

To analyze its stability around this equilibrium point, we switch our perspective from a global coordinate system to a local one. We zoom in and capture a small perturbation added to the equilibrium point, $z_t = x_t - x_e$. Inserting x_t into the update equation, yields

$$x_e + z_t = F(x_e + z_{t-1}).$$

Since z is small (by assumption), we can approximate the right-hand side as a Taylor expansion,

$$x_e + z_t = F(x_e) + F'(x_e)z_{t-1} + O(z_{t-1}^2)$$

where, F' is the derivative of F with respect to x.

Using $x_e = F(x_e)$, we obtain the simple linear difference equation,

$$z_t \approx F'(x_e)z_{t-1}.$$

> To determine the stability of fixed points in non-linear systems, we need to look at the derivative of $F(x_e)$ at the fixed point.

There are **three qualitatively distinct cases** for the linear stability of a steady state in a non-linear system.

1) $|F'(x_e)| < 1$: The equilibirum point x_e is stable.
2) $|F'(x_e)| > 1$: The equilibirum point x_e is unstable.
3) $|F'(x_e)| = 1$: The equilibirum point x_e is neutral [3]

For example, for the logistic map,

$$x_{t+1} = F(x_t) = x_t + rx_t(1 - \frac{x_t}{C})$$

we calculate the derivative of $F(x_t)$ as

$$F'(x) = 1 + r - 2r\frac{x}{C}.$$

At the fixed points $x_e = 0$ and $x_e = C$, we have

$$F'(x)|_{x=0} = 1 + r, \quad \text{and} \quad F'(x)|_{x=C} = 1 - r.$$

Graphically, we obtain Figure 21.

```
plot()
```

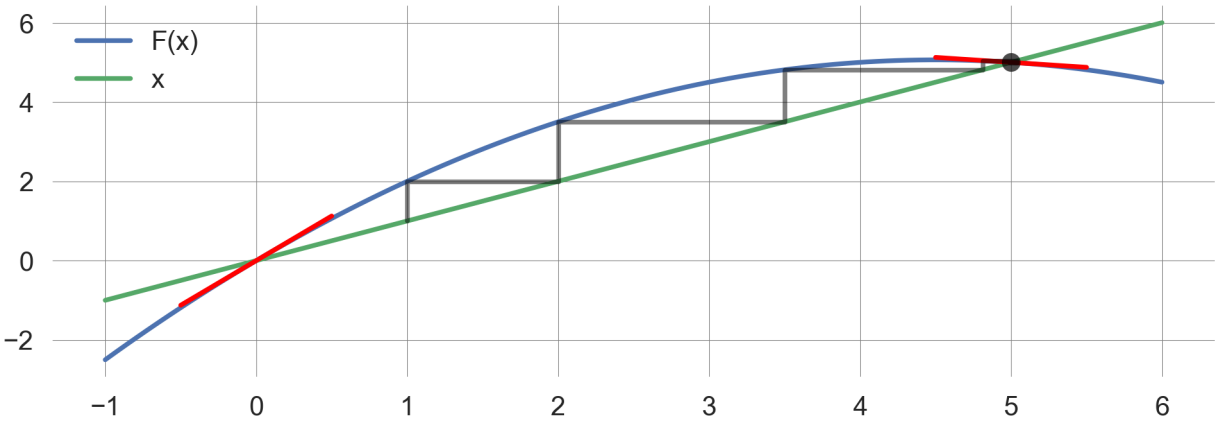


Figure 21: Linear stability shown at the nonlinear logistic map.

[3] also know as *Lyapunov* stable. More advanced nonlinear analysis is required to show that an equilibrium point is truly neutral.

We **observe**, that when $0 < r < 2$, the system convergens to $x_e = C$. When $r > 2$, the system converges to $x_e = 0$. When $r > 2$, $F'(x)|_{x=C} = 1 - r < -1$ causing unstable ossilations.

Learning goals revisited

- Define and describe the **components of a dynamic system**.
  - At their core, dynamic systems consist of stocks and flows.
- **Represent** dynamic system models in visual and mathematical form.
  - In general, a dynamic system iterates via $\mathbf{x}_{t+1} = F(\mathbf{x}_t)$.
- Explain the concepts of **feedback loops** and **delays**.
  - Reinforcing (positive) feedback loops lead to divergence/instability.
  - Balancing (negative) feedback loops lead to convergence/stability.
  - Considering delays makes system more complicated.
- Explain two kinds of **non-linearity** and how they are related.
  - Dynamic systems with linear changes can be represented as $\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t$, and can exhibit non-linear behavior, such as exponential growth or decay, periodic osscilations, or stationary states, or their combinations.
  - Dynamic systems with non-linear changes can exhibit more kinds of behaviors.
- **Implement** dynamic system models and **visualize model outputs** using Python, to interpret model results.
- **Analyze** the **stability** of equilibrium points in dynamic systems using linear stability analysis.

Bibliographical and Historical Remarks

Raworth (2017) (Chapter 4) and Page (2018) (Chapter 18) provide great conceptual introductions to the topic without going into mathematical details.

Sayama (2023) heavily inspired some of the material in this chapter.

References

Page, Scott E. 2018. *The Model Thinker: What You Need to Know to Make Data Work for You*. Basic Books.

Raworth, Kate. 2017. *Doughnut Economics: Seven Ways to Think Like a 21st-Century Economist*. Chelsea Green Publishing.

Sayama, Hiroki. 2023. *Introduction to the Modeling and Analysis of Complex Systems*. https://math.libretexts.org/Bookshelves/Scientific_Computing_Simulations_and_Modeling/Book%3A_Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama).

Steffen, Will, Wendy Broadgate, Lisa Deutsch, Owen Gaffney, and Cornelia Ludwig. 2015. "The Trajectory of the Anthropocene: The Great Acceleration." *The Anthropocene Review* 2 (1): 81–98. https://doi.org/10.1177/2053019614564785.

Sterman, John D., and Linda Booth Sweeney. 2007. "Understanding Public Complacency about Climate Change: Adults' Mental Models of Climate Change Violate Conservation of Matter." *Climatic Change* 80 (3): 213–38. https://doi.org/10.1007/s10584-006-9107-5.