Complex Systems Modeling of Human-Environment Interactions

Wolfram Barfuss

June 24, 2025

Table of contents

Appendices 257

Ex | Individual Learning

Preface

To my dad, the enthusiast of online lecture materials in Python.

These lecture notes (Web, PDF, Google Colab) provide an integrated perspective on different modeling approaches (dynamics, equilibrium, and agent-based) applied to human-environment interactions and sustainability transitions. Its scope is practical, utilizing Python within Jupyter Notebooks. Prerequisites are basic mathematical skills.

This is version 0.2. To be updated and improved.

Learning goals

After **working** through all materials,

- students can **model** *human-environment interactions* to answer relevant questions in sustainability science.
- students can **implement** *models of human-environment interactions* in the general-purpose computer language **Python**.
- students can critically **evaluate** *models of human-environment interactions* to judge their relevance to issues in sustainability science.

Integrated writing and reproducibility

The following steps are entirely optional for readers who want to focus on the learning goals and are not interested in the development of these lecture notes.

These lecture notes are written in Jupyter Notebooks, which are a popular format for interactive computing. Notebooks contain code, math, and text. The code is written in Python, a general-purpose programming language widely used in scientific computing and other fields.

Scholarly writing practices, such as citations and cross-references, are facilitated by Quarto, a powerful scientific and technical publishing system. Quarto also allows you to view these lecture notes in various formats, such as HTML and PDF.

Assuming you have installed the Quarto CLI, you can render these lecture notes by running the following command in the terminal:

```
!quarto render .
```

The comment `#| output: false` is a Quarto directive that prevents the output of this cell from being displayed in the rendered documents. This is useful for keeping these readable.

These lecture notes are made open-source and hosted in a GitHub repository. To convert this `index.ipynb` file (which is required in the Quarto Book project type) into the repository's README file, one may execute the following commands:

```
!quarto convert index.ipynb # convert into Quarto markdown
!tail -n +10 index.qmd > README.md  # remove some metadata+
!rm index.qmd  # remove the intermediate file
```

After configuring the settings for GitHub Pages, one can publish the web version of these lecture notes by running the following command (at the root of the cleaned main branch):

```
quarto publish gh-pages
```

Acknowledgements

I am grateful to all the students I had the pleasure of working with on this material. Their feedback has been, is, and will continue to be essential for shaping this content. I would also like to thank my teachers and mentors, who have influenced my thinking. I am thankful to all contributors and creators of the many open-source projects these notes build upon, such as the Python language and its ecosystem, Jupyter, and Quarto. Furthermore, I acknowledge many helpers who may use some form of generative AI, such as ChatGPT, Perplexity, GitHub Copilot, and Grammarly. All remaining errors remain my own.

1 Sustainability Systems Science

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

> This chapter introduces the basic rationale for **sustainability systems science**, i.e., complex systems modeling and its application to human-environment interactions and sustainability transitions.

1.0.1 Learning goals

After this chapter, students will be able to **explain**:

- Why sustainability transitions require a coupled systems approach.
- Why we must model, and how do it well?
- Systems reductionism and its relation to complex systems.
- Structural challenges for sustainability transitions and three types of models to tackle them.

1.1 Human-environment interactions for sustainability transitions

1.1.1 The state of the planet

Watch this TED talk by Johan Rockström, who offers the 2024 scientific assessment of the state of the planet and explains what must be done to preserve Earth's resilience to human pressure.

While watching, ask yourself the following questions (and make notes around your answers):

- What are the main challenges facing humanity in the 21st century?
- In which ways humans and the environment are interconnected?
- What is the most impressive fact you learned from the talk?

1.1.2 Why are we not acting?

Given the rather grim assessment of the planet's state, the question arises: **Why are we not acting more toward a safe and just future for all**

- despite all the scientific progress we have made so far,
- despite all the knowledge about the risks and undesirable consequences ahead if we do not change our course of action,
- despite all the knowledge we have obtained about possible solutions?

Reflect on this question on your own. Come up with a list of factors most relevant to you.

1.1.3 A failure of systems thinking

According to John Sterman, professor and director of the MIT System Dynamics Group, the underlying issue of inaction in the sustainability crises is a **massive failure of systems thinking**.

Watch the part of his talk (from minutes 12:19 until 14:17) where he explains this failure and **compare his assessment with your own from above**. Did he miss some critical factors? How can we complement his assessment?

These lecture notes offers a comprehensive and opinionated, but foremost, practical introduction to the field of complex systems modeling applied to human-environment interactions and sustainability transitions.

1.2 Modeling

1.2.1 We cannot not model

In information theory, a bit (i.e., a `0` or a `1`) stores the answer to a yes-or-no question. We can measure the rate of information transmitted with the number of bits per second.

11.000.000 bits/s   ~ 50 bits/s

Human physiology requires simplification
We model all the time
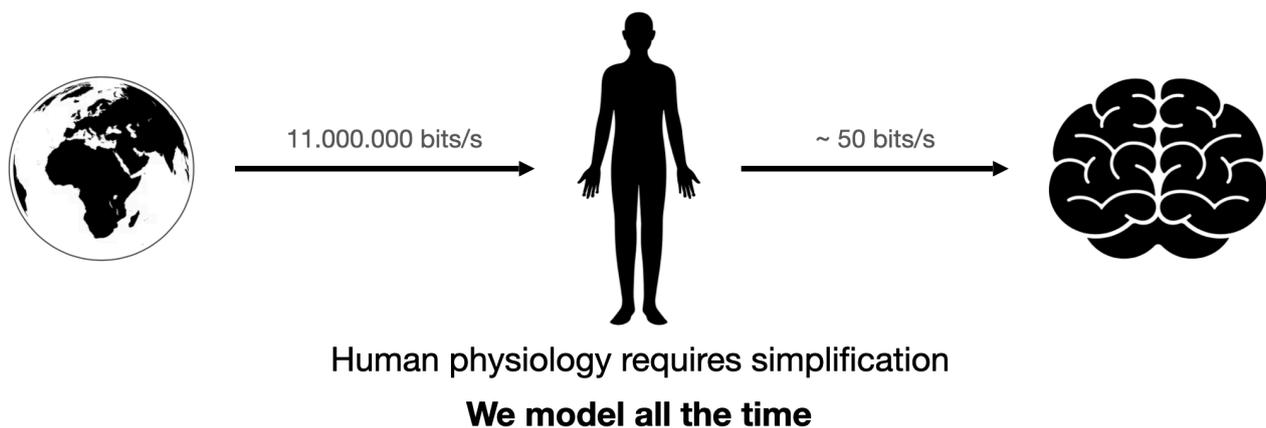
Figure 1.1: We cannot not model - Human physiology requires simplification

For example, it is known that our senses gather some 11 million bits per second from the environment (britannica.com). When applied to the human brain, you expect it to show tremendous information processing capability.

Interestingly, when researchers attempted to assess information processing capabilities during "intelligent" or "conscious" activities—like reading or playing the piano—they found a maximum capability of under 50 bits per second (Figure 1.1). *For instance*, a typical reading speed of 300 words per minute translates to about five words per second. Assuming an average of five characters per word and roughly two bits per character results in that 50 bits per second figure. The precise number can vary based on assumptions and may differ according to the individual and the specific task.

Thus, a tremendous amount of compression occurs if 11 million bits are reduced to less than 50. **Our human physiology requires simplifaction. We model all the time.** Note that the discrepancy between the amount of information being transmitted and the amount of information being processed is so large that any inaccuracy in the measurements is insignificant.

1.2.2 All models are wrong

> A model is a simplified representation of reality.

A model's simplification is necessary to make the phenomenon under question tractable and understandable. Simplifaction here is a feature, not a bug (Smaldino, 2017). **The purpose of a model is to be wrong**.

The models we use come in different forms or media.
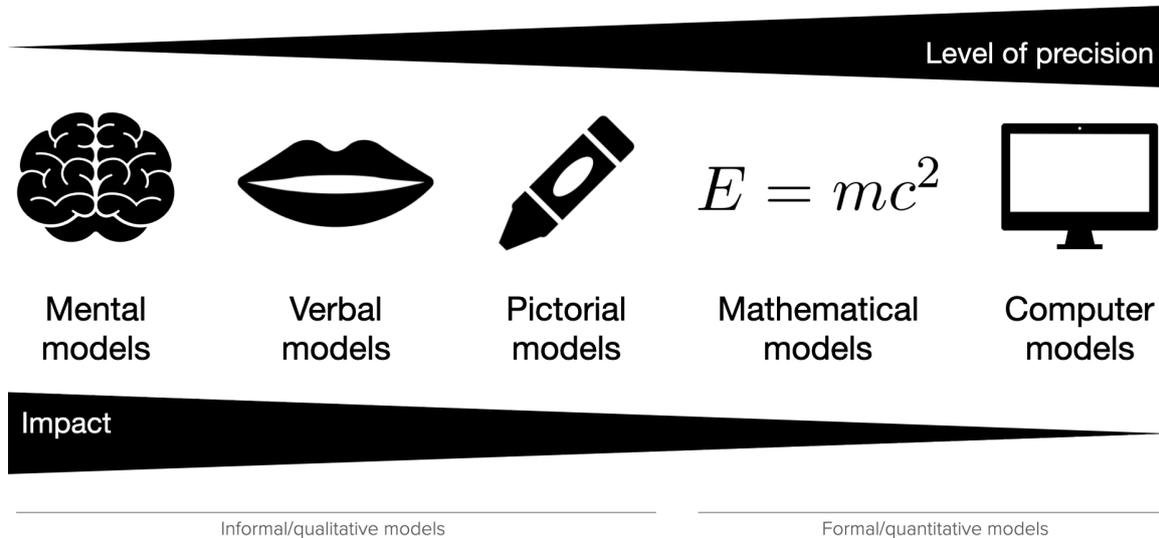
Figure 1.2: Different model forms or media

Some are *informal* and *qualitative*, while others are more *formal* and *quantitative* (Figure 1.2). **Mental models** are intuitive and often subconscious. **Verbal models** describe concepts through language. Both model media can be vague and open to multiple interpretations, giving an *illusion of understanding* without precise clarity (Smaldino, 2017). Furthermore, the last 100,000 years of evolution have shaped Homo sapiens in ways that make it difficult for us to comprehend a dynamic, unstable, and unpredictable world. Our *brains evolved to manage immediate, short-term situations* and *anticipate gradual, linear changes* with a tendency to seek *balance and stability* (Raworth, 2017). Thus, we must make a conscious effort to overcome these cognitive priors. Formal models can help with that.

Pictorial models enhance understanding through visual representations. Take, for example, maps, and diagrams, but also artistic paintings. **Mathematical models** use equations to quantify relationships, providing greater precision. **Computer models** require the highest level of precision; all entities and causal mechanisms must be defined unambiguously to allow a computer simulation to operate. This high level of precision makes them essential for scientific research and understanding. However, our subconscious mental models often have the highest impact on how we perceive and act in the world.

Thus, we will develop primarily formal mathematical and computational models in this book. However, this will automatically refine our mental and verbal models.

The challenge of informal models

Proverbs are a good example of mental models that are passed down through the generations. They concisely convey wisdom. However, they are often **contradictory** and provide no guidance on how to act. For example, consider the following proverbs (Page, 2018):

Proverb: Tie yourself to the mast Opposite: Keep your options open

Proverb: The perfect is the enemy of the good Opposite: Do it well or not at all

Proverb: Actions speak louder than words Opposite: The pen is mightier than the sword

The power of informal models

To illustrate the power of mental models, consider the following riddle:

> A father and son are in a horrible car crash that kills the dad. The son is rushed to the
> hospital; just as he's about to go under the knife, the surgeon says, "I can't operate – that
> boy is my son!"   *How can this be?*

Regardless of how obvious (or not) the answer appears to you, watch this video in which people in Vienna are asked this question (Autotranslation helps if you do not speak German). **Observe their reactions when their mental models are updated** (from minute 1:38 on).

We observe that **modeling is an iterative process**. When recognizing a mismatch between our models and reality, we get the opportunity to refine our models, and so gradually, we might become **less wrong** (Smaldino, 2017). Creating formal models of the systems we care about is the only method to achieve this in a structured, deliberate, and controlled manner.

However, as it is the defining feature of a model to simplify or, in other words, to be wrong, making them *more true* cannot be the purpose of a model per se. But, **what makes a model useful?**

1.2.3 Some models are useful

There is no universally agreed-upon classification of model use cases. I tend to distinguish between four clusters of model use cases (Figure 1.3): 1) Understanding & explaining, 2) Communication & learning, 3) Prediction & forecasting, and 4) Decision-making & action.
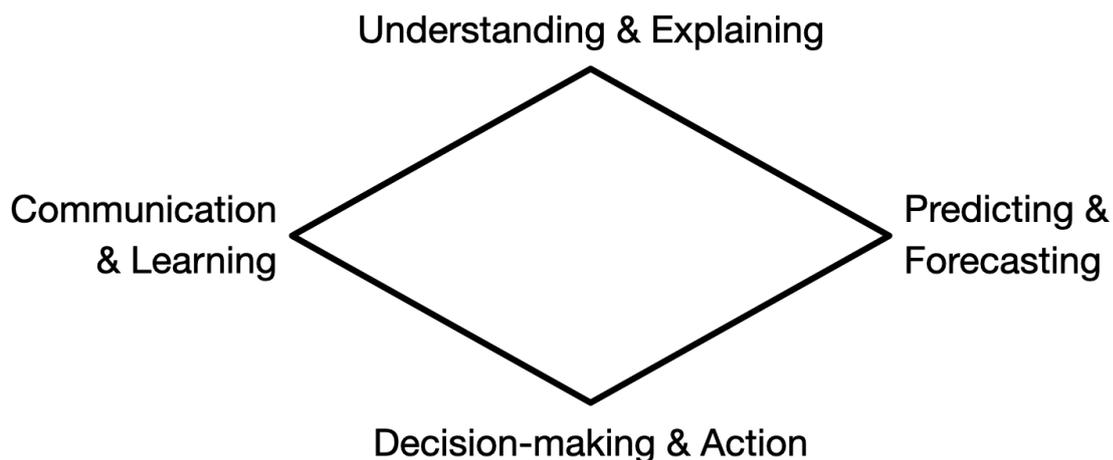
Figure 1.3: Possible model use cases

Understanding & Explaining

Understanding and explaining phenomena may occur in various ways. For example, models help **clarify assumptions** (Smaldino, 2017), allowing for a more transparent assessment of their implications and conclusions. Models help us to **reason**, i.e., to identify conditions and deduce logical implications. They also can provide (testable) **explanations** for empirical phenomena. (Page, 2018) And, models are helpful to **explore**, i.e., to consider different "what if" scenarios to investigate possibilities and hypotheticals (Page, 2018; Smaldino, 2017).

Communication & Learning

Formal models can serve as tools to **overcome our cognitive limitations**. They help in **systematizing** and **synchronizing** our understanding, ensuring that we discuss the same concepts and avoiding ambiguity that often accompanies verbal models.(Smaldino, 2017) Models can **guide scientific questions**. The precise specification of components and their relationships in a model helps clarify scientific questions and distinguishes them from unfalsifiable pseudo-theories (Smaldino, 2017).

Prediction & Forcasting

Prediction refers to making numerical and categorical predictions of future and unknown phenomena. Historically, explanation and prediction were often linked closely together. However, **prediction differs from explanation**. A model can predict without explaining. Deep learning algorithms can predict product sales, tomorrow's weather, price trends, and specific health outcomes; however, they provide minimal explanation. Also, a model can explain without predicting. Ecology models can explain speciation patterns but cannot predict new species (Page, 2018). Related concepts to prediction are **forecasting** and **projections**, which can mean various things in different contexts.

Decicion-making & Action

Formal models help **design** institutions, policies, and rules by providing frameworks for contemplating the implications of choices. Combining this process with empirical data, formal models are helpful for **action, guiding policy choices** and strategic actions of governments, corporations, and nonprofits (Page, 2018). Likewise, good mental models are helpful for good actions in our everyday lives.

It is important to note that, in general, a single model does not fulfill all use cases.

Some models might do, like, for example, Newtonian mechanics. It explains the motion of objects, predicts their future positions, guides the design of machines, and is learned in schools worldwide. However, in most instances, this is not the case. A model might explain a phenomenon but cannot predict it or vice versa. A helpful model for decision-making might neither make accurate predictions nor explain the underlying mechanisms. Take macroeconomic models as an example.

Beyon being useful or not, are there some quality criteria a *good* model should fulfill?

1.2.4 Some models are good

I argue that there are some quality criteria that make a model a *good* model. A good model must be

- coherent
- transparent, and
- sparse.

Coherence

Coherence means that the model is consistent. It does not contain contradictions or logical errors. Consider, for example, the proverbs from above.

A model can resolve these contradictions by specifying the conditions under which a particular statement holds. For example, under some conditions, it is best to *tie yourself to the mast*, while under others, it is best to *keep your options open*. A model can help to clarify these conditions.

This requirement of coherence or consistency **imposes a set of helpful constraints** within which the model development can take place.

Transparency

A good model makes its **assumptions explicit** and transparent. It also openly discusses its **limitations**. Bonus points if the model is transparent about **uncertainty** in empirical data, parameters, and processes. Transparency is a prerequisite for a model to be useful for communication and learning.

Sparsity

You should take the simpler model if you have two competing models of equal quality regarding their use case. This is also known under the term Occam's razor or the **principle of parsimony**. This principle helps us avoid going overboard with introducing new assumptions, entities, and processes into our models, making our models overly complex and difficult to understand without further benefit. However, sometimes, during the modeling process, we are unsure whether a newly introduced assumption is helpful to explain the phenomenon under question. Thus, for model development, the principle of parsimony is a guideline, not a strict rule. As the famous saying goes

> You should make things as simple as possible, but not simpler.

1.3 Systems reductionism

Given that we cannot not model, how should we make sense of the world?

1.3.1 Classical reductionism

With *classical reductionism*, I refer to the ideas of rationalism and empiricism that have dominated Western science since the last great transformation, the Enlightenment. Replacing religious dogmatism, this view argues, that

> **the whole can be understood from its parts**.

For example, this approach has been highly successful in physics and chemistry, where the behavior of atoms and molecules can be understood by studying their interactions.

As a result, scientific disciplines tend to be hierarchically clustered around specific parts of the whole. For example, the German Research Association (DFG) clusters disciplines around the engineering, life, natural, and humanities and social sciences (Figure 1.4). Within each cluster, there are multiple disciplines with subdisciplines.

Classical reductionism produced a lot of experts. Together, they drove the massive increase in wealth, health, and knowledge in the last 200 years.
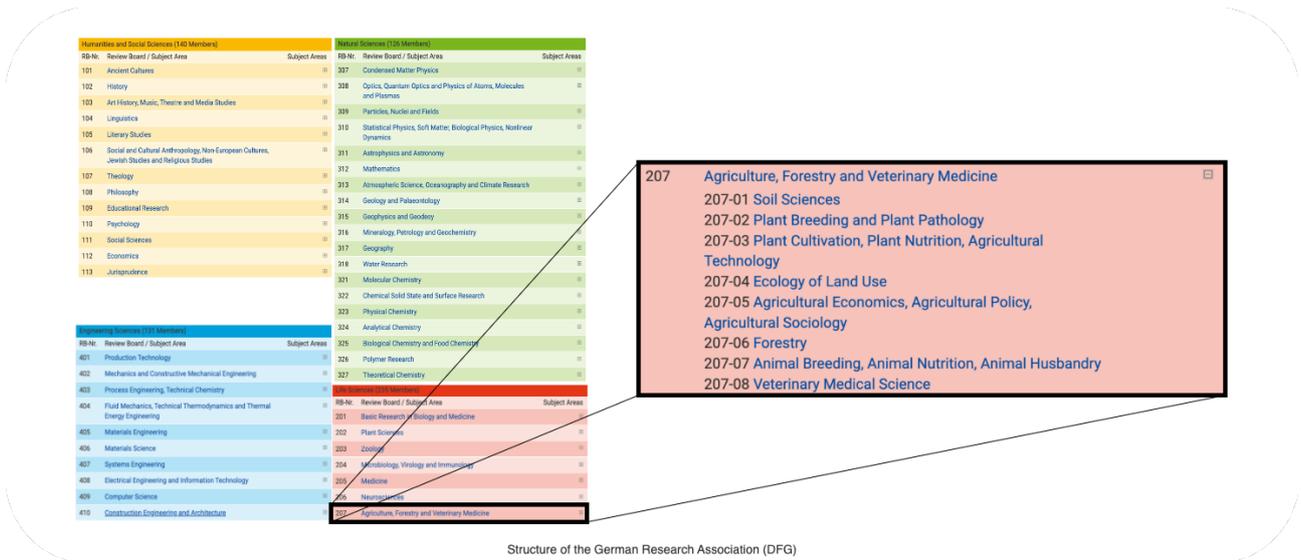
Figure 1.4: DFG classification of scientific disciplines

1.3.2 The problem with experts

Experts carry a risk of overrating the importance of their area of expertise (Brockmann, 2021). At the same time, experts tend to overlook the interactions within and beyond the system under investigation (Figure 1.5).

adapted from Brockmann (2021) Im Wald vor lauter Bäumen - Unsere komplexe Welt besser verstehen

Figure 1.5: The problem with experts

These problems with classical experts become particularly problematic in **complex systems**, which are characterized by their **interactions**.

1.3.3 Complex systems

The study of complex systems started around the 1950s and has been a diverse endeavor since then. See, for example, the map of complexity science.

In a complex system,

the whole is more than the sum of its parts.

The whole, the so-called macro-level, **emerges** from and **feeds backs** to the so-called micro-level, in which (often many, heterogeneous) entities or agents **interact** (in often non-linear ways) in a shared environment. Both levels are out-of-equilibrium, continuously evolving (Figure 1.6).
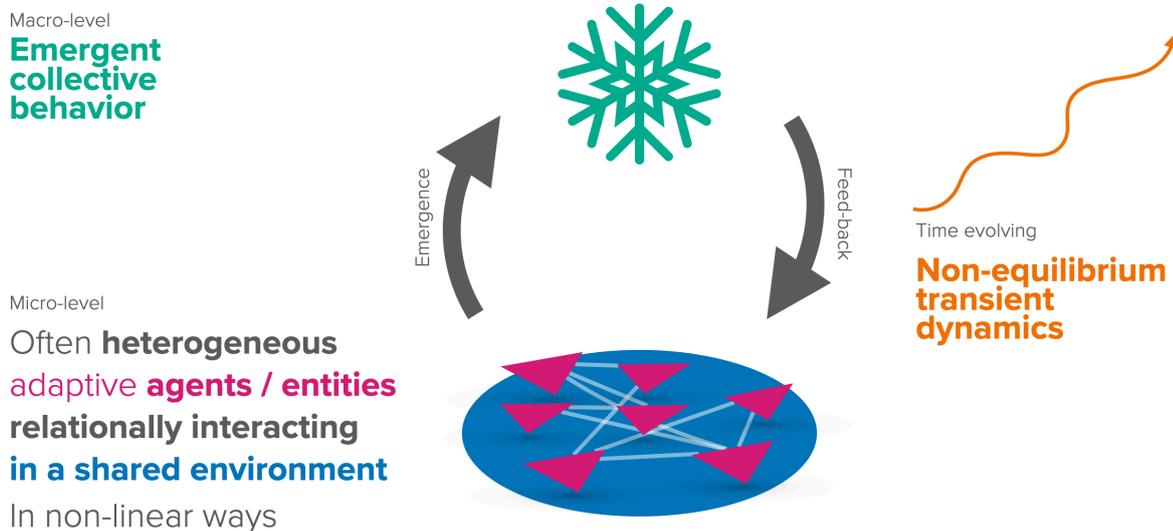
Figure 1.6: Properties of a complex system

To illustrate the idea of emergence, where the whole is more than the sum of its parts, consider the following quote:

> "There's no love in a carbon atom, No hurricane in a water molecule, No financial collapse in a dollar bill." – Peter Dodds

To observe a complex system in action, enjoy a video of bird flocking behavior. A century ago, the wonders of these highly coordinated yet leaderless flocks led people to believe that telepathy might be what guided these birds (phys.org).

See complexityexplained.github.io for more background information on complexity science.

So, what can we do to make sense of complex systems, given our limited information processing capacity and the consequences that we cannot not model?

1.3.4 Systems reductionism

To quote one of the founding fathers of complexity science,

> "It may not be entirely vain, however, to search for common properties among diverse kinds of complex systems" – Herbert Simon

As it turns out, flocking behavior, for example, can be explained by just three rules: separation, alignment, and cohesion (wikipedia.org/boids, Figure 1.7).

Systems and classic reductionism complement each other (Figure 1.8). While classical reductionism helps understand the parts of a system, systems reductionism helps understand the interactions between these parts. **Collaboration between the two approaches is key.**

Conceptually, **complex systems modeling combines the practice of (formal) modeling with the ideas of systems reductionism**.

COHESION

If agents are within the field of perception, steer towards the center of all neighbors.

SEPARATION

Agents that are too close to each other will try to maintain a safe distance from each other.

ALIGNMENT

If agents approach each other within their field of vision, they will align their trajectories.

Figure 1.7: The three rules to produce flocking behavior

Reality

Classic reductionism
Study the parts in detail

Systems reductionism
Study universal structures
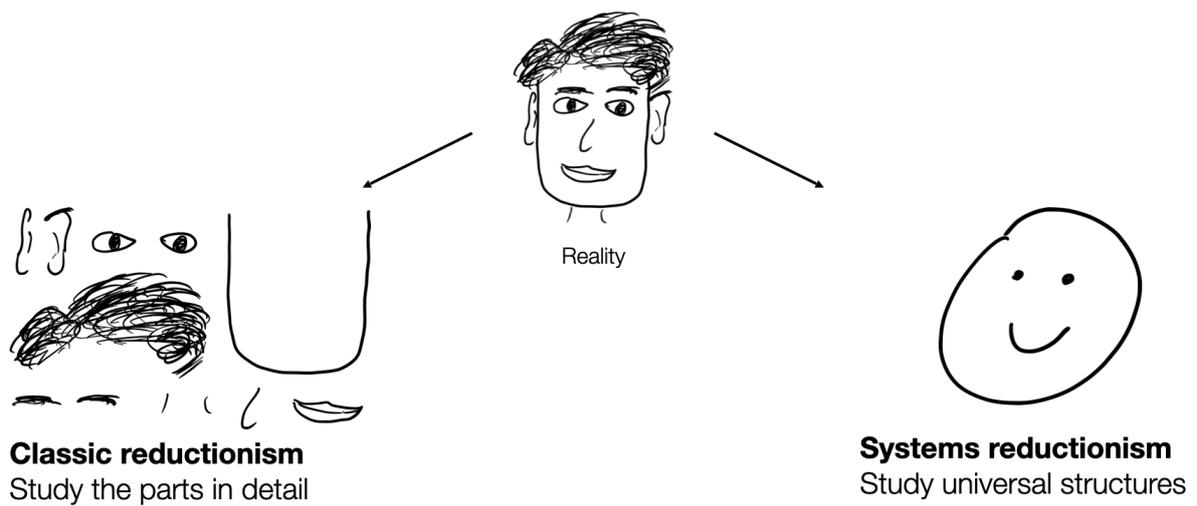
Figure 1.8: Systems and classic reductionism complementing each other

1.4 Sustainability Systems Modeling

This section conceptually synthesizes the practice of complex system modeling applied to the problem field of sustainability transitions and human-environment interactions.

1.4.1 Structural challenges

To operationalize systems thinking for human-environment modeling, we require a collection of the structural elements and processes that either hinder or may foster action toward sustainability.

Summarizing many fantastic review and perspective papers (Constantino et al., 2021; Elsawah et al., 2020; Farahbakhsh et al., 2022; Giupponi et al., 2022; Levin & Xepapadeas, 2021; Müller et al., 2020; Schill et al., 2019) we obtain the following list of structural challenges for sustainability transitions (Figure 1.9): *Complex system models of human-environment interactions* must account for the **dynamics** of the **collective behavior** emerging from **cognitive agents** within an **environmental context** (Barfuss, Flack, et al., 2024). They must also adhere to the **good modeling practices** of *coherence*, *transparency*, and *sparsity*, as discussed above.
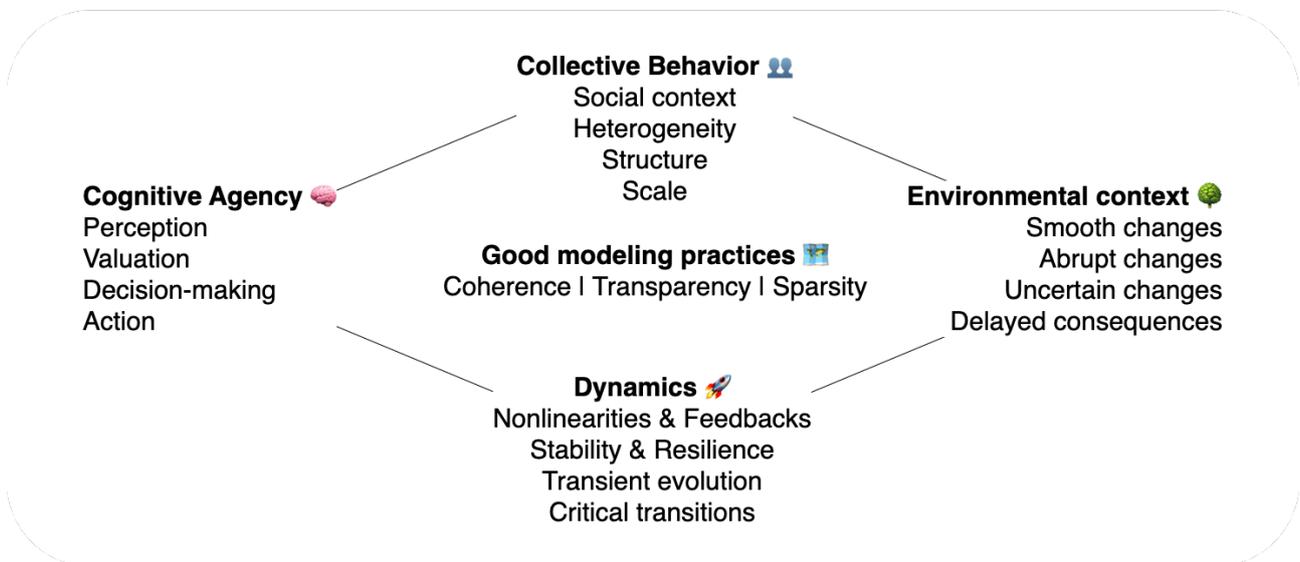
Figure 1.9: Structural challanges for sustainability transitions

Cognitive agency

Improving the representation of human behavior in models of social-ecological systems and human-environment interactions is a critical challenge (Constantino et al., 2021; Schill et al., 2019; Schlüter et al., 2017). Humans are neither hyper-rational nor overly simplistic, as many models assume. At the very least, they are cognitive agents who *perceive* their current environmental context, *evaluate* it and their options, *make decisions*, and *act* accordingly.

Environmental context

The environmental context refers to the decision-making challenge the agents face. The environmental context is not static. It may change *smoothly* or *abruptly*, based on human activities or via inherent dynamics. For example, climate damages gradually worsen with increasing global mean temperature. And crossing climate tipping points may abruptly lead to catastrophic outcomes. Furthermore, these

changes are *not certain* but stochastic in nature and may only be partially observable by humans. And often, the consequences of action are heavily delayed, impacting future generations. All these attributes make collective action for sustainability transitions tremendously challenging.

The environmental context includes the biophysical environment, such as the climate, biodiversity, and resources, as well as the social environment, out of which the collective behavior emerges.

Collective behavior

From cognitive agents within an environmental context, collective behavior emerges. This collective behavior depends on the *social context*, the *heterogeneity* of the agents, the *interaction structure* between them, and the scale on which they operate. Collective behavior refers to the dynamics of the system as a whole, which are not easily reducible to the characteristics of individual agents.

Dynamics

Our primary goal is to comprehend and advance sustainability transitions. Transitions are fundamentally dynamic in nature, so our modeling approach must reflect this dynamism, integrating *non-linear feedback loops* and *critical transitions*. Furthermore, the concepts of *stability* and *resilience* demand a dynamic viewpoint. Before a system reaches stability, its *transient evolution* offers crucial insights into the sustainability transitions itself.

How can we begin to make sense of this all?

What precisely do all of these elements mean?

And how do all of these elements relate to each other?

These questions will guide us through the following chapters.

We will tackle them with the help of a useful framework from transdisciplinary research: the **three types of knowledge**, applied to modeling.

1.4.2 Three types of models

When addressing societal challenges, the concept of the *three types of knowledge* helps to produce not only knowledge on problems but also knowledge that helps to overcome those problems (Buser & Schneider, 2021). In general, the concept applies to all research methodologies. We will specifically discuss it in the context of formal modeling, transforming it into *three types of models* (Figure 1.10).

The three types of models are:

Dynamic-systems models

Dynamic-systems models operationalize **systems knowledge**, typically understood as knowledge concerning the existing system or issue. This understanding is primarily analytical and descriptive. For instance, in the context of sustainability transitions, systems knowledge assesses the risk triggering climate tipping points, biodiversity loss dynamics, or a specific region's social-ecological dynamics. Systems knowledge is strongly associated with **facts** and asks **what is?**

In this regard, **dynamic-systems models** are often used to understand the system's behavior under specific conditions.

We will discuss dynamic-systems modeling in the first part of these lecture notes, covering
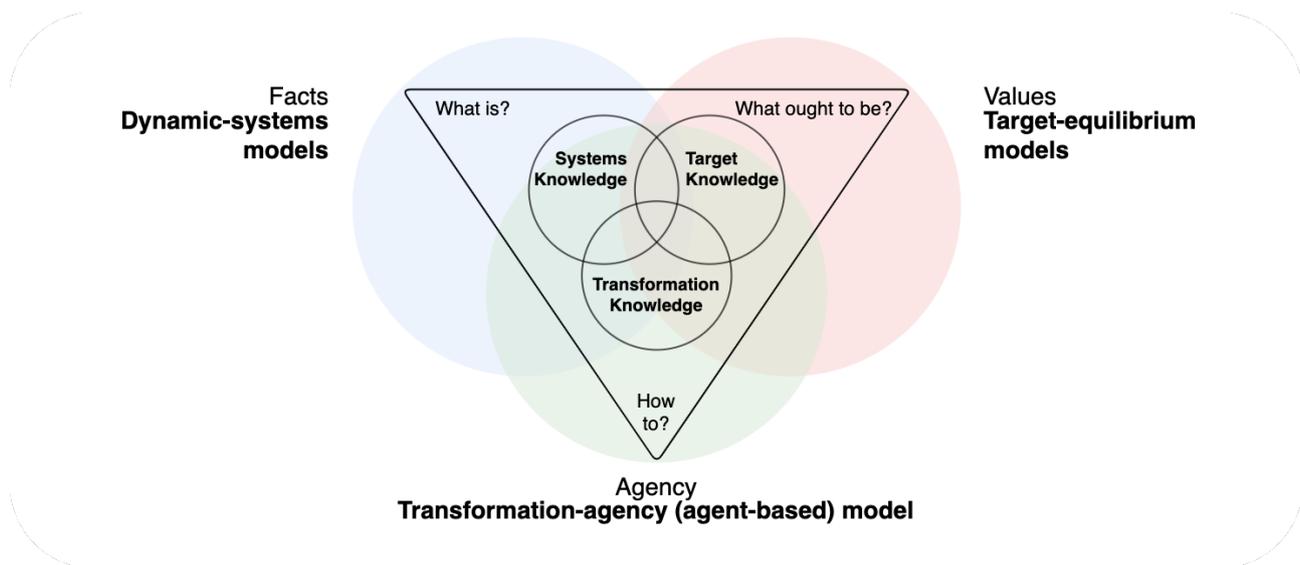
Figure 1.10: Three types of models based on three types of knowledge for transdisciplinary reserach

- Nonlinearity and feedback loops in Chapter 02.01
- Tipping elements and regime shifts in Chapter 02.02
- Resilience in Chapter 02.03, and
- Stochastic state transitions in Chapter 02.04.

Target-equilibrium models

Target-equilibrium models operationalize **target knowledge**, which is knowledge about the desired future and the values that indicate which direction to take. It relies on deliberation by different societal actors and is based on values and norms. In sustainability transitions, ways of producing target knowledge include participatory vision, scenario development with a wide range of stakeholders, and the public discourse at large. Target knowledge is strongly associated with **values** and asks **what ought to be?**.

Target-equilibrium (or *equilibrium-based models applied to sustainability transitions*) are primarily used in economics. In theory, the equilibrium is the outcome of an optimization procedure where the specified normative target is reached.

We will discuss target-equilibrium modeling in the second part of the book, covering

- Sequential decisions of a single agent in a dynamic environment in Chapter 03.01
- Strategic interactions of multiple agents in a static environment in Chapter 03.02, and
- Strategic interactions of multiple agents in a dynamic environment in Chapter 03.03.

Transformation-agency models

Transformation-agency models operationalize **transformation knowledge**, which is knowledge about how to move from the existing system to the desired future. This knowledge includes concrete strategies and steps to take. In sustainability transitions, producing transformation knowledge could involve developing policy instruments, designing new institutions, or implementing new technologies. Transformation knowledge is strongly associated with **agency** and asks **how to?**.

Transformation-agency models (or *agent-based models applied to sustainability transitions*) are a flexible tool that *combines* the *dynamics* of how to get to a desired outcome with *agency* that defines what is desirable and possible to do.

We will discuss transformation-agency modeling in the third part of the book, covering

- Rule-based behavioral agency in agent-based models in Chapter 04.01
- Individual reinforcement learning in Chapter 04.02, and
- The non-linear dynamics of reinforcement learning in Chapter 04.03.

It is important to note that the three **knowledge types are interdependent**. For example, knowledge about 'how to' would be of limited use or even dangerous if it was not oriented toward desirable target values and based on sound facts. In the same vein, we will integrate the different types of models toward the end of this book.

1.5 Learning goals revisited

- Understanding and promoting **sustainability transitions** requires a **coupled human-environment systems approach**, as the **challenges** and possible **solutions** are **tightly coupled** between humans and the biosphere.
- Limited human information processing demands us to model the world around us. **Formal models help overcome imprecise mental models and cognitive limitations**. Models are **helpful** for *understanding*, *communicating*, *predicting*, and *making decisions*. **Good models** are *coherent*, *transparent*, and *sparse*.
- **Systems reductionism complements classical reductionism** to **avoid unintended side effects** in complex systems. Complex systems are characterized by **interactions, emergent properties, and feedback loops**.
- Complex systems models of human-environment interactions must account for the **dynamics** of the **collective behavior** emerging from **cognitive agents** in **environmental contexts**. Three types of models, *dynamic-systems model*, *target-equilibria models*, and *transformation-agency* (agent-based) model, will help us achieve these desiderata.

The exercises for this chapter offer a thorough introduction to the programming language `Python`, preparing you for the modeling exercises in the subsequent chapters.

Part I

Dynamic Systems

In this part, we cover dynamic-systems models. They operationalize **systems knowledge**, typically understood as knowledge concerning the existing system or issue. This understanding is primarily analytical and descriptive. For instance, in the context of sustainability transitions, systems knowledge assesses the risk triggering climate tipping points, biodiversity loss dynamics, or a specific region's social-ecological dynamics. Systems knowledge is strongly associated with **facts** and asks **what is?**

In this regard, **dynamic-systems models** are often used to understand the system's behavior under specific conditions.
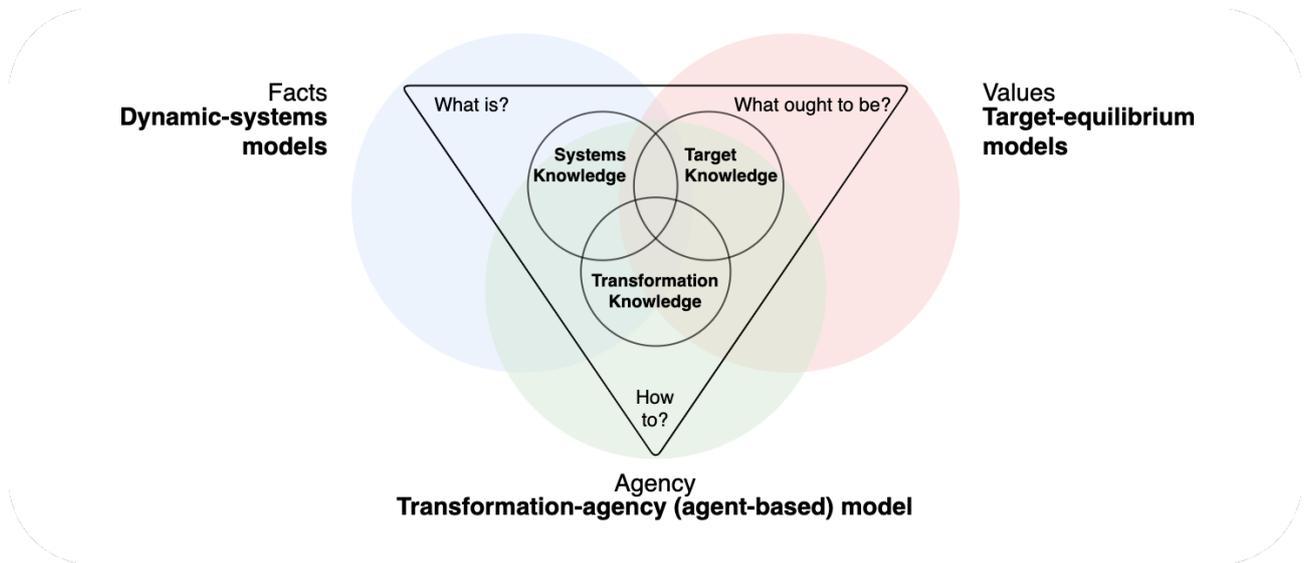
Figure 1.11: Three types of models based on three types of knowledge for transdisciplinary reserach

Specifically, we will cover

- Nonlinearity and feedback loops in Chapter 02.01
- Tipping elements and regime shifts in Chapter 02.02
- Resilience in Chapter 02.03, and
- Stochastic state transitions in Chapter 02.04.

2 Nonlinearity

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

2.1 Motivation

2.1.1 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 A: Global CO$_2$ emissions from human activity

Figure B: Atmospheric CO$_2$ concentration

Figure C: Average global surface temperature [°C]
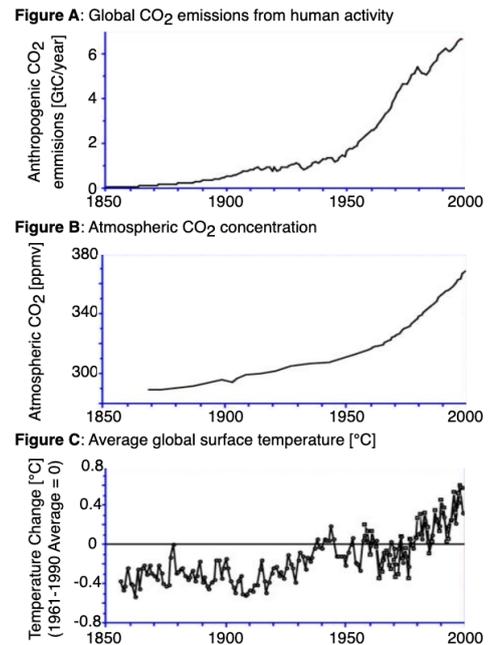
Figure 2.1: The issue of climate change

Now consider the following questions:

Consider a scenario in which the concentration of CO_2 in the atmosphere gradually rises to 400 ppm, about 8% higher than the level today, then stabilizes by the year 2100, as shown on the right.
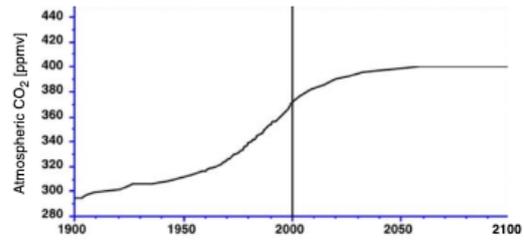
The graph on the right shows anthropogenic CO2 emissions from 1900-2000, and current net removal of CO2 from the atmosphere by natural processes. **Sketch**

1. Your estimate of likely future net CO2 removal, given the scenario above.
2. Your estimate of likely future anthropogenic CO2 emissions, given the scenario above.
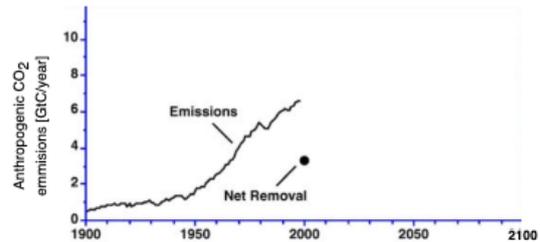
Adapted from Sterman & Sweeney (2007) Understanding public complacency about climate change: adults' mental models of climate change violate conservation of matter

Figure 2.2: A scenario of climate change

Typical responses. This experiment was conducted with MIT graduate students, a group of highly educated adults (Sterman & 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.
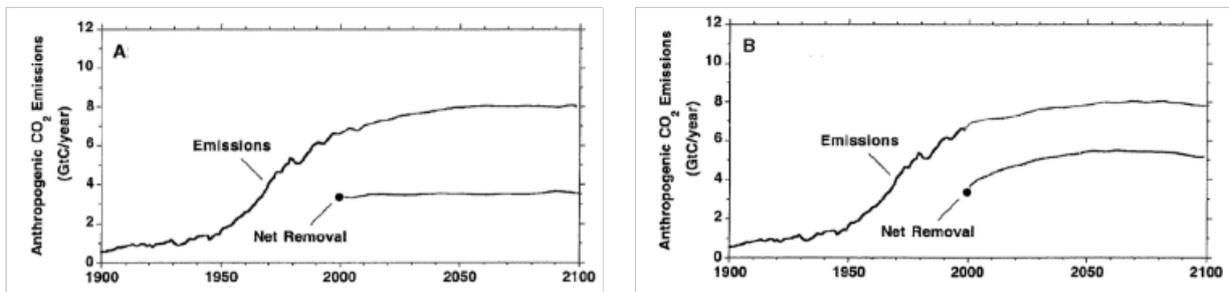
Adapted from Sterman & Sweeney (2007) Understanding public complacency about climate change: adults' mental models of climate change violate conservation of matter

Figure 2.3: Typical Responses

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

2.1.2 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 2.4: Carbon Bathtub

2.1.3 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.

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

2.2 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.

2.2.1 Pictorial representation

Graphically, we can compose (often called) stock-and-flow or causal (loop) diagrams via the following building blocks.
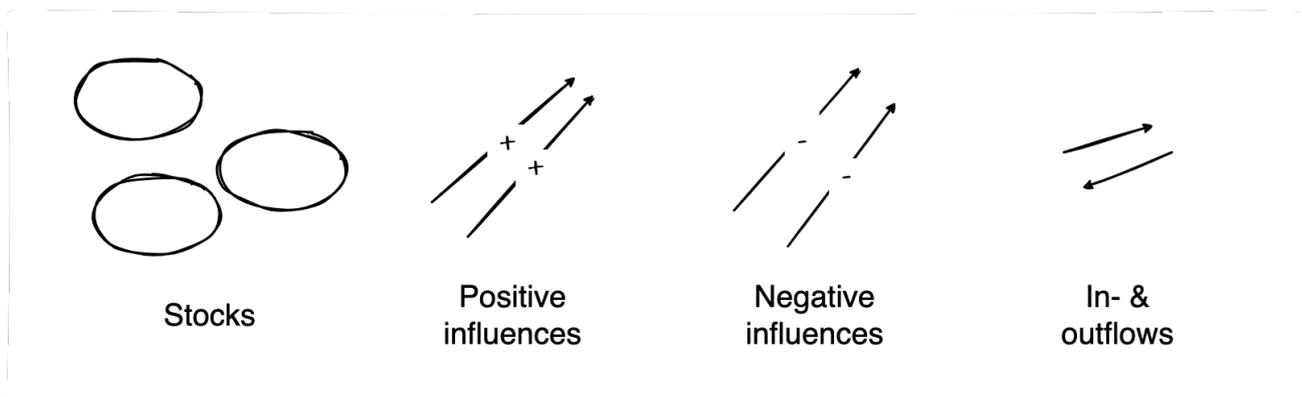
Figure 2.5: Graphical elements of a dynamic system

For instance, the carbon bathtub could resemble this:
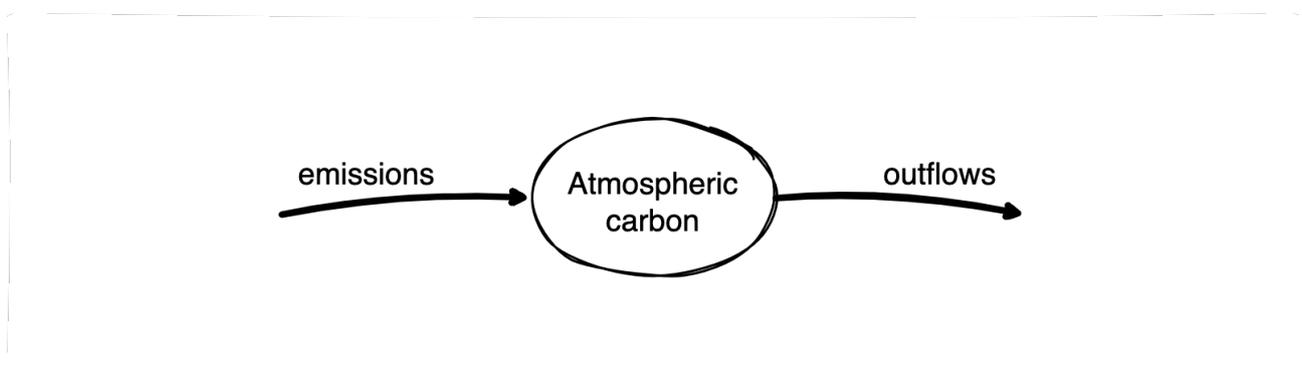
Figure 2.6: Graphical elements of a dynamic system

---

[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.

2.2.2 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.

2.2.3 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**.

```python
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).

```python
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.

```python
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]');
```
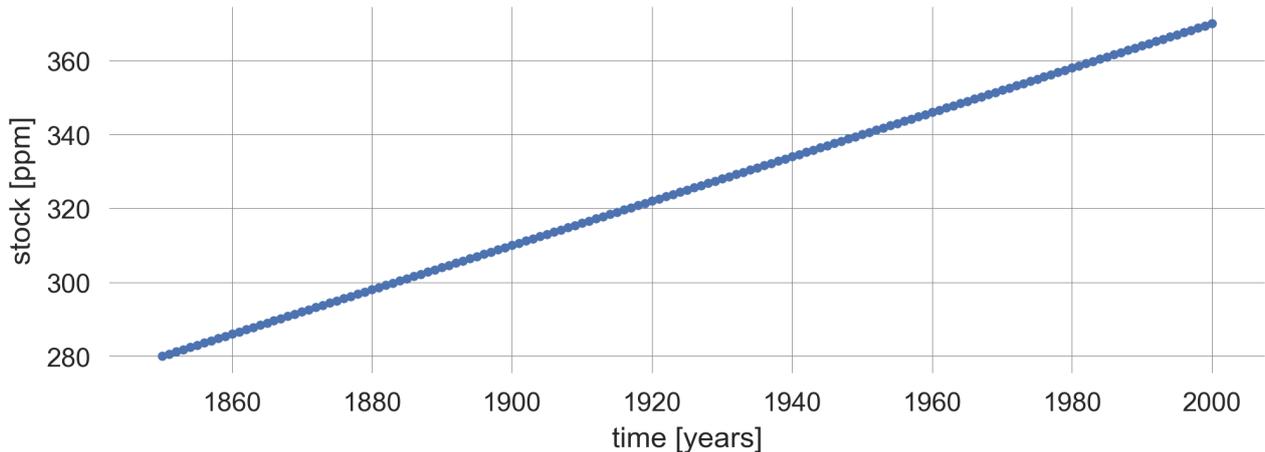

Figure 2.7: Linear growth

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.

2.2.4 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**.

2.3 Feedback

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

2.3.1 Positive feedback loops

Consider, for example, the following system, pictorially,
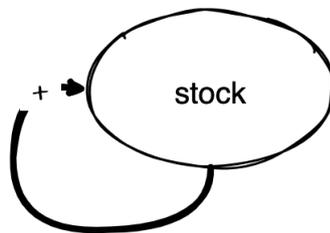
Figure 2.8: Positive Feedback Loop

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.

```python
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.

```python
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.

```python
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();
```
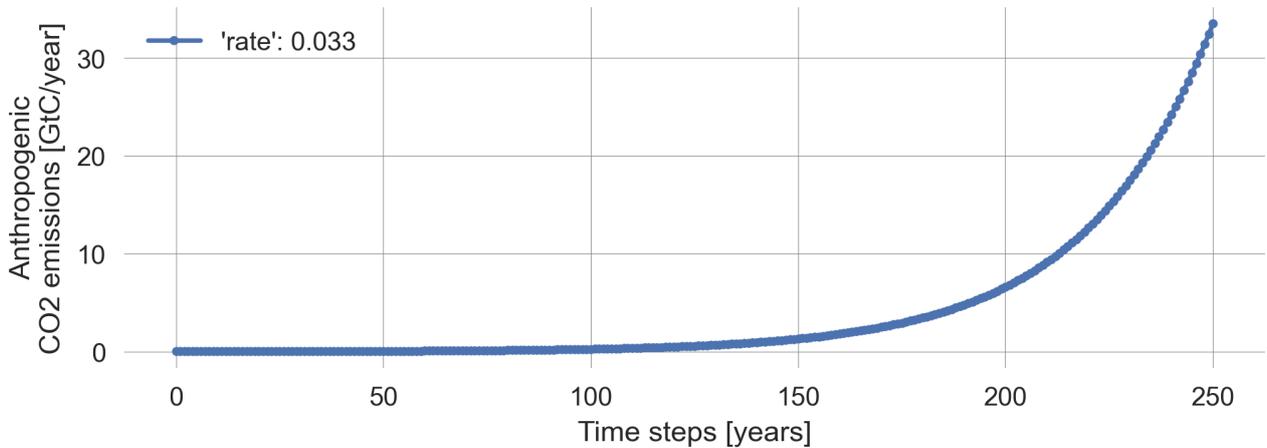

Figure 2.9: Exponential growth

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 2.7).

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 2.10).
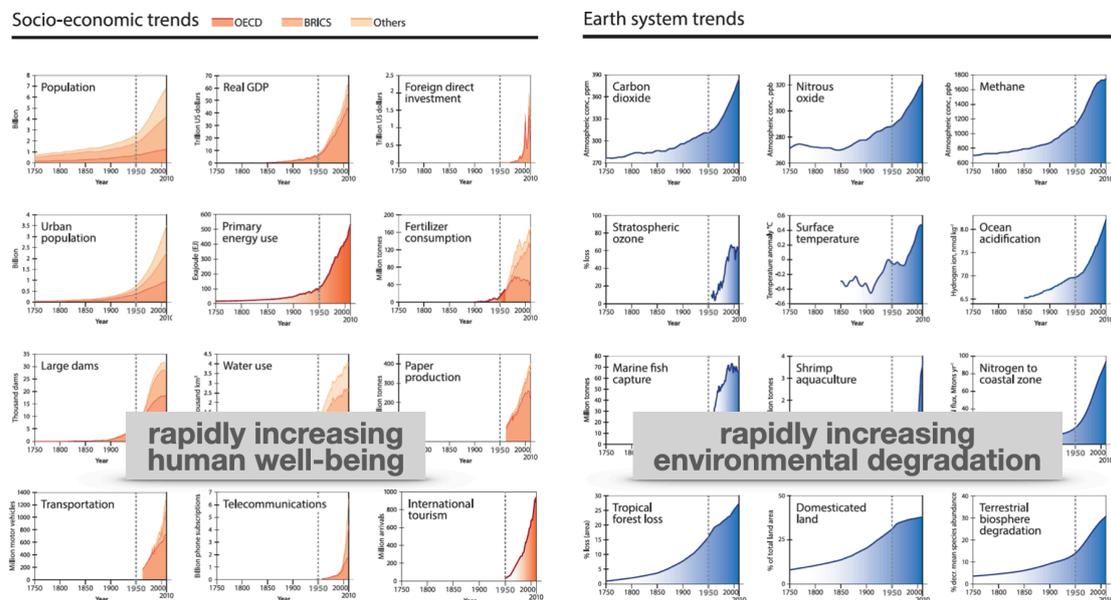
Figure 2.10: The great acceleration

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();
```
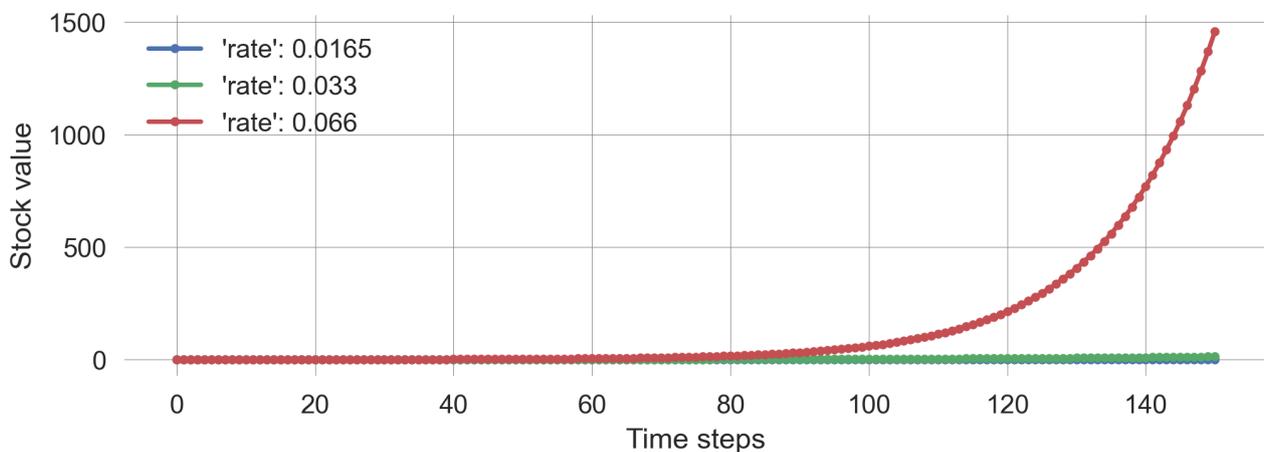

Thus, a halved rate leads to about a **10th of the stock** at the end:

```
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**.

2.3.2 Negative feedback loops

At first glance, negative feedback loops appear quite similar to positive ones.
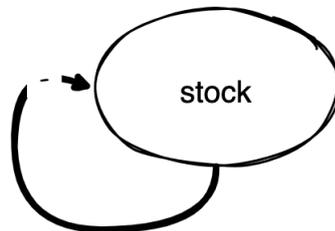
Figure 2.11: Negative Feedback Loop

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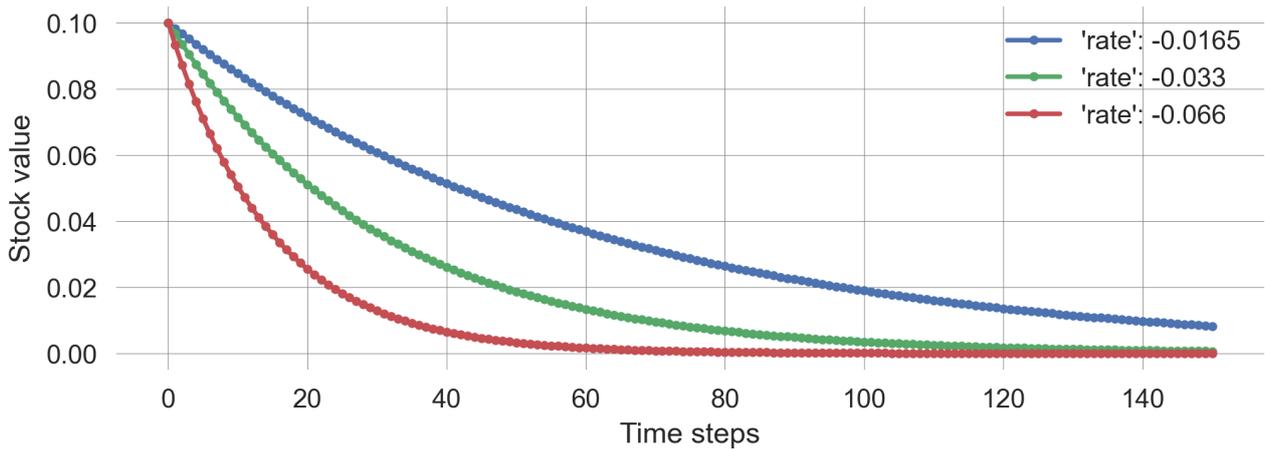
Thus, the stocks shrink **faster than linearly** toward zero, with more negative growth rates causing faster decay. This process is also called **exponential decay**.

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**.

2.4 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**.

2.4.1 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.
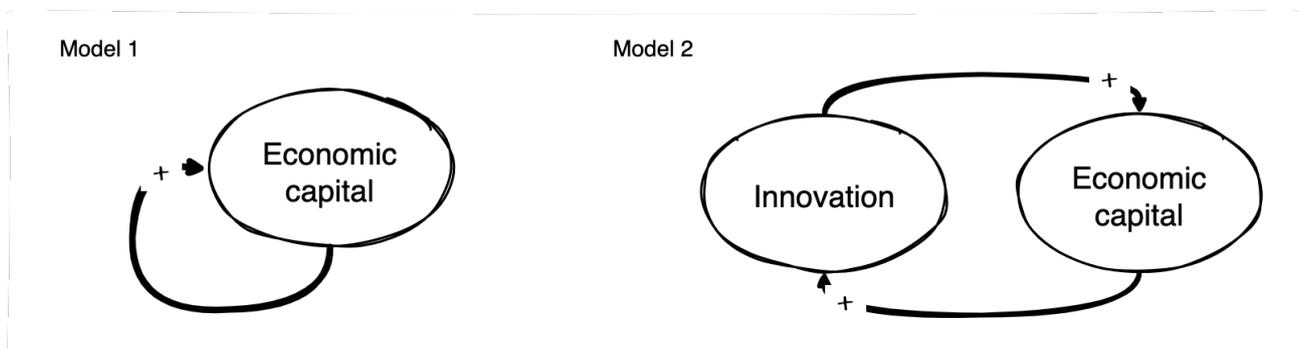
Figure 2.12: Two models of economic growth

35

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:

$$x_{t+1} = x_t + ay_t \tag{2.1}$$
$$y_{t+1} = y_t + bx_t \tag{2.2}$$

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.

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

```python
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.

```python
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
```

```python
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()
```
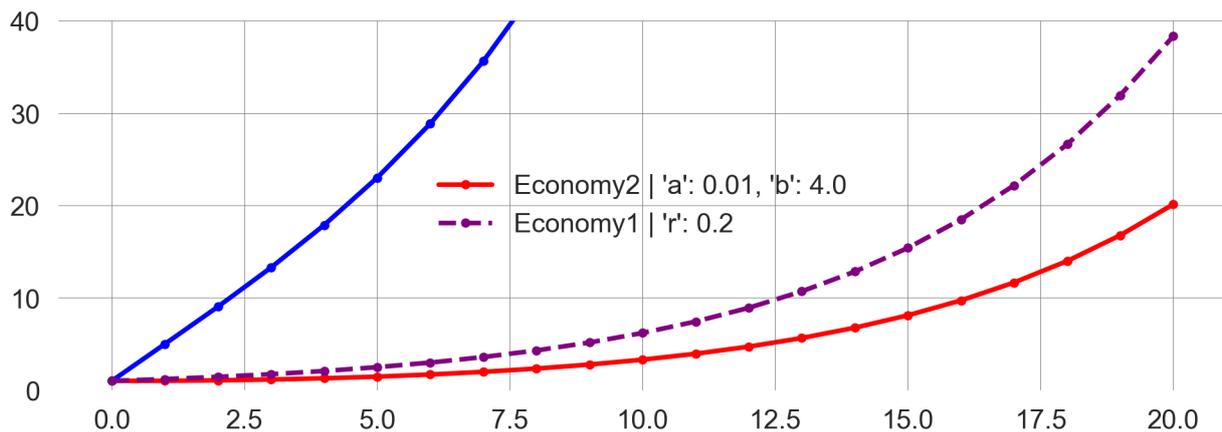

Figure 2.13: Default Economy-Innovation interactions

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 2.14 with Figure 2.13.

```
compare_models(innovation=20)
```
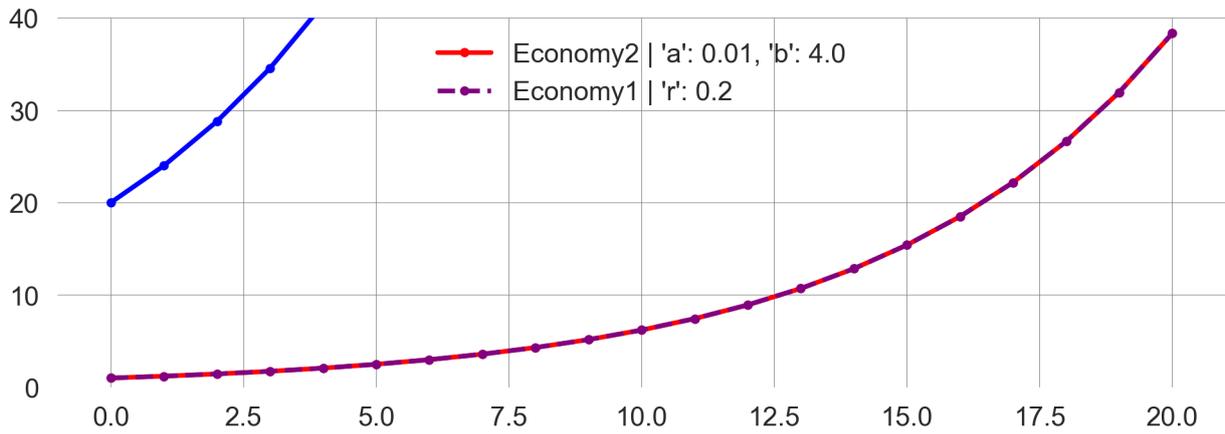
Figure 2.14: Economy-Innovation interactions with a 20-fold increase in innovations

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 2.15).

```python
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)
```
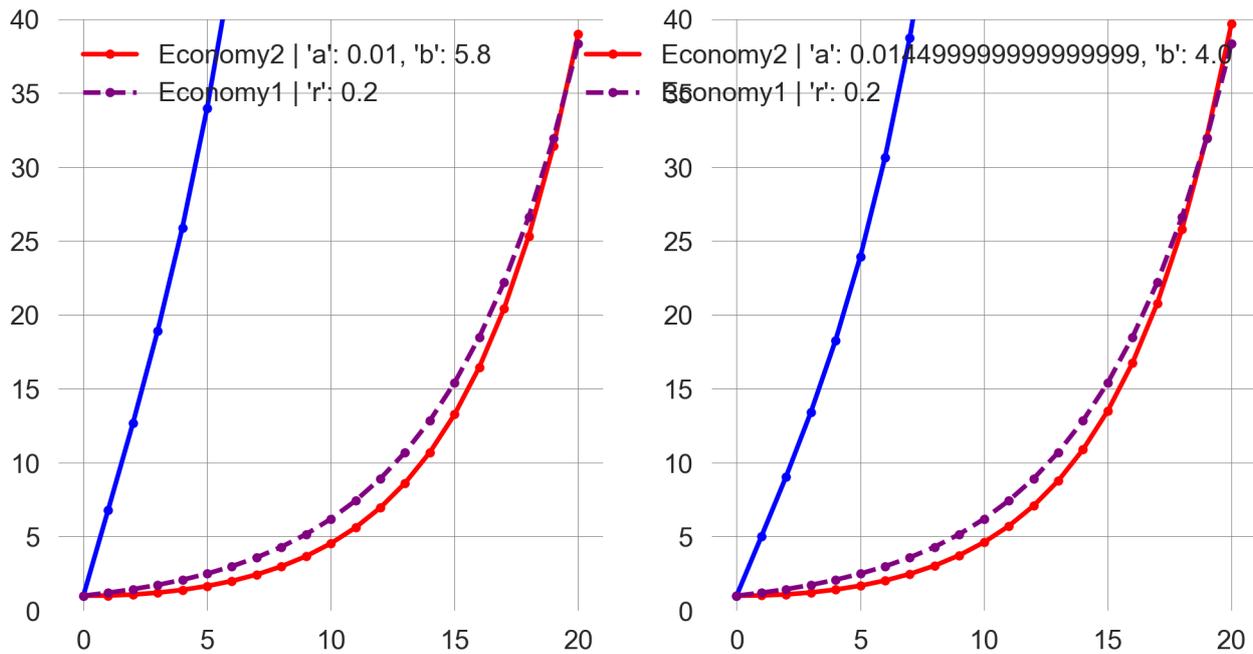

Figure 2.15: Economy-Innovation interactions with a 1.45-fold increase of a single conversion rate

38

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.

2.4.2 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.
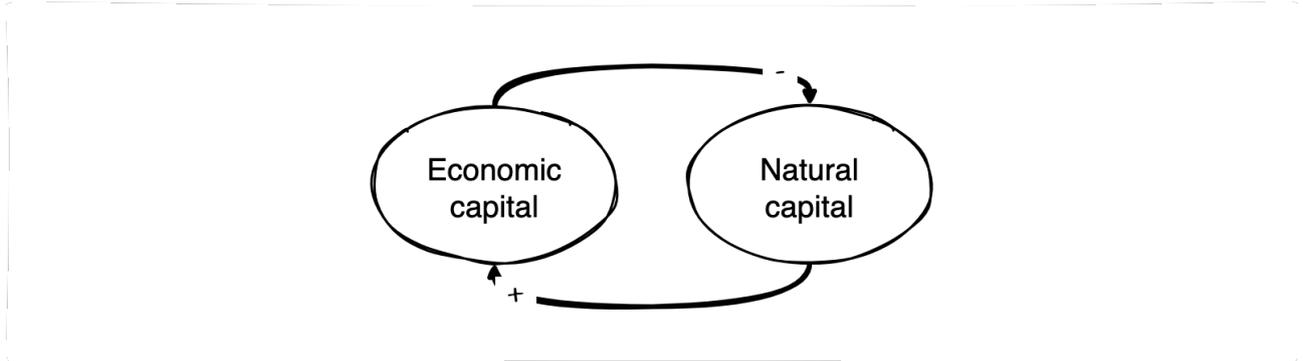
Figure 2.16: Economy-Nature interactions

Assuming the feedback structure defined in Figure 2.16, 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()
```
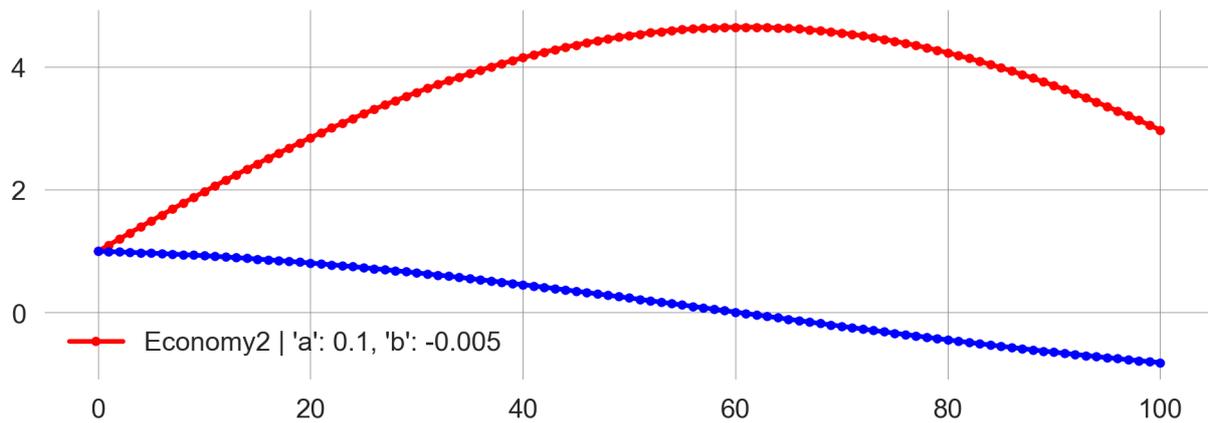
Figure 2.17

What happens if we continue the simulation for 1000 periods?

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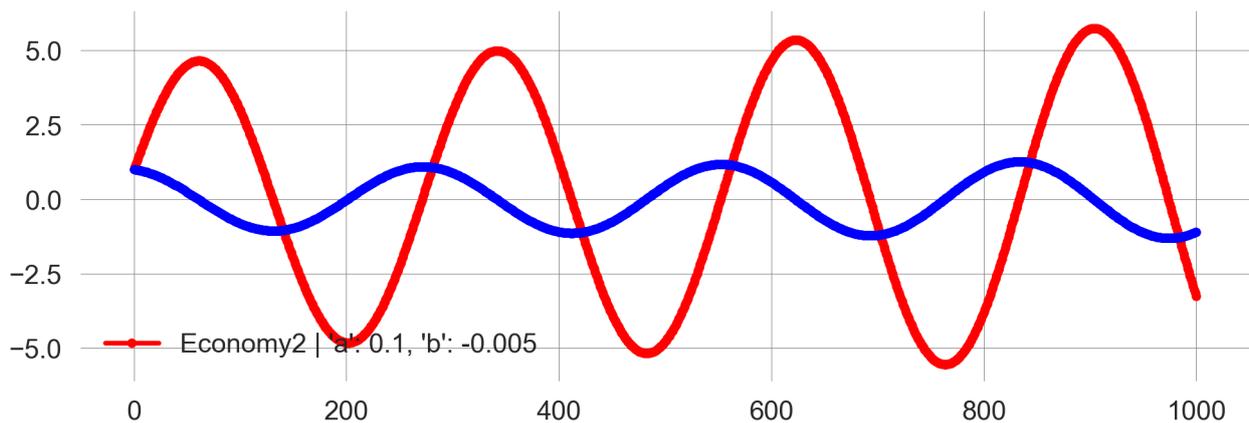

Figure 2.18

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.

2.4.3 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

$$x_t = x_{t-1} + y_{t-1} \tag{2.3}$$
$$y_t = x_{t-1} \tag{2.4}$$

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.

2.4.4 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,

$$x_{t+1} = ax_t + by_t, \tag{2.5}$$
$$y_{t+1} = dy_t + cx_t. \tag{2.6}$$

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 $i, j \in \{1, 2, ..., n\}$, 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,

$$x_{t+1}^1 = a_{11}x_t^1 + a_{12}x_t^2 + \cdots + a_{1n}x_t^n \tag{2.7}$$
$$x_{t+1}^2 = a_{21}x_t^1 + a_{22}x_t^2 + \cdots + a_{2n}x_t^n \tag{2.8}$$
$$\vdots \quad = \quad \vdots \qquad \cdots \qquad \vdots \tag{2.9}$$
$$x_{t+1}^n = a_{n1}x_t^1 + a_{n2}x_t^2 + \cdots + a_{nn}x_t^n. \tag{2.10}$$

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

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

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.

2.5 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**.

2.5.1 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 $x_2 = ax_1 = aax_0 = a^2 x_0$. Thus, generalizing this pattern yields the system state at time t to be

$$x_t = x_0 a^t$$

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.

```python
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()
```

```python
compare_solutions()
```
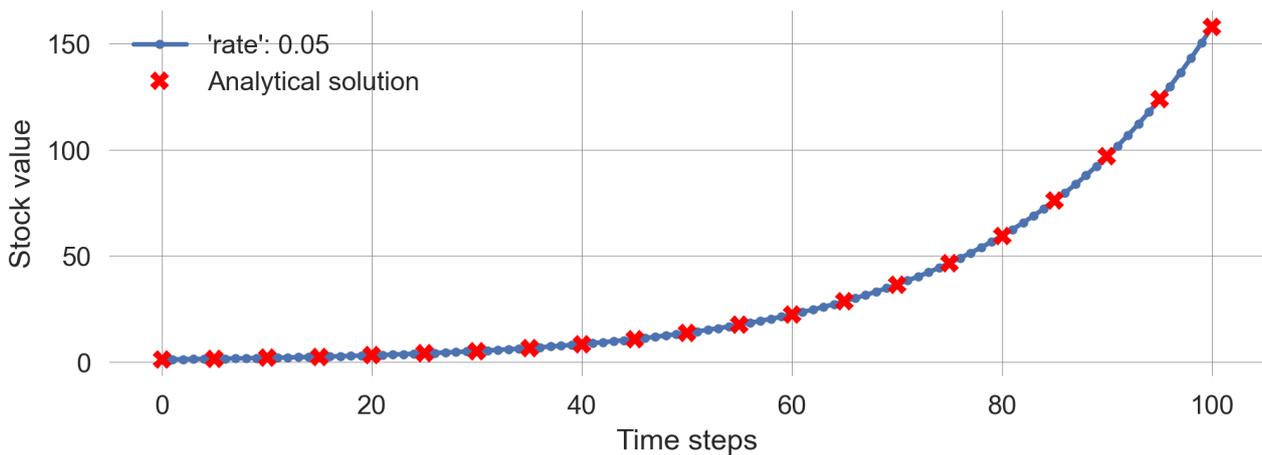

Try it out with different parameter values and observe that the closed-form solution matches the simulation results perfectly.

2.5.2 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.

```python
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 $x_{t+1} = ax_t$.

```python
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 2.19.

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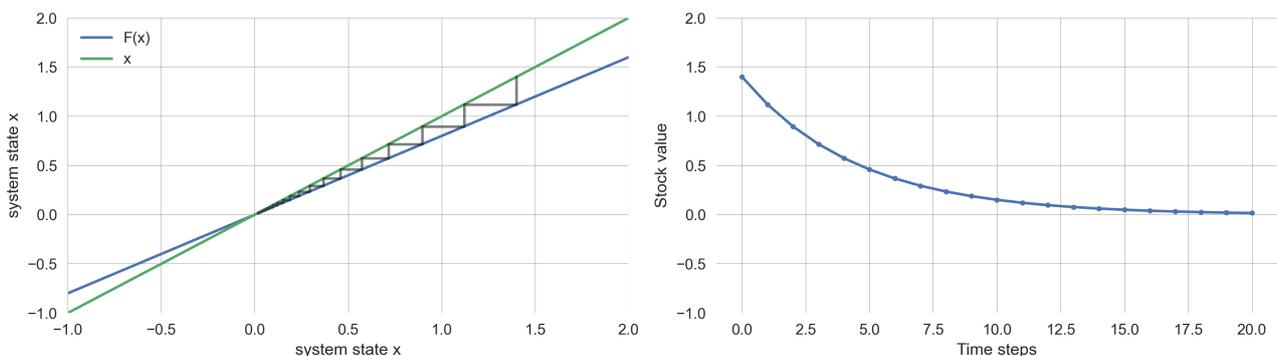

Figure 2.19

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?

2.5.3 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)
```
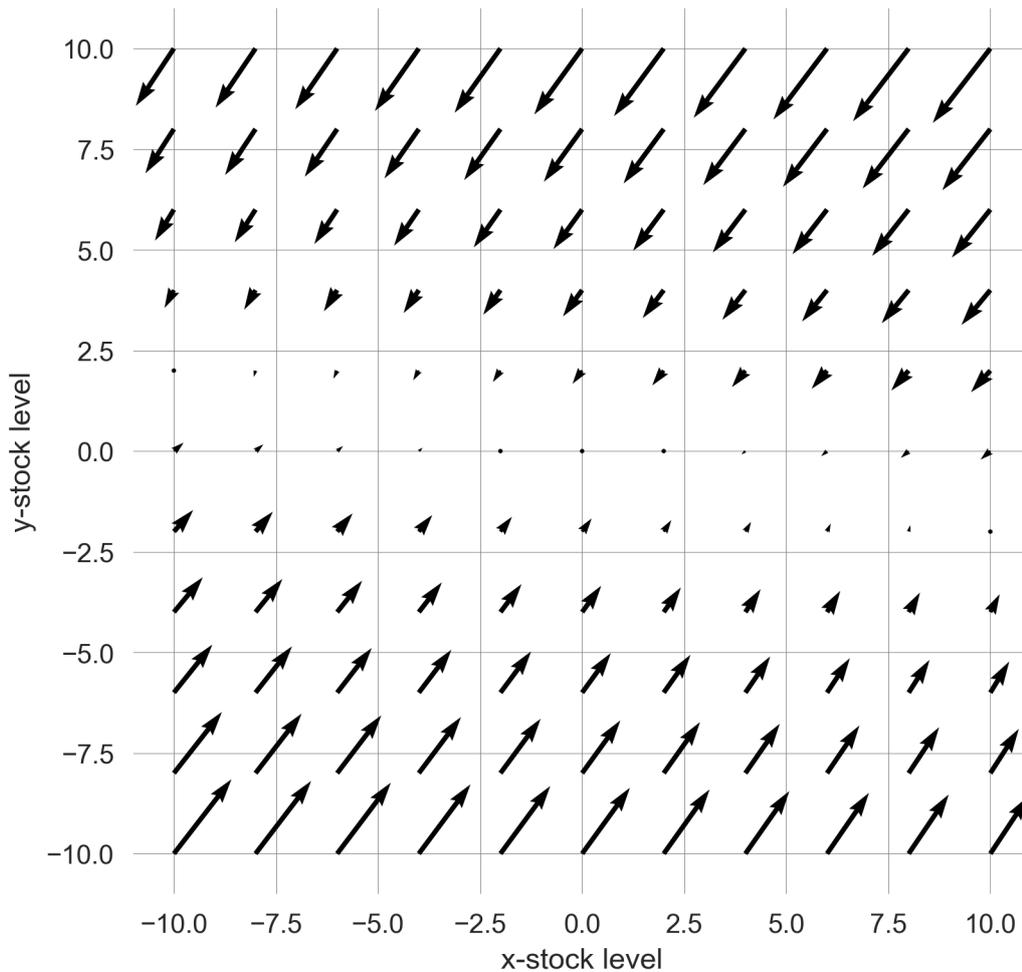
Now, we can visualize the flow of any two-dimensional system, including a trajectory, in the phase space.

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.

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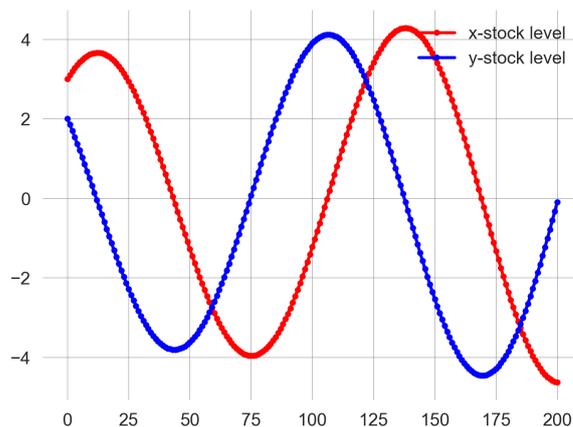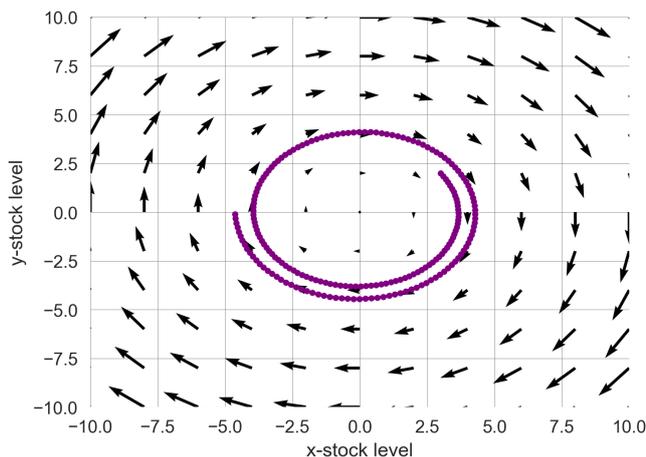
Such a phase space visulaization is a powerful tool, connecting the time-evolution of a dynamic systems with a **geometrical representation**.

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

2.5.4 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

$$\mathbf{x}_k = \mathbf{A}^k \mathbf{x}_0 \tag{2.11}$$
$$= \mathbf{A}^k (c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n) \tag{2.12}$$
$$= c_1 \mathbf{A}^k \mathbf{v}_1 + c_2 \mathbf{A}^k \mathbf{v}_2 + \cdots + c_n \mathbf{A}^k \mathbf{v}_n \tag{2.13}$$
$$= c_1 \lambda_1^k \mathbf{v}_1 + c_2 \lambda_2^k \mathbf{v}_2 + \cdots + c_n \lambda_n^k \mathbf{v}_n \tag{2.14}$$
$$\tag{2.15}$$

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)
```
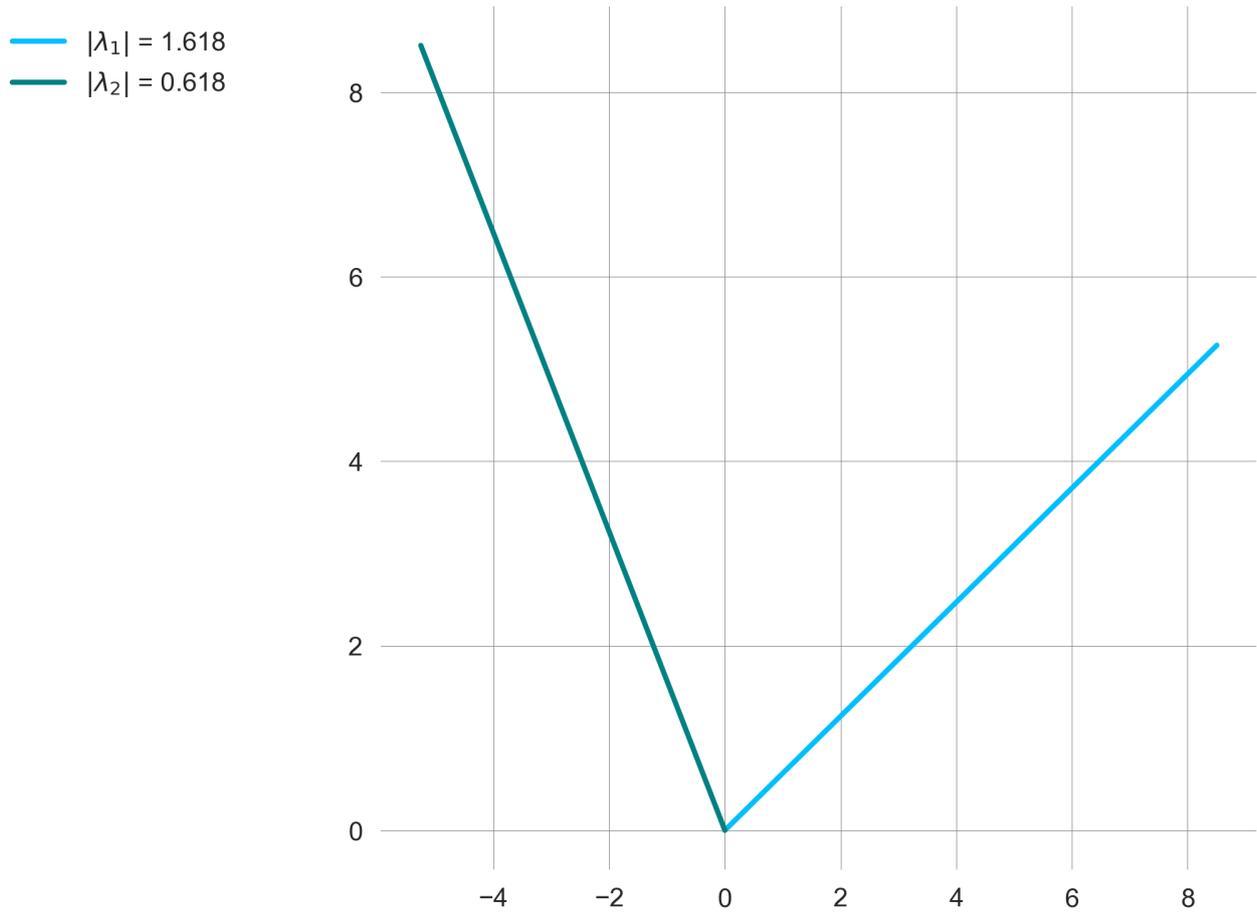
Putting it all togehter, we observe how the eigenvector represent the long-term behavior of the system (Figure 2.20).

```
plot_flow_trajectory_with_ev()
```
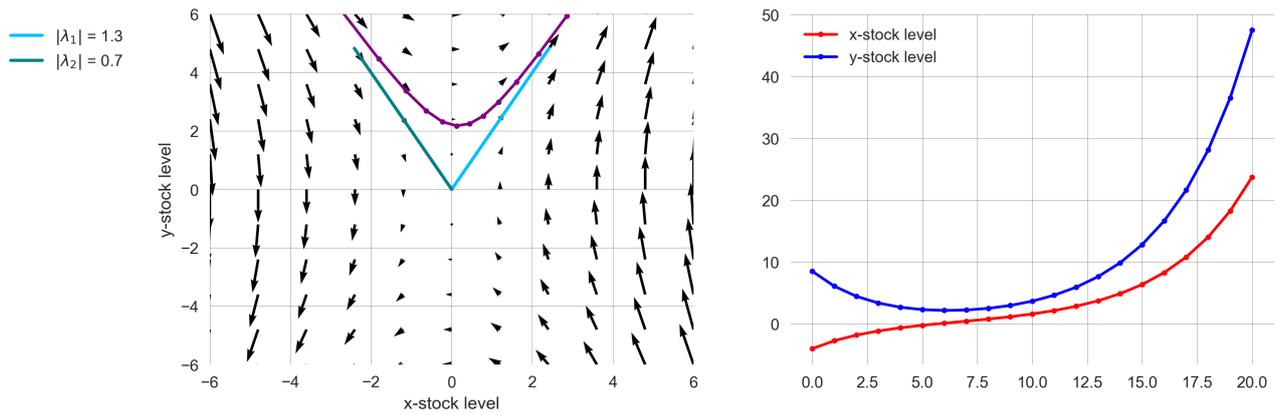
Figure 2.20

2.5.5 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.

2.6 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:

```python
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);
```
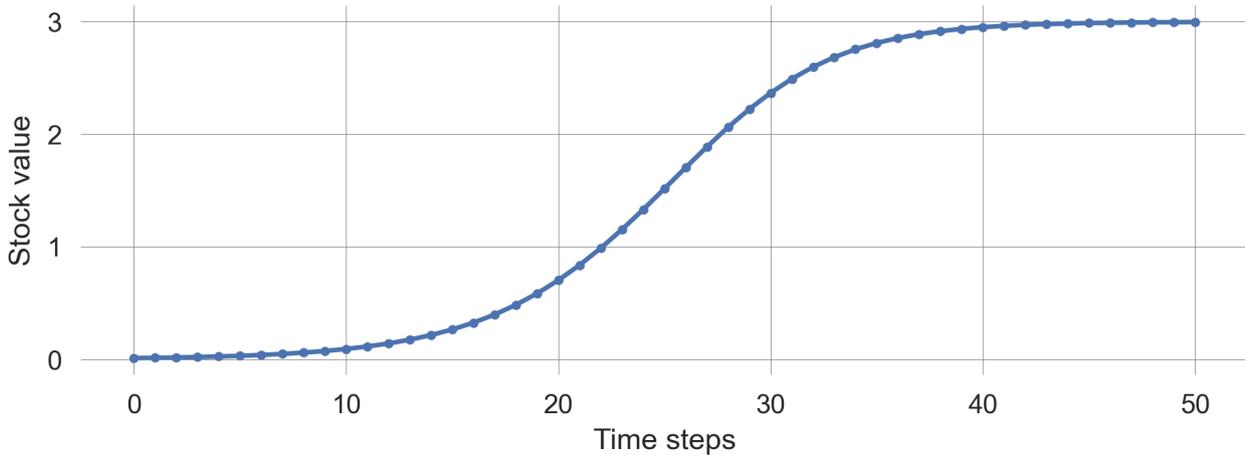

2.6.1 Finding equilibrium points

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,

$$x_e = x_e + rx_e(1 - \frac{x_e}{C}) \tag{2.16}$$

$$0 = 0 + rx_e(1 - \frac{x_e}{C}) \tag{2.17}$$

$$0 = rx_e(1 - \frac{x_e}{C}) \tag{2.18}$$

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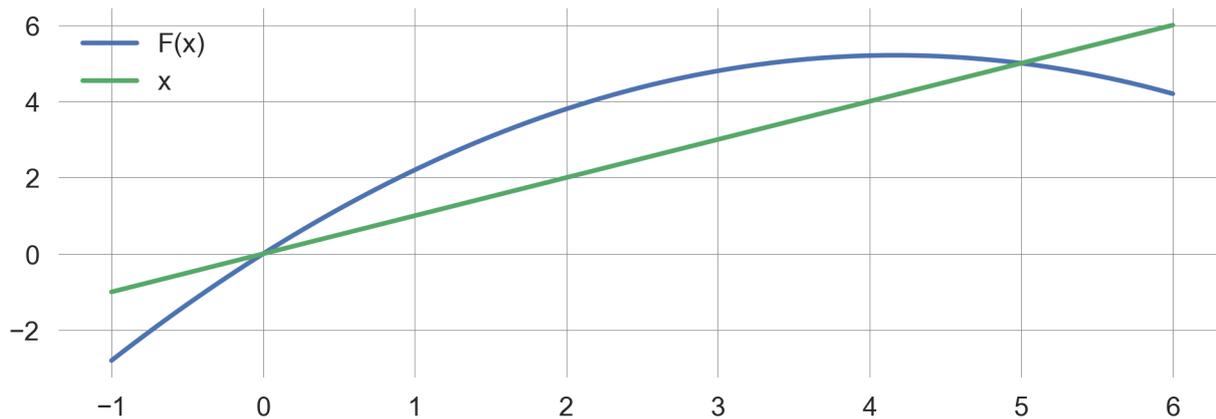
We can enrich this representation with the cobweb plot, which shows the system's time evolution from an initial state to the equilibrium point.

```
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();
```
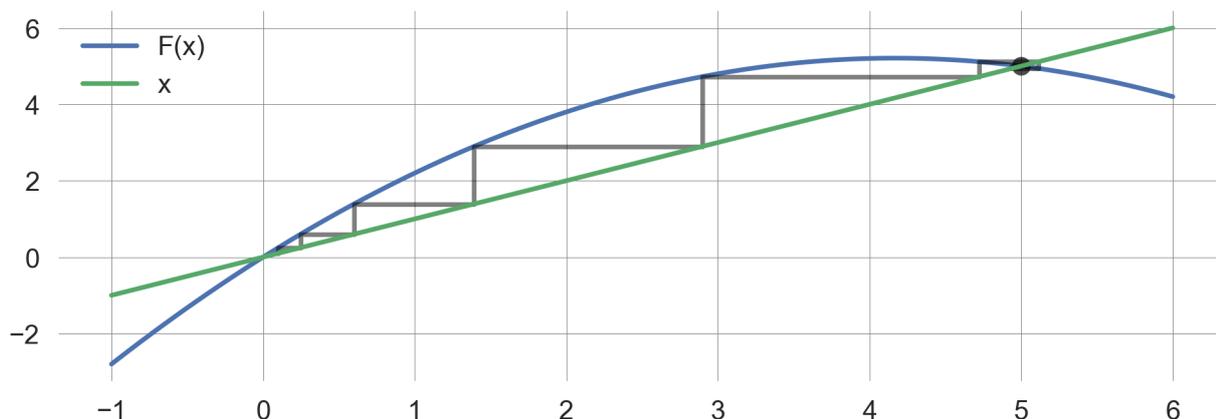

2.6.2 Linear stability in nonlinear systems

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 2.21.

```
plot()
```
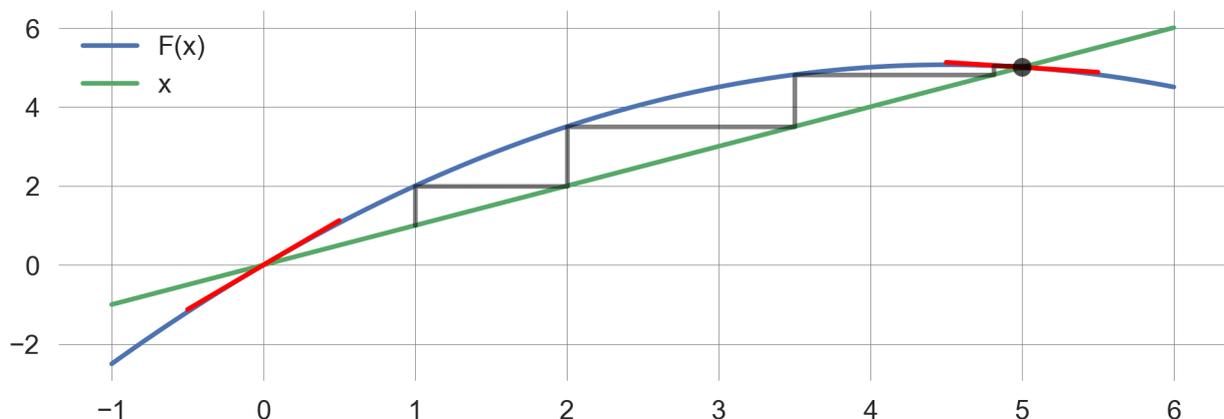

Figure 2.21: Linear stability shown at the nonlinear logistic map.

---

[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.

2.7 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.

2.7.1 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.

3 Tipping elements

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

3.1 Motivation

Think of the term "tipping point" in the context of sustainability. What do you associated with it? What does it mean? What are examples of tipping points in the context of human-environment interactions?
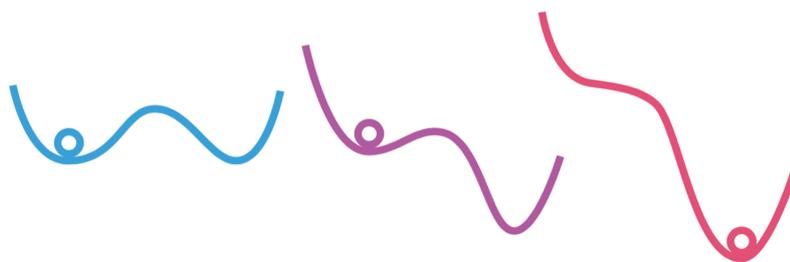
"the point at which a series of small changes or incidents becomes significant enough to cause a larger, more important change" (Oxford English Dictionary)
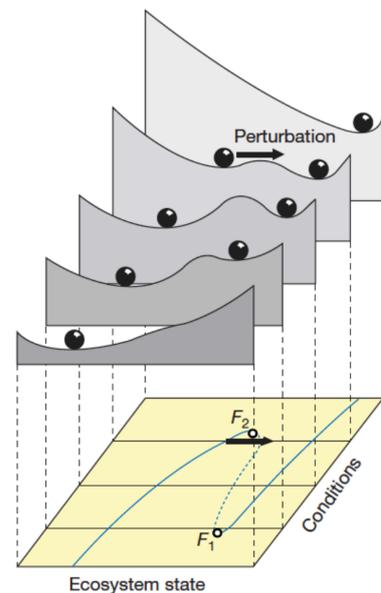
Lenton et al. (2023) *The Global Tipping Points Report 2023*   Scheffer et al. (2001) Catastrophic shifts in ecosystems

Figure 3.1: Tipping points, elements and regime shifts

Tipping points, elements and regime shifts   The concepts of tipping elements and regime shifts are closely related aspects of complex systems dynamics. While tipping elements refer more to the components of a system with the potential for abrupt change, regime shifts refer more to the actual transitions that occur when these elements cross their critical thresholds. Also, the term tipping elements is used more in the context of Earth system science, while regime shifts are used more in the context of social-ecological systems.

Exceeding 1.5°C global warming could trigger multiple climate tipping points
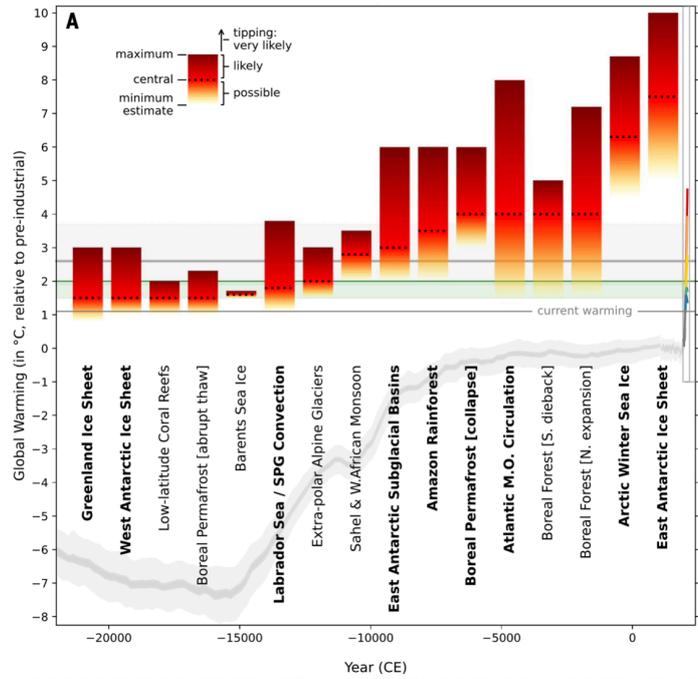
Armstrong-McKay (2022) *Exceeding 1.5°C global warming could trigger multiple climate tipping points*

Figure 3.2: Climate tipping risks

Climate tipping risks Climate tipping points are thresholds in the **Earth's climate system** that, such as a slight increase in global average temperature, when crossed, can lead to significant and potentially irreversible changes. These changes can trigger **reinforcing feedback loops** that push the system into a new equilibrium, potentially leading to severe consequences like accelerated ice melt or shifts in ocean currents. For instance, the collapse of the West Antarctic ice shelves is a potential climate tipping point that could lead to substantial sea level rise and other impacts. While some tipping points may be **triggered within the 1.5-2°C Paris Agreement range**, many more become likely at 2-3°C of warming (Armstrong McKay et al., 2022).

Parts of the Earth system identified as featuring tipping points
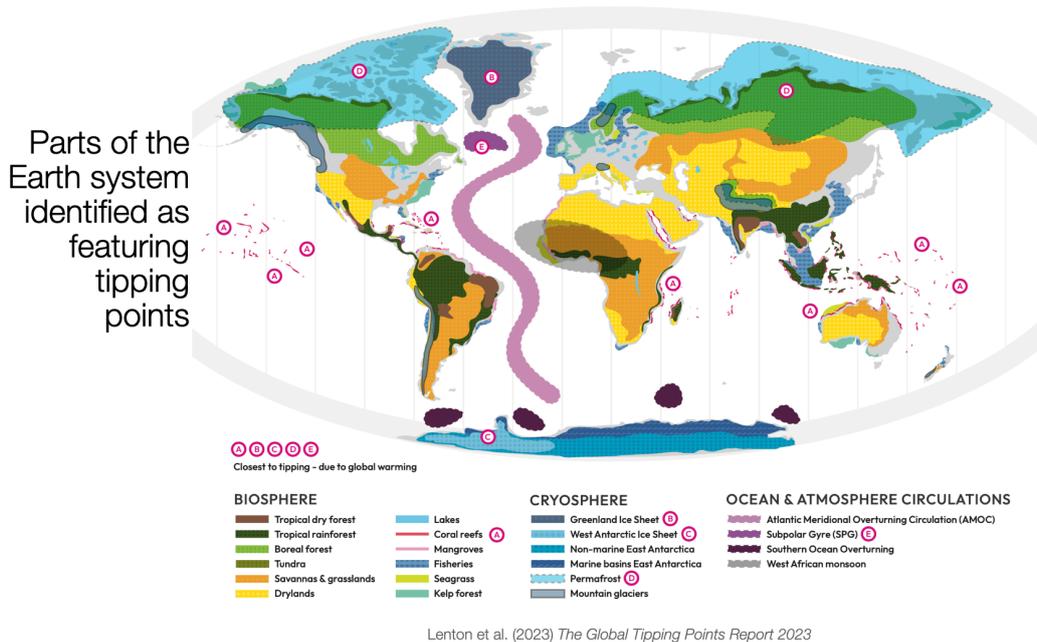
Lenton et al. (2023) *The Global Tipping Points Report 2023*

Figure 3.3: Parts of the earth system with tipping points

Global Tipping Points While climate tipping points are specific to the Earth's climate systems and their feedback mechanisms, global tipping points (Lenton et al., 2023) consider a wider array of **interconnected systems, including human and ecological dimensions**, highlighting the complex interplay between natural and societal changes. **Natural tipping points** may occur over the entire Earth system, from the Biosphere to the Cryosphere, the Oceans and the Atmosphere.

Currently, several major tipping points are at imminent risk due to global warming, with more projected as temperatures rise above 1.5°C. The cascading effects of these negative tipping points could overwhelm global social and economic systems, outpacing some countries' adaptive capacities. Addressing these crises requires a transformative shift away from incremental changes **towards a robust global governance framework** that prioritizes **rapid emission reductions** and **ecological restoration**.

Social tipping points to the rescue?

Lenton et al. (2023) *The Global Tipping Points Report 2023*

Figure 3.4: Social tipping points

Simultaneously, it's crucial to identify and harness **positive tipping points**, where beneficial changes can become self-sustaining, potentially offsetting some negative impacts. There is an urgent need to build **resilient societies** capable of withstanding impending challenges and seizing opportunities for sustainable progress. The paradigm of 'business as usual' is obsolete; instead, a proactive approach to governance and global cooperation is essential to navigate towards a sustainable future, leveraging both the threats and opportunities posed by tipping points (Lenton et al., 2023).

Challenges **So what exacltly are tipping elements and regime shifts?**

How can we identify them?

And how can we manage them?

Here, the mathematics of bifurcations can help.

3.1.1 Learning goals

After this chapter, students will be able to:

- **Explain** the concept of a **bifurcaton** and how it relates to tipping points and regime shifts.
- Explain a simple **dynamic system to model a tipping element** or regime shift.
- Explain what an *attractor*, *transient*, *basin of attraction* and *separatrix* are.
- Conduct a **bifurcation analysis** in a simple dynamic system using Python.
- **Construct** a **potential function** and explain its role in bifurcation analysis.
- Explain and recognize **hysteresis** and its consequences for sustainability transitions.

3.2 Bifurcations | The mathematics of tipping elements

The key question bifurcation theory addresses is: **How does the system's long-term behavior depend on its parameters?**

The **distinction** between a system's *state* and its *parameters* is **crucial**. The state of a system is the set of variables that describe the system at a given time, while the parameters are the constants that define the system's behavior.

Often, a small change in parameter values causes only a small or even no quantitative change in the system's state. However, sometimes, a slight change in parameter values causes a drastic, qualitative change in the system's behavior.

> A **bifurcation** is a qualitative, topological change of a dynamic system's phase space that occurs when some parameters are slightly varied across their **critical thresholds**.

Here, we cover the very basics of bifurcation theory in dynamic systems. These provide a rich understanding of tipping points and regime shifts in the sustainability sciences.

We start by importing the necessary libraries and setting up the plotting environment.

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

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;
```

We also inlcude some keys functions to simulate dynamic systems from 02.01-Nonlinearity.

To run the model we copy and refine 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)
        if np.abs(stock)>10e9: break  # stop the simulation when x becomes too large
```

```
        time_series.append(stock)
    return np.array(time_series)
```

We also copy 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("System state");
    return time_series
```

And last, we copy the cobweb plot function over.

```
def cobweb(F, x0, params, iters=10, ax=None):
    h=[x0]; v=[x0]; x=x0  # lists for (h)orizontal and (v)ertical points
    for _ in range(iters):  # iterate the dynamical system
        x_ = F(x, **params)  # get the next system's state
        if np.abs(x)>10e9: break  # stop the simulation when x becomes too large
        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 axv
    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', alpha=0.7)  # plot dot

    return h, v
```

3.2.1 A minimal model of tipping elements

Let x denote the **property of a system we are interested** in, such as the amount of ice in the Arctic, the population of a species, or the fraction of a lake's surface coverd by vegetation (Scheffer et al., 2001). Thus, we describe the system's state over time t by x_t.

Conceptually, the system's dynamics are influenced by a **reinforcing feedback loop**, a **balancing feedback loop**, an **external influence** c. The exertnal influence c, for example, represents the global mean temperature in the case of climate tipping elements, or the level of nutrients in the example of a lake regime shift.
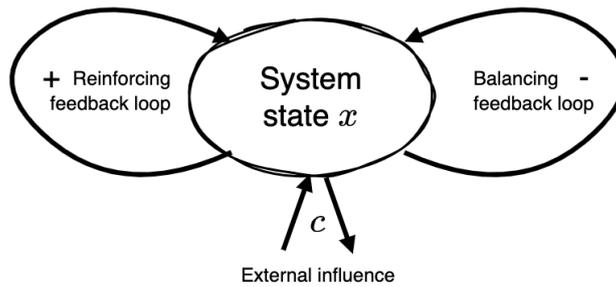
Figure 3.5: Tipping element model

A simple mathematical representation of such a system has the difference euqation,

$$\Delta x = (x - ax^3 + c)\frac{1}{\tau},$$

where τ represents the typical time scale of the system, and thus, inverse strength of the system's change, and a is a parameter that determines the strength of the balancing feedback loop in relation to the reinforcing feedback loop (with unit stength).

We define the `update_stock` function, $F(x_t$ for the udpate $x_{t+1} = F(x_t)$, to iterate the stock x according to the difference equation above.

```
def F_tipmod(x, c, a=1, tau=10): return x + (x - a*x**3 + c)/tau
```

We explore the stock's evolution over time from two different initial conditions by iterating the model for 200 time steps.

```
def compare_initial_conditions(nr_timesteps=200, c=0.0, tau=50, a=1):
    plot_stock_evolution(nr_timesteps, 1.2, F_tipmod, c=c, tau=tau, a=a);
    plot_stock_evolution(nr_timesteps, -1.2, F_tipmod, c=c, tau=tau, a=a);
    paramstring = f"c={c}, tau={tau}, a={a}"
    plt.gca().annotate(paramstring, xy=(0, 1.0), xycoords='axes fraction',
↪   va='bottom', ha='left'); plt.show()
```

```
compare_initial_conditions()
```


63
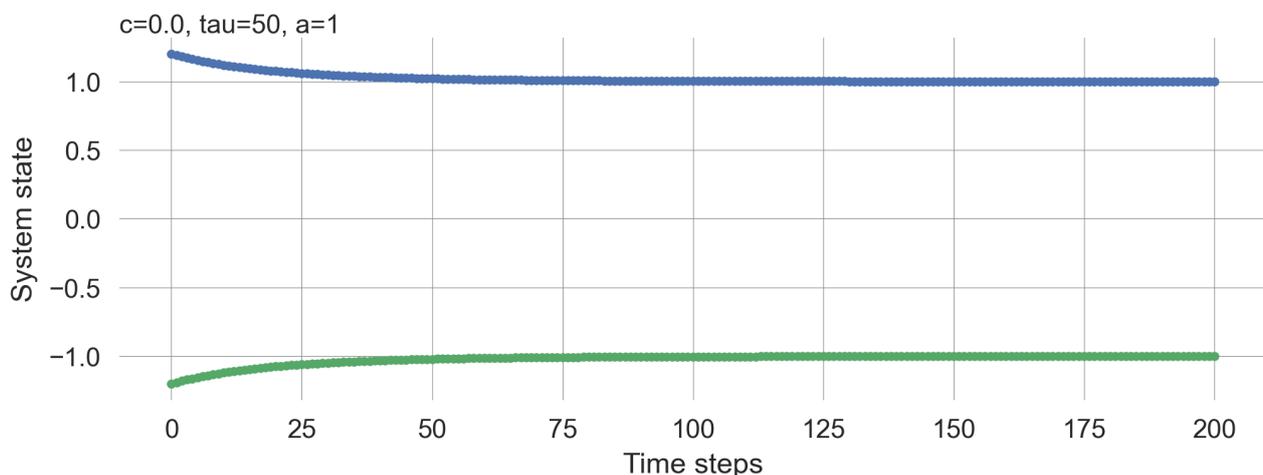
We observe bi-stability. Depending on where the dynamical system starts, it will either converge to the fixed point $x_e = 1.0$, or the the fixed point $x_e = -1.0$

How do the parameters influence the system's evolution? We can convince ourselves that the timescale parameter τ determines the speed of the system's evolution (vary τ and the total number of simulation staps proportionally: the curves's shapes look identical). The parameter a scales the system's fixed points (vary a and observe the system's behavior). Finally, We can also observe, that the external influence c can change the system's equilibrium state. Run this notebook interactivly and confirm these observations for yourself!

3.2.2 Cobweb plot

Let us observe thhis phenomenon of bi-stability in a cobweb plot.

```
def cobweb_plot(c=0, tau=1.5, a=1):
    xs = np.linspace(-2,2,101); plt.xlabel('x'); plt.ylim(-1.3,1.3);
↪   plt.xlim(-1.6,1.6);
    plt.plot(xs, F_tipmod(xs, c,a,tau), label='F(x)');
    plt.plot(xs, xs, label='x', color='k', alpha=0.5); plt.legend();
    cobweb(F_tipmod, x0=0.3, params=dict(c=c, a=a, tau=tau), iters=100,
↪   ax=plt.gca());
    cobweb(F_tipmod, x0=-0.3, params=dict(c=c, a=a, tau=tau), iters=100,
↪   ax=plt.gca());
```

```
cobweb_plot()
```
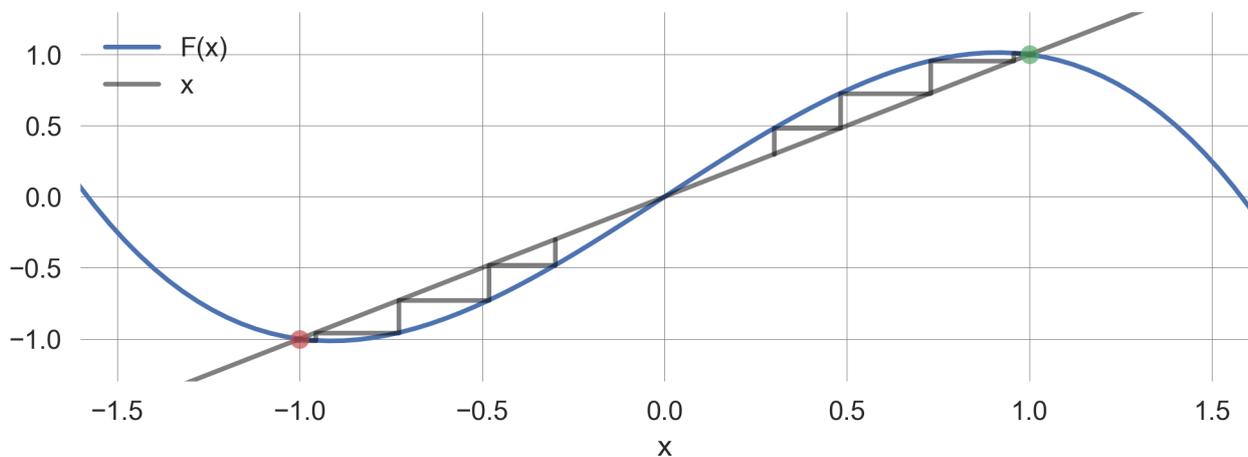

We see, that it depends on where the update function $F(x_t)$ intersects the diagonal line $y = x$ whether an inital condition converges to the fixed point $x_e = 1.0$ or $x_e = -1.0$. The external influence parameter c determines this intersection point

Some definitions.

An **attractor** is a set of states toward which a dynamic system tends to evolve over time. These states represent the system's long-term behavior. Once the system reaches an attractor, it typically remains there. For example, in the system above the attractors are the fixed points $x_e = 1.0$ and $x_e = -1.0$.

A **transient** refers to the behavior of a system during a limited period of time before it reaches an attractor. For example, the cobweb plot shows the transient behavior of the system.

A **basin of attraction** the set of all the initial states from which you will eventually end up falling into that attractor. For example, in the system above, the basin of attraction for the fixed point $x_e = 1.0$ are all point greater than the intersection point between $F(x_t)$ and $y = x$. The basin of attraction for the fixed point $x_e = -1.0$ are all points less than the intersection point between $F(x_t)$ and $y = x$.

If there are more than one attractor in the phase space, you can divide the phase space into several different regions. In this case, a **separatrix** is the boundary between distinct basins of attractions. For example, in the system above, the separatrix consists only of the intersection point between $F(x_t)$ and $y = x$.

3.2.3 Empirical bifurcation diagram

A bifurcation diagram is a powerful tool to visualize the system's long-term behavior as a function of its parameters. To create a bifurcation diagram, we iterate the model for a range of parameter values and plot the system's equilibrium states.

```
def simulate_bifurcation_diagram(F, x0s, params, iters=1000, cextent=[-0.5,0.5],
                                 pointsize=2.0):
    c_s = np.linspace(cextent[0], cextent[1], 501) # The external parameter to be
↪    varied
    for x0 in x0s:   # Loop through all initial conditions
        endpoints = []   # Container to store the endpoints
        for c in c_s:   # Loop through all external parameter values
            trj = iterate_model(iters, x0, F, c=c, **params)   # Simulate the system
            endpoints.append(trj[-10:])   # Taking the last 10 points of the
↪    trajectory
        # Plotting the endpoints
        cpoints = [[c_s[i]]*l for i, l in enumerate(map(len, endpoints))]   # create
↪    cpoints that may work for different endpoint lengths
        plt.scatter(np.hstack(cpoints), np.hstack(endpoints), c='k', alpha=0.5,
                    s=pointsize);   # np.hstack unpacks everything
    plt.ylabel(r'Equilibrium state $x$'); plt.xlabel(r'External influence $c$')
```

```
simulate_bifurcation_diagram(F_tipmod, x0s=[-1.5, 1.5], params=dict(tau=10 , a=1.0),
↪    iters=1000)
```
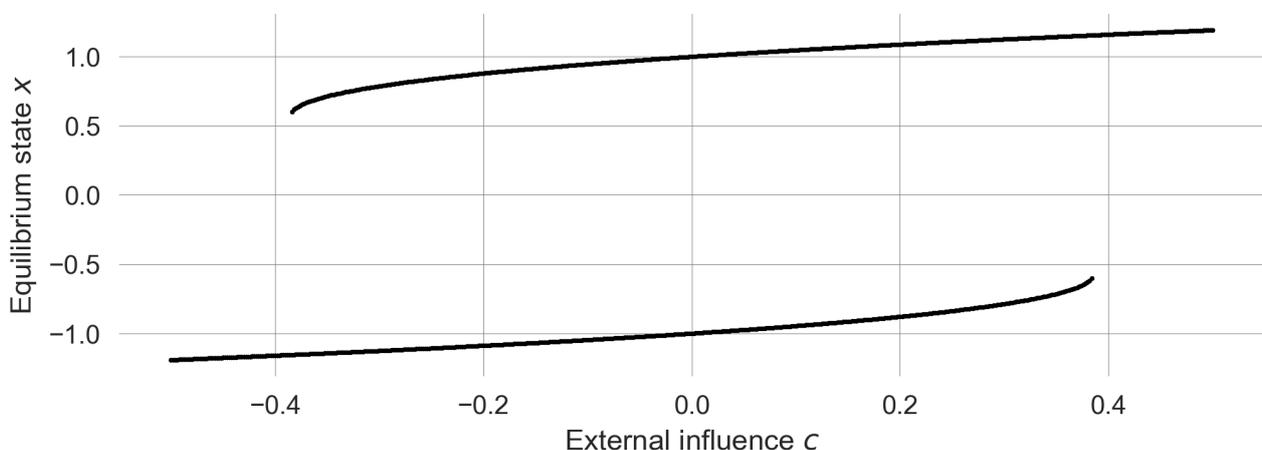
This **empirical bifurcation diagram** allows us to identify the system's stable fixed points as a function of the parameter c. We observe the **range of parameter values** for which the system **converges** to the fixed points around $x_e = 1.0$ and $x_e = -1.0$, as well as the range where the system is **bi-stable**. We also observe the **critical values** of c where the system undergoes a qualitative change in its behavior. When the external parameter c changes around these critical values in c (close to -0.4 and 0.4 here), a tiny change causes a drastic effect on the system state.

3.2.4 Conducting a bifurcation analysis

Local bifurcations occur when the stability of an equilibrium point changes between stable and unstable.

1) **Determine** the **equilibirum points** in dependence of the model parameters
2) **Determine** the **stability** of the equilibirum points in dependence of the model parameters. For one-dimensional systems $x_{t+1} = F(x_t)$, an equilibrium point is stable when $|F'(x_e)| < 1$.
3) **Bifurcations** occur at parameter values at which the **stability changes**. For one-dimensional systems $x_{t+1} = F(x_t)$, local bifurcations occur when $|F'(x_e)| = 1$.

3.2.5 Step 1 | Equilibrium points

The equilibrium points for $\Delta x = \frac{1}{\tau}(x - ax^3 + c)$ fulfill,

$$c = ax^3 - x.$$

It is not straightforward to solve the equation, $c = ax^3 - x$, analytically, i.e., to give an expression for how the system's equilibirum depends on the parameters c and a. However, we can plot the parameter c as a function of the equilibirum points x_e and the parameter a.

```python
def plot_equilibirum_points_tipmod(a=1.0, cextent=[-2.0,2.0]):
    xe=np.linspace(-2.0,2.0,501) # equilibrium points
    c = a*xe**3 - xe # parameter c
    plt.plot(c, xe, "--"); # plot
    plt.xlabel(r'External parameter $c$'); plt.ylabel(r'Equilibrium points $x_e$');
    # plt.xlim(cextent);
```

```python
plot_equilibirum_points_tipmod()
```
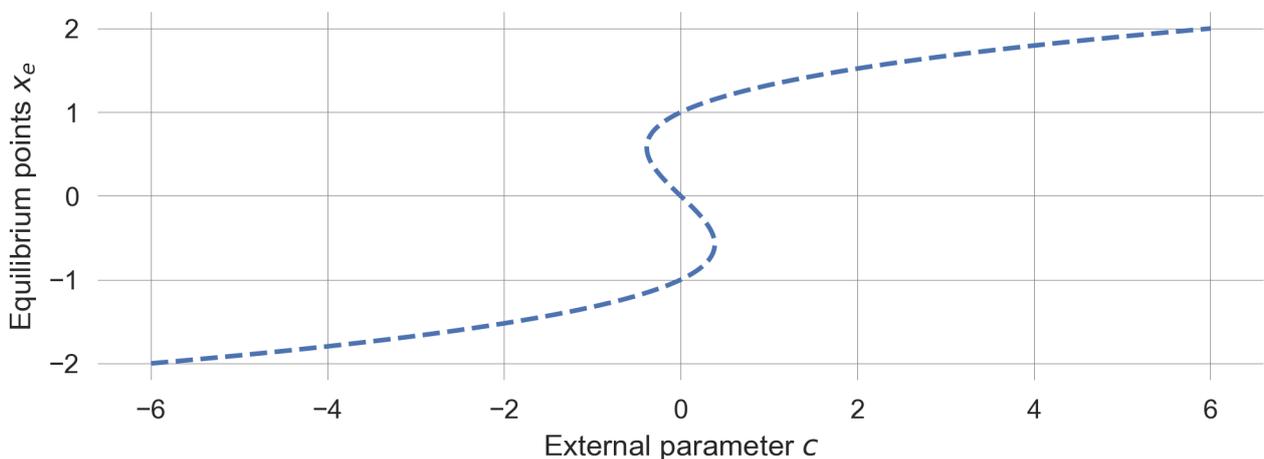
3.2.6 Step 2 | Stability

Computing the derivative of the update function $F(x_t) = x + \frac{1}{\tau}(x - ax^3 + c)$, we find,

$$\frac{dF}{dx} = 1 + \frac{1}{\tau}(1 - 3ax^2).$$

We create a Python function to plot the whether an equilibrium point is stable or not using the `np.logical_and` function.

```python
def plot_stability_tipmod(a=1.0, tau=3.0, cextent=[-3.0,3.0]):
    xe=np.linspace(-3, 3, 1001) # equilibrium points
    c = a*xe**3 - xe # parameter c

    def F_(x, a,tau): return 1 + (1-3*a*x**2)/tau
    cond=np.logical_and(F_(xe, a,tau)<1, F_(xe, a,tau)>-1)
    plt.plot(c[cond], xe[cond], ".", c='red')

    plt.xlabel(r'External parameter $c$'); plt.ylabel(r'Equilibrium points $x_e$');
    plt.xlim(cextent);
```

Brining stability and equilibrium points together, we can plot an analytical bifurcation diagram.

```python
a = 1.0; tau=3.0
plot_equilibirum_points_tipmod(a=a);
plot_stability_tipmod(a=a, tau=tau)
```
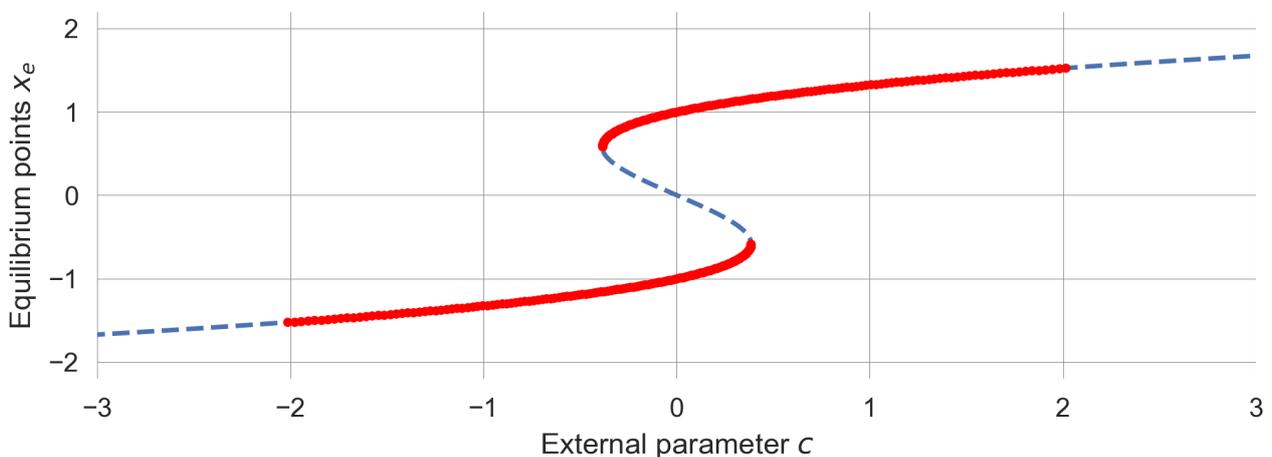

3.2.7 Step 3 | Bifurcation diagram

We enrich this bifurcation diagram by solving $\frac{dF}{dx} = 1 + \frac{1}{\tau}(1 - 3ax^2)$ for $\frac{dF}{dx} = 1$ and $\frac{dF}{dx} = -1$ yields the **stability boundaries**

$$x_b = \pm\sqrt{\frac{1}{3a}} \quad \text{and} \quad x_b = \pm\sqrt{\frac{2\tau + 1}{3a}}.$$

We create a Python function to plot the stability boundaries.

```
def plot_stability_boundaries_tipmod(a= 1.0, tau=3, cextent=[-3.0, 3.0]):
    styl = dict(ls=":", lw=0.75, color='green')
    plt.plot(cextent,[np.sqrt(1/(3*a)), np.sqrt(1/(3*a))], **styl)
    plt.plot(cextent,[-np.sqrt(1/(3*a)), -np.sqrt(1/(3*a))], **styl)
    plt.plot(cextent,[np.sqrt((2*tau + 1)/(3*a)), np.sqrt((2*tau + 1)/(3*a))],
↪   **styl)
    plt.plot(cextent,[-np.sqrt((2*tau + 1)/(3*a)), -np.sqrt((2*tau + 1)/(3*a))],
↪   **styl)
```

Bringing all togehter, we obtain our analytical bifurcation diagram.

```
def plot_analytical_bifurcation_tipmod(a = 1.0, tau = 3.0, extent=3.0):
    plot_equilibirum_points_tipmod(a=a, cextent=[-extent, extent]);
    plot_stability_tipmod(a=a, tau=tau, cextent=[-extent, extent]);
    plot_stability_boundaries_tipmod(a=a, tau=tau, cextent=[-extent, extent]);
```

```
a = 1.0; tau = 3.0
plot_analytical_bifurcation_tipmod(a = a, tau = tau, extent=1.0)
```
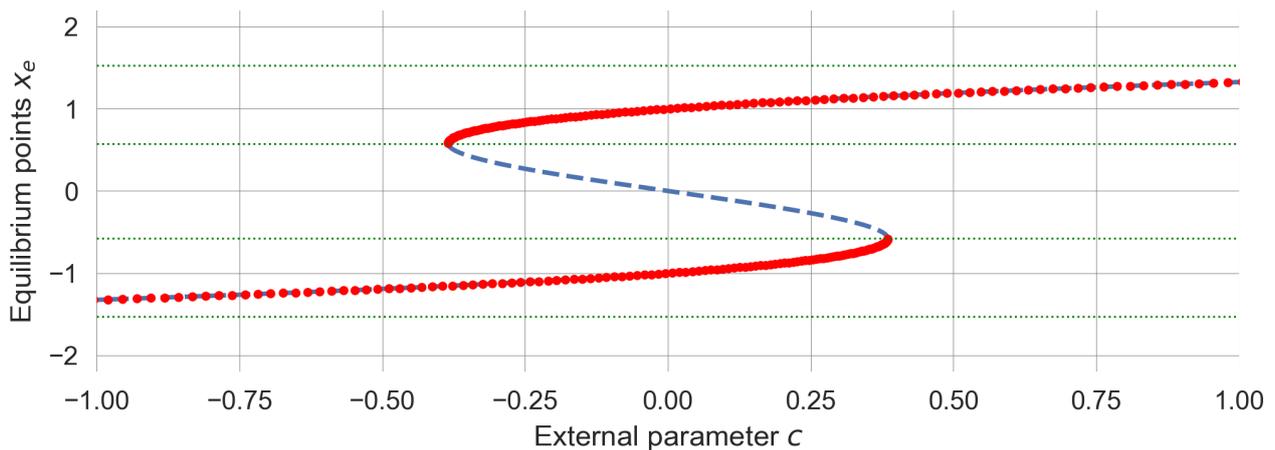

Lastly, we can compare the empirical bifurcation diagram with the analytical bifurcation diagram, and observe that both match perfectly.

```
a = 1.0; tau = 3.0
plot_analytical_bifurcation_tipmod(a = a, tau = tau, extent=2.0)
simulate_bifurcation_diagram(F_tipmod, x0s=[-0.5, 0.5], params=dict(tau=tau , a=a),
                             iters=500, pointsize=30, cextent=[-2.0, 2.0])
```
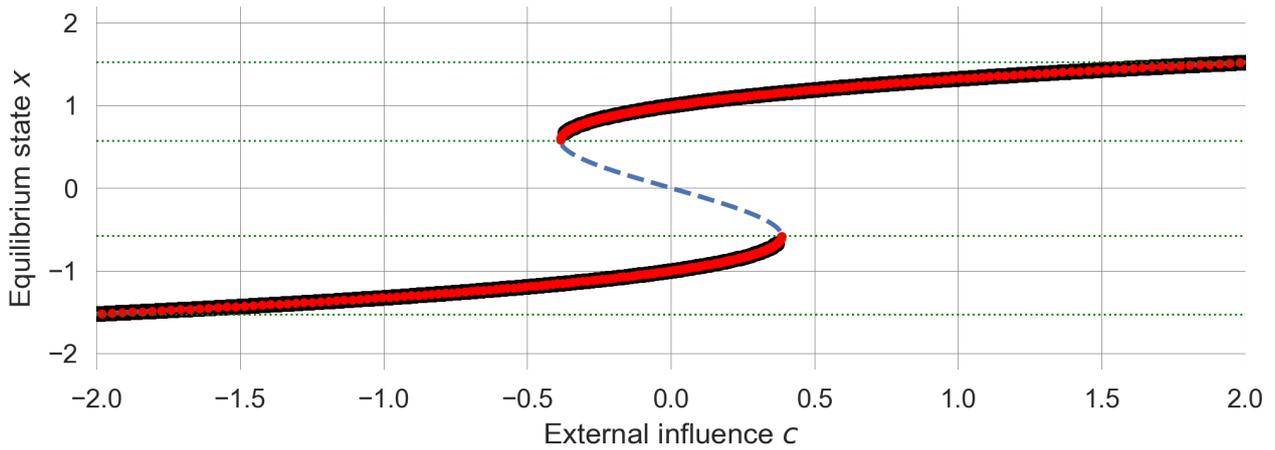
Our bifurcation analysis produces the same diagram we observed in the literature. What is still missing is the changing stability landscape portrayed in Figure 3.6?
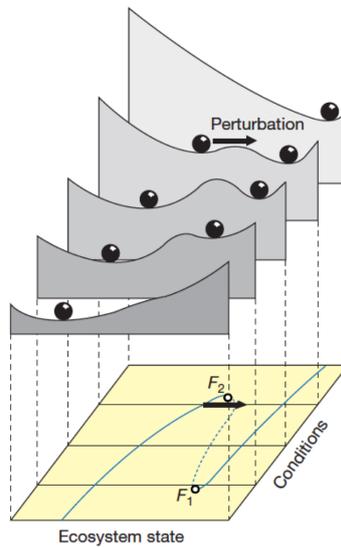
Scheffer et al. (2001) *Catastrophic shifts in ecosystems*

Figure 3.6: Conceptual Regime Shift

3.2.8 Potential function

A potential is a function that describes the *energy* of a system. In the context of dynamic systems, a potential function can help us understand the system's behavior by visualizing the system's *energy* landscape.

Figure 3.7: Illustrations of potential functions

In general, there are multiple ways to define a potential function. Here, we define a potential function G as the negative integral of the system change Δx. Thus, for a system $x_{t+1} = F(x_t) = x_t - \frac{G(x)}{dx}\big|_{x=x_t}$, we have

$$\Delta x = -\frac{G(x)}{dx}.$$

The idea is, that the system changes as if *rolling* downard (according to the first derivative of) the potential landscape $G(x)$.

Thus, for the difference equation $\Delta x = \frac{1}{\tau}(x - ax^3 + c)$, we have

$$G(x) = -\frac{1}{\tau}\left(\frac{1}{2}x^2 - \frac{1}{4}ax^4 + cx\right).$$

Converting this into Python yields,

```python
def G_tipmod(x, c,a,tau): return - (x**2/2 - a*x**4/4 + c*x)/tau
```

which we use in a `plot_potential` function to visualize the potential landscape.

```python
def plot_tipmod_potential(c=0.2, a=1.0, tau=2):
    xs=np.linspace(-2,2,501); plt.ylim(-0.5, 0.5);
    plt.plot(xs, G_tipmod(xs, c,a,tau), color='blue')
    plt.ylabel(r'Potential $G(x)$'); plt.xlabel(r'System state $x$')

    #  numerically find and plot equilibrium points
    c_ = a*xs**3 - xs
    xeq = xs[np.isclose(c_-c, 0.0, atol=0.02)]
    plt.plot(xeq, G_tipmod(xeq, c, a, tau), 'o', ms=12, color='k')
```

```
plot_tipmod_potential()
```
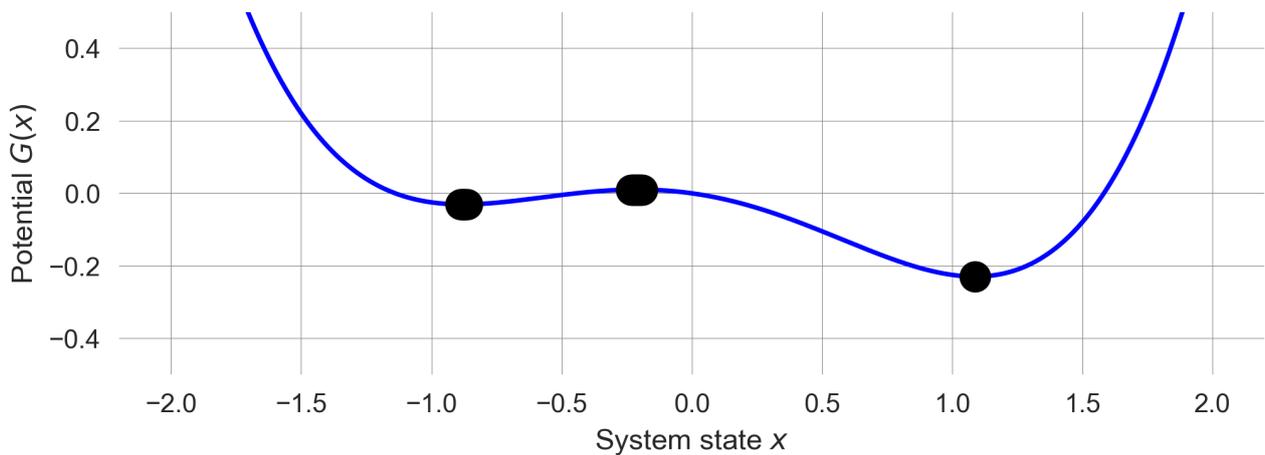

Finally, we bring all pieces together to visualize the system's potential landscape, bifurcation diagram and time evolution.

```
def plot_all_tipmod(c=0.2, a=1.0, tau=2):
    fig = plt.figure(figsize=(9, 4))

    fig.add_subplot(221)
    plot_tipmod_potential(c=c, a=a, tau=tau)

    fig.add_subplot(222)
    plot_analytical_bifurcation_tipmod(a=a, tau=tau, extent=1.4)
    plt.plot([c,c], [-2,2], "-", color='black')

    fig.add_subplot(313)
    compare_initial_conditions(nr_timesteps=50, c=c, tau=tau, a=a)

    plt.tight_layout();
```

```
plot_all_tipmod();
```

```
<Figure size 2340x750 with 0 Axes>
```
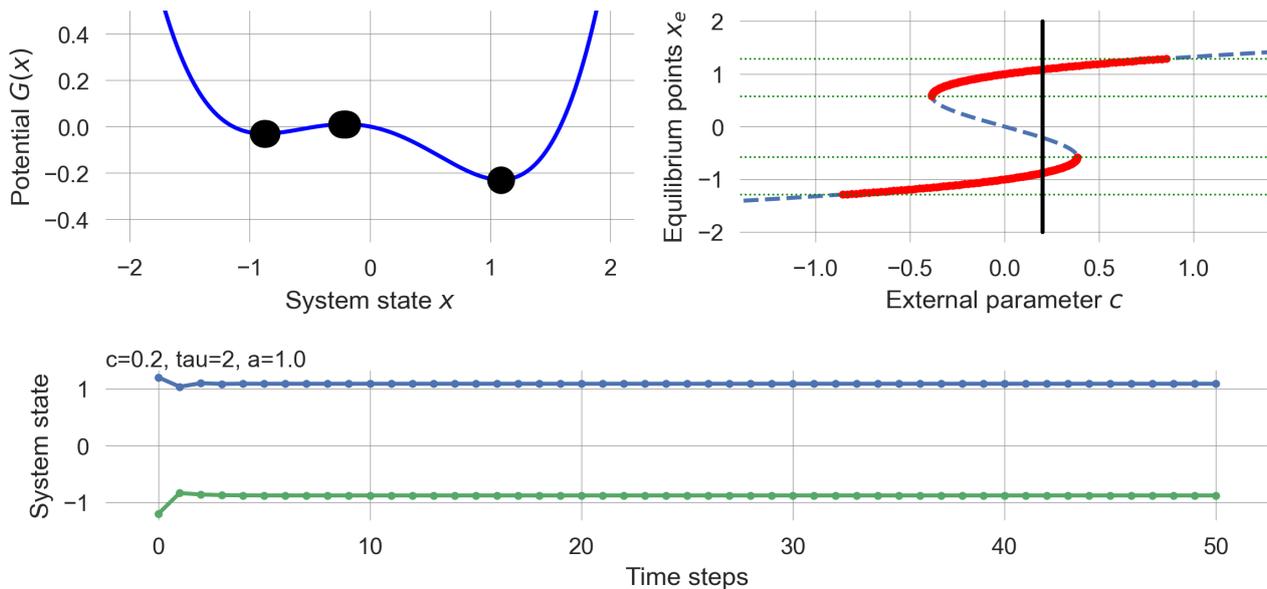
3.2.9 Hysteresis

The last phenomenon we want to explore is **hysteresis**. Hysteresis occurs when the system's behavior depends on its history, i.e., the system's current state depends on its past states.

We let our tipping element model iterate until it reaches an equilibirum point and then slighlty change the external influence parameter c.

1) We start from a low value of external influence parameter c such that the system equilibriates toward the negative equilibrium point and then increae c into the range where only the positive equilibrium point is stable.
2) Then, we decrease c back to the range where only the negative equilibrium point is stable.

```python
def plot_hysteresis():
    x=-1; xs = [] # inital condition and container for the system state
    cvs = np.linspace(-0.8, 0.8, 101);  # values of parameter a to go through
    cvs = np.concatenate((cvs, cvs[::-1])); # first we go up, then we go back down

    for c in cvs: # looping through all parameter values
        for _ in range(100): x=F_tipmod(x, c=c, a=1.0, tau=2.0);  # iterating the
        ↪  system 100 times
        xs.append(x); # storing the last system state

    plt.plot(cvs, xs,'-',alpha=0.5, color='gray',zorder=-1) # Plot background line
    plt.scatter(cvs, xs, alpha=0.9, s=np.arange(len(cvs))[::-1]+1,
    ↪  c=np.arange(len(cvs)), cmap='viridis'); # Colorful plot
    plt.xlabel(r"Influence parameter $c$"); plt.ylabel(r"System state $x$");
```

```python
plot_hysteresis()
```
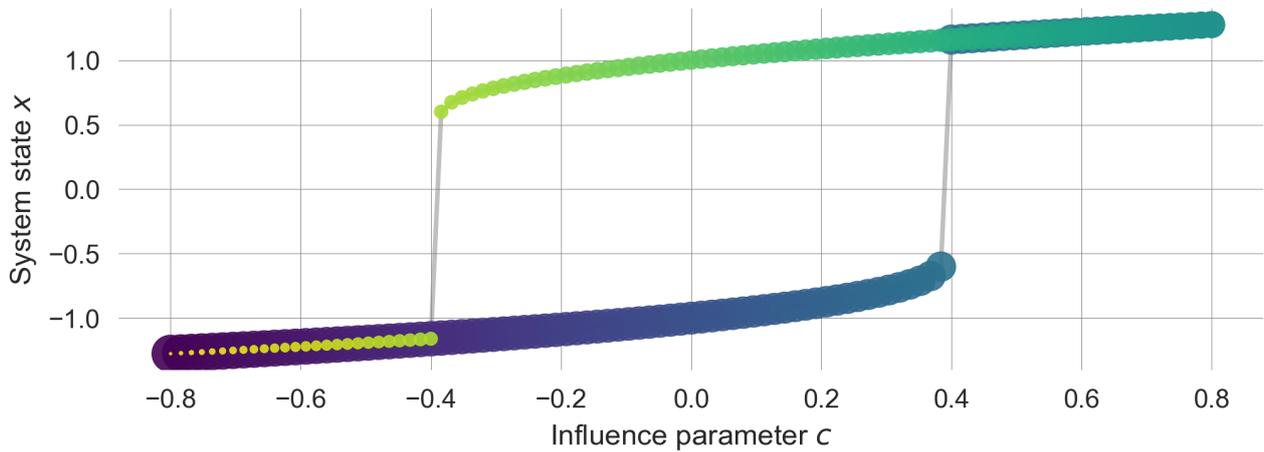
Time moves from *large* to *small* and *dark* to *light* dots.

Hysteresis is not only a theoretical construct. It occurs in many practical real-world domains from physics, chemistry, engineering, biology, to economics.
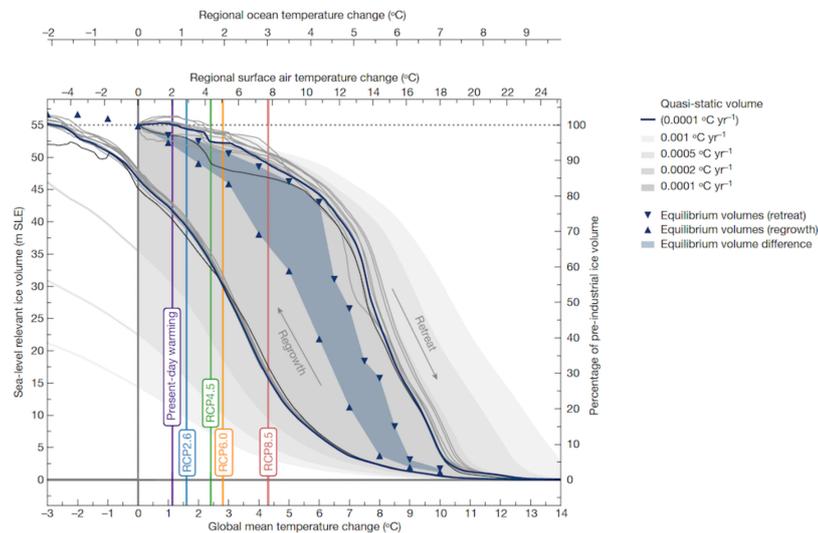
Garbe et al. (2020) *The hysteresis of the Antarctic Ice Sheet*

Figure 3.8: Hysteresis of the Antarctic Ice Sheet

The hysteresis of the Antarctic Ice Sheet refers to the phenomenon where the ice sheet's response to temperature changes is not symmetric; i.e., the thresholds for ice growth and decline differ significantly (Figure 3.8). This behavior has critical implications for understanding future sea-level rise under global warming scenarios.

The Antarctic Ice Sheet exhibits multiple temperature thresholds, beyond which ice loss becomes irreversible. For instance, at 2°C warming, West Antarctica faces long-term partial collapse due to marine ice-sheet instability. A significant loss of over 70% of the ice volume is anticipated with 6 to 9°C warming, primarily driven by surface elevation feedback (Garbe et al., 2020).

3.3 Learning goals revisited

In this chapter, we have explored the concept of a **bifurcation** and its significance in understanding tipping points and regime shifts. We examined how small changes in system parameters can lead

to substantial shifts in behavior, highlighting the importance of bifurcations as precursors to critical transitions.

We then introduced a simple **dynamic system** model to represent a **tipping element** or regime shift, giving us a framework to analyze and simulate how systems behave under the influence of varying forces and feedback mechanisms. In this context, we explained key concepts such as **attractors**, **transients**, **basins of attraction**, and **separatrices**. These elements helped us understand the structure of the system's state space, illustrating how it is shaped by stable and unstable regions.

To deepen our analysis, we conducted a **bifurcation analysis**, demonstrating how a system's behavior changes as we adjust specific parameters. This analysis allowed us to identify potential tipping points and provided a practical approach to studying system dynamics.

Furthermore, we constructed a **potential function** and examined its role in bifurcation analysis, as it represents the energy landscape and stability of a system. By analyzing the shape and contours of this potential function, we gained insight into where attractors are located and how the system may transition between states.

Finally, we discussed **hysteresis** and its implications for **sustainability transitions**. We observed that hysteresis introduces a kind of path-dependence, where returning to an original state may require more than simply reversing parameter changes. This phenomenon has critical consequences for sustainability, underscoring the challenges in restoring systems after they have undergone significant transformations. Together, these insights equip us with a deeper understanding of complex system dynamics, emphasizing the importance of identifying and managing critical transitions effectively.

4 Resilience

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

4.1 Motivation | Resilience in sustainability contexts

Think of the term "resilience" in the context of sustainability and human-environment interactions. What does it mean to you? How can we model it? How can we measure it? What are the key challenges and opportunities in this field?

4.1.1 Resilience everywhere

Capacity of a system to cope with shocks

from latin **resiliō** ("to spring back")

Resilience is a widly used term in many different fields, from psychology to engineering, from ecology to social-ecological systems.

Figure 4.1: https://www.mind-berry.com/wp-content/uploads/2023/06/Resilience-image.png

Psychology   source: mind-berry.com | What is resilience?

Resilience is the capacity to recover from challenges and use them as learning opportunities. Resilient peole are perceived as having a positive outlook, handling difficulties calmly, and managing negative emotions effectively.
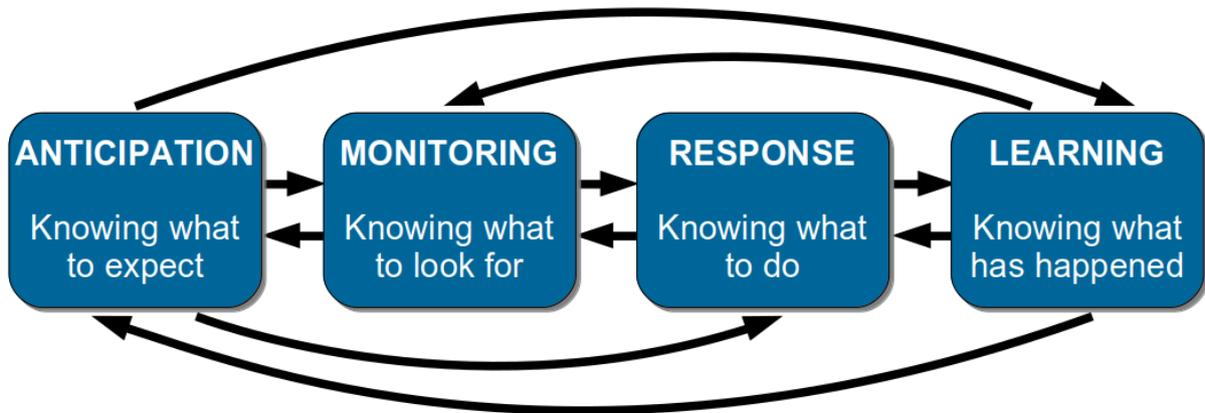
Figure 4.2: Resilience engineering

Engineering   source: rote.se | Resilience engineering

Resilience in engineering refers to the ability of complex systems to anticipate, adapt to, and recover from unexpected disruptions or failures. This field emphasizes not just the prevention of failures but also the capacity to maintain functionality and performance in the face of unforeseen challenges.
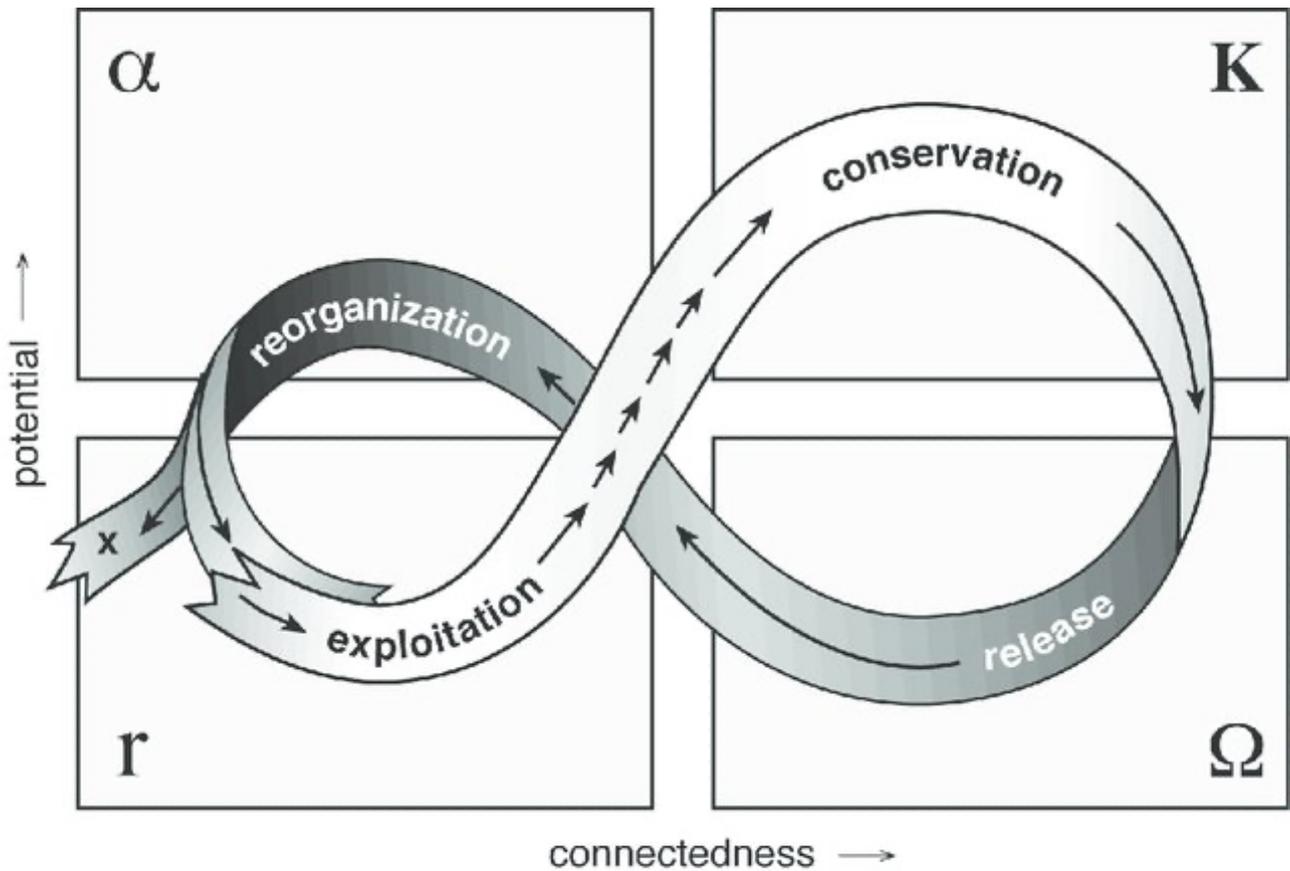
The adaptive cycle (from Panarchy, edited by Lance H. Gunderson and C.S. Holling: Figure 2-1 (page 34). Copyright © 2002 Island Press.

Figure 4.3: Resilience in ecology | the adaptive cycle

Ecology   source: resalliance.org | Adaptive cycle

Resilience in ecology refers to the capacity of an ecosystem to endure disturbances while preserving its core functions, structures, and processes. This concept includes the ability to recover and adapt to environmental changes, allowing ecosystems to endure challenges and thrive.
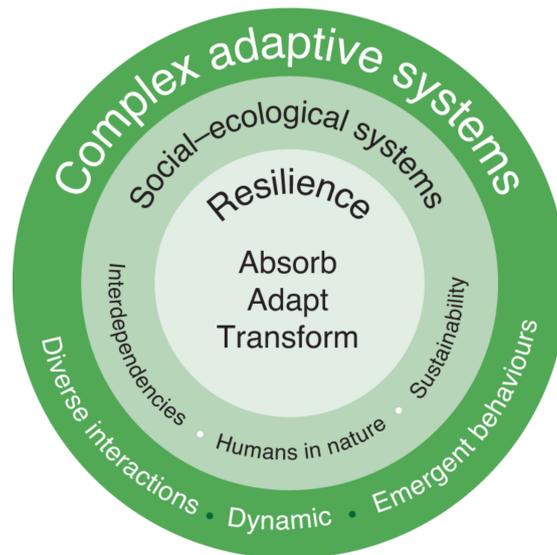
Reyers et al. (2022) *The contributions of resilience to reshaping sustainable development*

Figure 4.4: Resilience in human-environment interactions

Human-environment interactions Resilience in sustainability and human-environment interactions means the ability of social and ecological systems to absorb disturbances, adapt to changes, and maintain functionality. This concept is crucial as it highlights how communities and ecosystems withstand environmental stressors like climate change, pollution, and resource depletion, but also social stressors like economic crises, conflicts, and pandemics.

According to (Reyers et al., 2022), resilience has reshaped sustainable development in six ways by 1) shifting focus from static capitals to **dynamic capacities**, 2) emphasizing **relational processes** over isolated objects, 3) prioritizing **adaptive processes** over fixed outcomes, 4) considering systems as **open and interconnected** rather than closed, 5) tailoring interventions to specific **contexts** rather than applying generic solutions, and 6) recognizing **complex causality** over linear cause-effect relationships. These shifts have led to innovative practices that better address the complexities of sustainability, although challenges remain in aligning practice with theoretical and methodological advancements in resilience science.

4.1.2 Resilience vs. dynamics

Resilience differs from merely being static or unchanging over time; resilient systems are often quite dynamic. Conversely, systems that remain constant over time can lack resilience. Acknowledging the **difference between static stability and resilience** is crucial (Meadows, 2009).

- **Static stability** is observable; it can be assessed by analyzing changes in the system's conditions over weeks or years.
- **Resilience**, on the other hand, is often only noticeable when the system is pushed beyond its limits and breaks down. Because resilience may not be apparent without a systems view, individuals frequently prioritize stability, productivity, or other more immediately observable characteristics over resilience.

For example, **just-in-time deliveries** of products to retailers and parts to manufacturers have minimized inventory fluctuations and lowered costs across various industries. Nonetheless, this mode of operation has rendered the production system more vulnerable to disruptions in fuel supply, computer failures, labor shortages, and other potential shocks.

Another example constitutes the **intensive management of European forests**. Over centuries, it has transformed native ecosystems into single-age, single-species plantations, frequently composed of nonnative trees. These plantations aim to produce wood and pulp consistently over time. However, these forests have lost their resilience without multiple species interacting with each other and their environment. As a result, we are witnessing their vulnerability to threats such as industrial air pollution and pests like the bark beetle.

4.1.3 Resilience in the sustainability sciences

Resilience as a metaphor related to sustainability   Resilience and sustainability are closely related concepts in the context of social-ecological systems (SES). Resilience refers to the capacity of a system to absorb disturbances, adapt to changes, and maintain its core functions and structures. Sustainability, on the other hand, is the ability to meet the needs of the present without compromising the ability of future generations to meet their own needs. A resilient system can be more sustainable because it can withstand and adapt to shocks and stresses, ensuring long-term stability and functionality. Therefore, enhancing resilience is often seen as a key strategy for achieving sustainability.

Resilience as a property of dynamic systems   Dynamic systems are those that change over time, often in response to internal or external stimuli. Resilience in these systems is about how well they can absorb shocks and continue to operate effectively. For example, an ecosystem might experience a natural disaster but still maintain its biodiversity and functionality.

Resilience as a measurable quantity   Resilience is considered a measurable quantity through various indicators and metrics that capture the capacity of systems to absorb disturbances, adapt to changes, and maintain functionality. In field studies of social-ecological systems (SES), resilience can be assessed using indicators related to ecological, social, economic, and institutional dimensions. These indicators help researchers quantify resilience and understand how different systems respond to various stressors and shocks.

It is important to **acknowledge these different meanings** of resilience when discussing sustainability and human-environment interactions.

4.1.4 Resilience of what to what?

The resilience of what to what is **a key question** when applying the concept of resilience to sustainability and human-environment interactions (Carpenter et al., 2001).

The **'of what'** refers to system function or configuration to be sustained, such as biodiversity, ecosystem services, social cohesion, or economic stability.

The **'to what'** refers to the disturbances, shocks, or changes that the system needs to withstand, such as climate change, natural disasters, economic crises, or social conflicts.

4.1.5 Specified vs. general resilience

Another key distinction in resilience research is between specified and general resilience (Folke et al., 2010).

Specified resilience refers to the resilience of a system function to specific challenges or disturbances (i.e., a narrowly defined *what* to *what*). *For example*, - a community might have specified resilience to flooding by building dams and flood protection walls; - a farmer might use pest-resistant crops

to increase resilience to specific pest infestations; or - an individual might get vaccinated against a particular disease to increase resilience to that disease.

Generalized resilience refers to the system's capacity to deal with the unknown, uncertainty, and surprise (i.e., a broadly defined *what* to *what*). *For example*, - a community or society with functioning institutions and social networks has generalized resilience to various shocks and stresses; - a framer with a healthy business model and diversified crops has generalized resilience to various economic and environmental changes; - a functioning immune system can provide generalized resilience to a wide range of diseases;

4.1.6 Example | Resilience of farming systems
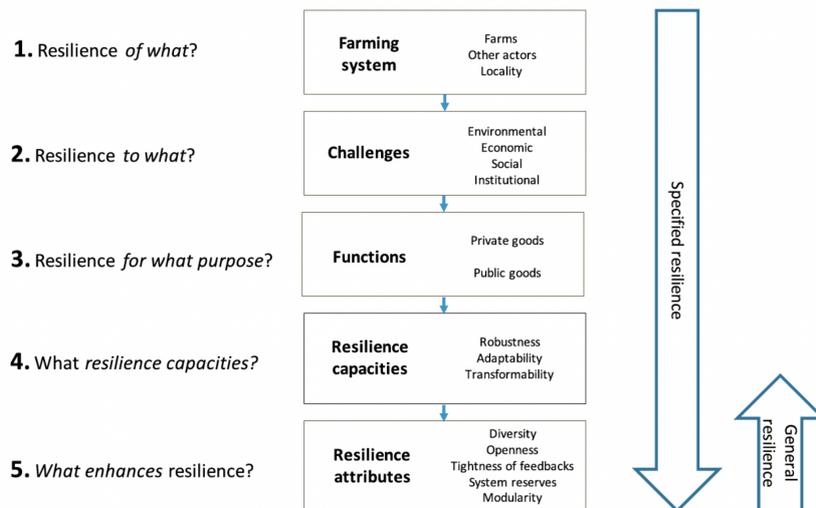
Fig. 2. Framework to assess resilience of farming systems.

Meuwissen et al. (2019) *A framework to assess the resilience of farming systems*

Figure 4.5: Framework to assess resilience of farming systems

4.1.7 Challenges

A lot of resilience scholarship utilizes **qualitative methods, case studies, and conceptual frameworks** (i.e., mental, verbal and pictorial models) to understand the dynamics of social-ecological systems. While these approaches are invaluable for generating diverse insights and hypotheses, they have **difficulty** in providing a **precise understanding** of resilience that allows for **quantitative predictions** and **generalizable results** in the sense of identifying universal system structures of relevance.

Here, the mathematics of stochastic dynamics and bifurcations can help.

4.1.8 Learning goals

After this lecture, students will be able to:

- **Explain** how **resilience concepts** related within the context of **sustainability science**.
- **Implement** and **simulate** nosiy dynamic system models to **illustrate different resilience types** using Python.
- **Quantify** changes in resilience to **measure when a systems approaches a tipping point**.

4.2 Resilience types

While resilience, in general, is defined as the capacity of a system to absorb disturbances, adapt to changes, and maintain functionality, it is useful to differentiate between three types of resilience that can be distinguished based on the system's response to stressors and shocks.

These three types are are often illustrated by ball-and-cup diagrams
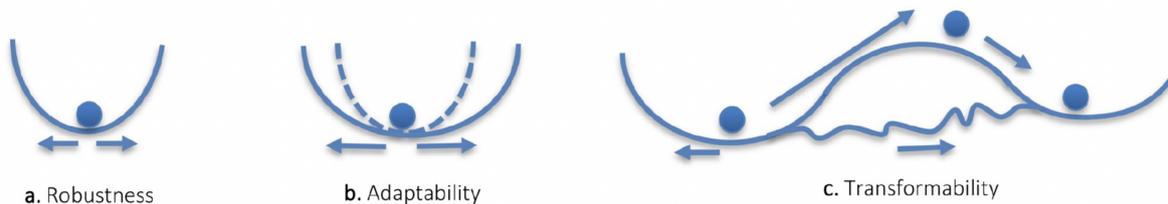
Fig. 4. Illustration of the three resilience capacities of farming systems (based on Holling et al., 2002).

Meuwissen et al. (2019) *A framework to assess the resilience of farming systems*

Figure 4.6: Resilience types

How to formalize these concepts?

We start by importing the necessary libraries and setting up the plotting environment.

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

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;
```

4.2.1 Robustness resilience

> The capacity to resist (or absorb) change and continue to function in its present state

Robustness resilience (sometime only called robustness (Anderies et al., 2013)) is the **most established** and **straightforward** type of resilience. It refers to the system's ability to resist or absorb changes and **continue to function in its present state**. A system with high robustness resilience can withstand disturbances and shocks without significant changes to its structure or function. This type of resilience is often associated with stability and persistence in the face of external stressors.

While it is the simplest form of resilience, it acknowledges that the **future is inherently unpredictable**. We cannot observe the current state to full precision. And we cannot process and extrapolate all the information and uncertainty. Therefore, robustness resilience is a key concept in the context of **uncertainty** and **complexity**.

Robustness | Ball-and-cup diagram

The **ball-and-cup pictorial model** of the robustness resilience portrays a fixed cup (reprensting the potential) and a ball (representing the system state)

Meuwissen et al. (2019) *A framework to assess the resilience of farming systems*

Figure 4.7: Robustness resilience

External shocks change the system state along the x-axis.

However, this pictorial model leaves crucial questions unanswered:

- How does the **system** state **change over time**?
- How large and frequent are the **shocks**?
- What happens if the system state **exceeds the cup**?

Converting this pictorial model into a **mathematical model** requires us to become **more specific**.

Robustness | System dynamics and potenial

We formalize the idea of having a single basin of attraction by the following difference equation,

$$\Delta x = x^3 - cx,$$

where c is a parameter that controls the system's stability and x is the system state. As a side note, this model is also known as the normal form of a subcritical pitchfork bifurcation in dynamical systems theory (see exercise on Tipping Elements).

Integrating the negative difference equation, we obtain the **potenial function** $G(x)$ by $\Delta x = -G(x)/dx$ as

$$G(x) = \frac{c}{2}x^2 - \frac{1}{4}x^4.$$

Converting the potential function into Python yields,

```
def G_robustness(x, c): return c/2*x**2 - x**4/4
```

We devise a function to plot the potential function, togehter with the system's equilibirum points and their stability (if $c > 0$, $x_e = 0$ is stable and $x_e = \sqrt{c}$ and $x_e = -\sqrt{c}$ are unstable; if $c < 0$, $x_e = 0$ is unstable; see the exercise on Tipping Elements).

```python
def plot_robustness_potential(c=2):
    xs=np.linspace(-2,2,101); plt.plot(xs, G_robustness(xs, c), '--', color='blue')
    plt.xlabel(r"System state $x$"); plt.ylabel(r"Potential $G(x)$")

    if c>0:  # draw fixed points
        xs=np.linspace(-np.sqrt(c), np.sqrt(c), 101); plt.plot(xs, G_robustness(xs,
↪   c), color='blue')
        plt.scatter(np.sqrt(c), G_robustness(np.sqrt(c), c), s=200,  c='w',
↪   edgecolor='blue')
        plt.scatter(-np.sqrt(c), G_robustness(-np.sqrt(c), c), s=200,  c='w',
↪   edgecolor='blue')
        plt.scatter(0, G_robustness(0, c), s=200,  color='blue')
    else:
        plt.scatter(0, G_robustness(0, c), s=200,  c='w', edgecolor='blue')
    plt.ylim(-0.2, 1.1)
```

```python
plot_robustness_potential()
```

We observe a **single basin of attraction** for the system state x. For $c > 0$, the unstable fixed points indicate where boundaries of the *cup* lie.

Robustness | Stochastic dynamics

To **account for shocks** or external changes to the system, we **refine the update equation** as follows,

$$x_{t+1} = F_N(x_t) = F_D(x_t) + n\eta_t = x_t + (cx_t + x_t^3) + n\eta_t.$$

The new stochastic or nosiy udpate equation $F_N(x_t)$ is composed of the orignial **deterministic map** F_D, **plus a stochastic random variable** η_t of mean zero. The paramter n regulates the **strength** of the noise term.

We model the shocks by a **normally distributed** random variable n_t with mean zero and unit variance. The corresponding Python function is,

```python
def F_robustness_noise(x, c, n): return x + x**3 - c*x + n*np.random.randn()
```

We define a plotting function to illustrate the system dynamics under stochasticity. It also shows the basin of attraction, i.e., the region where the system state converges to the stable fixed point under the purely deterministic dynamics.

```python
def plot_robustness_noise(noiselevel=0.1, c=1.0, xinit=0.5):
    iters=250
    params=dict(c=c, n=noiselevel)

    x = xinit  # re-storing the initial values
    trajectory = [x]  # container to store the trajectory
    for t in range(iters):  # looping through the iterations
        x_ = F_robustness_noise(x, **params)  # the ** notation extracts the dict.
↪   into the func. as parameters
```

```
        if np.abs(x)>3: break  # stop the simulation when x becomes too large
        trajectory.append(x_)  # storing the new state in the container
        x = x_  # the new state of the system `x_` becomes the current state `x`

    plt.plot(trajectory, 'k'); plt.xlabel('time steps'); plt.ylabel('system state');
↪   # makes plot nice
    plt.fill_between([0, iters], [-np.sqrt(c), -np.sqrt(c)],  [np.sqrt(c),
↪   np.sqrt(c)], color='blue', alpha=0.25)
    plt.xlim(0,250); plt.ylim(-3,3)
```

If the noise level is low, the system is resilient to shocks and remains in the basin of attraction.

```
plot_robustness_noise()
```

However, if the noise level is high, the system can escape the basin of attraction and diverge.

```
np.random.seed(42); # fixing the random seed for reproducibility
plot_robustness_noise(noiselevel=0.3)
```

The level of **resilience** of the system is the width between the unstable fixed points. This quantity gives the maximum magnitude of a shock the system can still tolerate.

However, resilience in **(social-)ecological systems** is **not always adequately described** by this form of resilience.

This has lead scholars to broaden the meaning of resilience.

Robustness | Real-world examples

Infrastructure and **technial systems**, such as bridges, buildings, and buildings, are often engineered for robustness with a **safety margin** to withstand natural disasters like earthquakes or hurricanes. For example, buildings in earthquake-prone areas are constructed with materials and designs that allow them to absorb and dissipate seismic energy, minimizing damage and maintaining structural integrity. Or an elevator can carry more weight than its maximum load capacity to account for unexpected situations. However, these systems are not able to adapt to changing conditions or recover from severe damage without external intervention.

Robust **software systems** are often designed to maintain functionality in the face of errors or unexpected inputs. However, this robustness might be achieved through **rigid error-handling mechanisms** that don't necessarily scale or adapt to the severity of the issue. For instance, in **cyber security**, multi-layered security protocols, such as encryption, two-factor authentication, and fraud detection algorithms, help maintain the security and reliability of digital systems. However, these systems might not be able to adapt quickly to new types of cyber threats or changing regulatory requirements without significant investment and effort.

In the **mobility sector**, the german car industry's adherence on **combustion engines** can also be seen as robustness resilience. The industry has been able to maintain its market share and profitability for some time despite increasing pressure to transition to electric vehicles. However, this robustness might not be sustainable in the long term as the industry faces challenges related to climate change, air pollution, and changing consumer preferences.

4.2.2 Adaptation resilience

Capacity of a system to adjust its responses to changing external drivers and continue developing within the current stability domain or basin of attraction

Adaptation | Ball-and-cup diagram

The **ball-and-cup pictorial model** of the adaptation resilience portrays a **variable** cup (reprensting the potential) and a ball (representing the system state). As before, shocks change the system state along the x-axis.
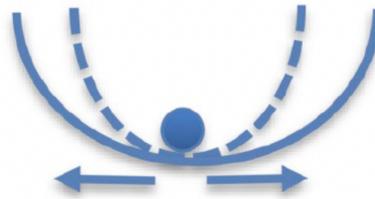
Meuwissen et al. (2019) *A framework to assess the resilience of farming systems*

Figure 4.8: Adaptation resilience

In adaptation resilience, the system can **adjust its responses** to changing external impacts. The capacity of a system to absorb shocks is **linked** to the strenght of the shocks. Adaptation resilience makes the resilience concept more flexible and adequate for (social-)ecological systems.

However, this pictorial model leaves the **crucial question** of **how** the system adjusts its responses unanswered.

Converting this pictorial model into a **mathematical model** requires us to become **more specific**. How could we convert the adaptation ball-and-cup diagram into a mathematical model?

Adaptation | System dynamics

We start from our previous dynamic systems model,

$$x_{t+1} = x_t + (x_t^3 - cx_t) + n\eta_t,$$

where x is the system state, c is the parameter that controls the system's stability, and n is the parameter that regulates the strength of the noise term η_t.

To link the system's responses to the external drivers, we introduce a **feedback mechanism** that adjusts the parameter c based on the **magnitude of the shock**. In other words, the parameter c becomes a **function of the shock's strength** $c(n)$.

$$x_{t+1} = x_t + (x_t^3 - c(n)x_t) + n\eta_t.$$

Adaptation | Feedback mechanism

The crucial question is how to **formulate the feedback mechanism** that adjusts the parameter c based on the magnitude of the shock n.

From a **stability analysis** of the subcritical pitchfork bifurcation, we know that the system is stable for $0 < c < 2$ (see Tipping Elements Exercise). Thus, we want the **maximal value** of $c(n)$ to be 2. This is an upper limit of how much noise the system can tolerate beyond which it cannot adapt anymore. We use the tanh function $\tanh(x)$ to achieve this, which results in values from -1 to 1. Thus we shift it up by 1.

Furthermore, we want the **minimal value** of $c(n)$ at $n = 0$ to have a base level b. This is a lower limit of how much noise the system can tolerate. Thus, we add b to the tanh function and devide the tanh-part by $(2 - b)/2$ to scale it to the interval $[b, 2]$.

Last, we model the location where the tanh function switches from b to 2 by the parameter l and controll the steepness of the transition by the parameter s.

Together, the feedback mechanism is formulated as,

$$c(a; b, s, l) = b + (1 + \tanh(s(a - l))) \frac{(2 - b)}{2}.$$

In Python, this function is implemented as,

```python
def cfunc(n, base, loc, steep): return base +
↳   ((1+np.tanh(steep*(n-loc))))*(2-base)/2
```

Visualizing the function c(a) for different parameters yields

```python
def plot_cfunc(n, base=0.25, loc=0.5, steep=5):
    plt.plot(n, cfunc(n, base, loc, steep), label=f"b={base}, l={loc}, s={steep}")
    plt.xlabel(r"Noise level $n$"); plt.ylabel(r"Control function $c(n)$")
```

```python
n = np.linspace(0, 1.5, 101)
plot_cfunc(n, base=0.6, loc=0.6, steep=4)
plot_cfunc(n, base=0.4, loc=0.8, steep=8)
plot_cfunc(n, base=0.2, loc=1.0, steep=12)
plt.ylim(-0.05, 2.05); plt.legend();
```

We include this feedback mechanism in the dynamic systems update,

```python
def F_adaptation_noise(x, n, base, loc, steep):
    return x + x**3 - cfunc(n,base,loc,steep)*x + n*np.random.randn()
```

and define a plotting function which illustrates the feedback mechansim of how the control parameter c responds to the noise strength a together with the time evolution of the system under stochasticity.

```python
def plot_adaptation_noise(noiselevel=0.01, base=0.75, loc=0.5, steep=5.0):
    iters=250; xinit = 0.5; ylim=(-1.5, 2.01)
    params=dict(n=noiselevel, base=base, loc=loc, steep=steep)

    fig = plt.figure(figsize=(10, 4))
    basinstyle = {'color':'blue', 'alpha':0.25}

    plt.subplot(1,2,1)
    plt.plot(n, cfunc(n, base, loc, steep), label='$c(n)$', color='green')
    plt.plot(n, np.sqrt(cfunc(n, base, loc, steep)), label='$\sqrt{c(n)}$',
↪    **basinstyle)
    plt.plot(n, -np.sqrt(cfunc(n, base, loc, steep)), **basinstyle)
    plt.plot([noiselevel, noiselevel], [-2, 2], 'k--')
    plt.ylim(ylim); plt.xlabel('Noise level $n$');
    plt.legend()

    plt.subplot(1,2,2)
    x = xinit  # re-storing the initial values
    trajectory = [x]  # container to store the trajectory
    for t in range(iters):  # looping through the iterations
        x_ = F_adaptation_noise(x, **params)  # the ** notation extracts the dict.
↪    into the func. as parameters
        if np.abs(x)>3: break  # stop the simulation when x becomes too large
        trajectory.append(x_)  # storing the new state in the container
        x = x_  # the new state of the system `x_` becomes the current state `x`

    plt.plot(trajectory, 'k');
    plt.xlabel('Time steps $t$'); plt.ylabel('System state $x$');
    cval = cfunc(**params)
    plt.fill_between([0, iters], [-np.sqrt(cval), -np.sqrt(cval)],
                    [np.sqrt(cval), np.sqrt(cval)], **basinstyle)
    plt.xlim(0,250); plt.ylim(ylim)
```

For a small noise level n, the system is resilient to shocks and remains in the basin of attraction, independent of the designed feedback mechanism.

```python
plot_adaptation_noise(noiselevel=0.01, base=0.5, loc=0.5, steep=5.0)
```

For large noise leves, the location l where the feedback mechanism kicks in becomes crucial. If l is too large, the system cannot adapt to the shocks and diverges.

```python
np.random.seed(42); # fixing the random seed for reproducibility
plot_adaptation_noise(noiselevel=0.35, base=0.5, loc=0.4, steep=5.0)
```

Decreasing the location l allows the system to adapt to the shocks and remain in the basin of attraction.

```python
np.random.seed(42); # fixing the random seed for reproducibility
plot_adaptation_noise(noiselevel=0.35, base=0.5, loc=0.2, steep=5.0)
```

It is important to note, that the way we **implemented** adaptation resilience is just one of **many possible ways** to make this concept more precise. Instead of widening the basin of attraction with increased noise levels, the location of the basins minimum could be shifted gradually to areas with less noise.

Adaptation resilience makes the resilience concept more flexible and adquate for (social-)ecological systems. However, sometimes, a system response to shocks by a complete **reorganization**, instead of just absorbing a shock.

Adaptation | Real-world examples

Natural ecosystems are being used as part of adaptation strategies to enhance resilience. For example, **coastal mangrove forests** show adaptation to sea level rise and storm surges. As water levels increase, mangroves accumulate sediment and organic matter to elevate their root systems, allowing them to keep pace with gradual sea level changes. This **natural adaptation** helps protect coastlines from erosion and storm damage (UNEP).

Even **infrastructure** can be designed to adapt to changing conditions. Instead of building higher and more robust defenses, the Netherlands adopted a "**Room for the River**" strategy (Dutch Water Sector). The key idea is *to restore the river's natural flood plain in places where it is least harmful in order to protect those areas that need to be defended.*, i.e., to live with the water instead of fighting it: the strategy includes the *lowering the levels of flood plains, creating water buffers, relocating levees, increasing the depth of side channels, and the construction of flood bypasses.*

In the **mobility sector**, **electric cars** can be seen as another example of adaptation resilience. As the world shifts towards sustainable energy sources, electric vehicles are becoming more popular, replacing fossil fuel-powered cars. Yet, while the transition to electric vehicles requires a significant change in infrastructure, including charging stations, battery production, and recycling facilities, the dominance of private cars as a mode of transportation remains largely unchanged.

4.2.3 Transformation resilience

> Capacity to create a fundamentally new system when ecological, economic, or social structures make the existing system untenable

Transformation | Ball-and-cup diagram

The **ball-and-cup pictorial model** of the transformation resilience portrays **multiple** cups (reprensting the potential) and a ball (representing the system state). As before, shocks change the system state along the x-axis.
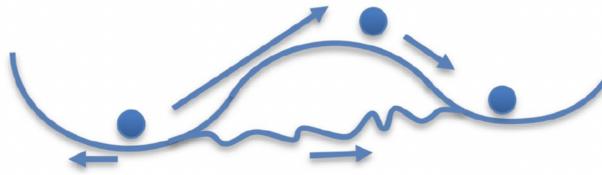
Meuwissen et al. (2019) *A framework to assess the resilience of farming systems*

Figure 4.9: Transformation resilience

In transformation resilience, the system can **reorganize into a fundamentally new regimes, or state** when the existing system state or regime becomes untenable. Thus, in contrast to the other resilience types, transformation resilience conceptualized one existing state and at least one new state. There must be at least two basins of attraction.

However, this pictorial model leaves the **crucial questions** unswered **how the basins of attraction are shaped**, in addition to how the system state changes over time and how large and frequent the shocks are.

Converting this pictorial model into a **mathematical model** requires us to become **more specific**.

How could we convert the transformation ball-and-cup diagram into a mathematical model?

We need a dynamical system with multiple stable states.

Transformation | Alternative-stable-states system

We refine the system from the lecture on Tipping Elements with the difference equation,

$$\Delta x = (x - ax^3 + c + n\eta)\frac{1}{\tau},$$

where η represents the noise term with mean zero and n the strength of the stochasticity. As before, τ represents the typical time scale of the system, and thus, inverse strength of the system's change, and a is a parameter that determines the strength of the balancing feedback loop in relation to the reinforcing feedback loop (with unit stength). The parameter c represents the external driver that can push the system over the tipping point.

Again, we model the shocks by a normally distributed random variable η_t with mean zero and unit variance. The corresponding Python function is,

```python
def F_tipmod_noise(x, drive, shape=1, timescale=0.1, noiselevel=0):
    return x + (x - shape*x**3 + drive + noiselevel*np.random.randn())/timescale
```

We define a plotting function to illustrate the system dynamics over time under stochasticity. We set the default values for the shape parameter $a = 1$ and the timescale paramter $\tau = 2$.

```python
def plot_transformation_trajectory(drive=-0.3, shape=1, timescale=2, noiselevel=0.0,
↪   initalstate=0.5):
    iters=2500; params=dict(drive=drive, shape=shape, timescale=timescale,
↪   noiselevel=noiselevel)

    x = initalstate  # re-storing the initial values
    trajectory = [x]  # container to store the trajectory
    for t in range(iters):  # looping through the iterations
        x_ = F_tipmod_noise(x, **params)
        if np.abs(x)>3: break  # stop the simulation when x becomes too large
        trajectory.append(x_)  # storing the new state in the container
        x = x_  # the new state of the system `x_` becomes the current state `x`

    plt.plot(trajectory, 'purple'); plt.ylim(-1.4, 1.4)
    plt.xlabel('Time steps $t$'); plt.ylabel('System state $x$');  # makes plot nice
    return np.array(trajectory);
```

With the right system characteristics (i.e., its parameters) we observe a noise induced transition between the two stable states.

```python
np.random.seed(0); # fixing the random seed for reproducibility
plot_transformation_trajectory(drive=-0.3, noiselevel=0.15);
```
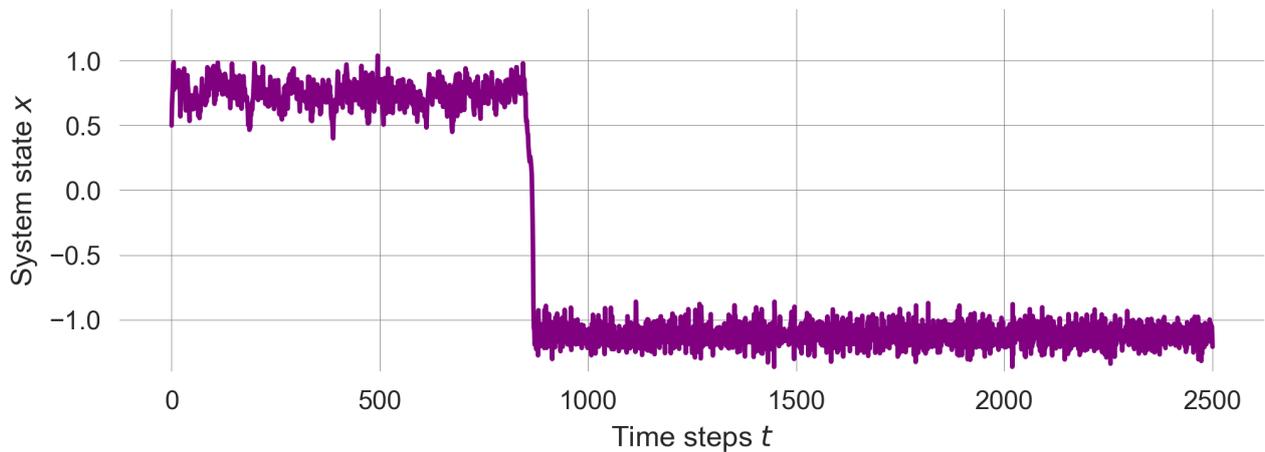

Without noise, the system converges and remains to the positive equilibrium point.

```
plot_transformation_trajectory(drive=-0.3, noiselevel=0.0);
```
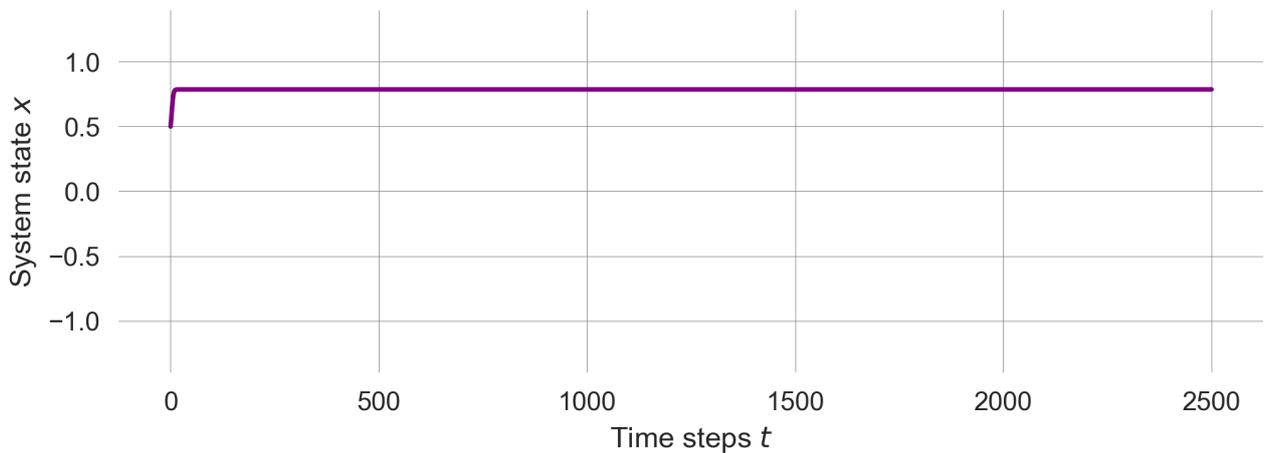

Under which conditions does the system transition between the two stable states as a result of the random shocks?

How can we understand better under what conditions the system switches between the two stable states under stochastic shocks?

Transformation | Potential function

We make use of the potential function (see Tipping Elements) to improve our understanding of the system dynamics. As a reminder, the potential function $G(x)$ is defined as the negative integral of the system change Δx. Thus, for the difference equation $\Delta x = \frac{1}{\tau}(x - ax^3 + c)$, we have

$$G(x) = -\frac{1}{\tau}\left(\frac{1}{2}x^2 - \frac{1}{4}ax^4 + cx\right).$$

In Python, we have,

```python
def G_tipmod(x, drive, shape, timescale): return - (x**2/2 - shape*x**4/4 +
↪   drive*x)/timescale
```

To visualize the potential function, we define

```python
def plot_tipmod_potential(drive=-0.3, shape=1.0, timescale=2):
    xs=np.linspace(-2,2, 501); plt.ylim(-0.5, 0.5);
    plt.plot(xs, G_tipmod(xs, drive, shape, timescale), color='k')
    plt.ylabel(r'Potential $G(x)$'); plt.xlabel(r'System state $x$')

    #  numerically find and plot equilibrium points
    drive_ = shape*xs**3 - xs
    xeq = xs[np.isclose(drive_-drive, 0.0, atol=0.02)]
    plt.plot(xeq, G_tipmod(xeq, drive, shape, timescale), 'o', ms=12, color='k')
```

```
plot_tipmod_potential()
```
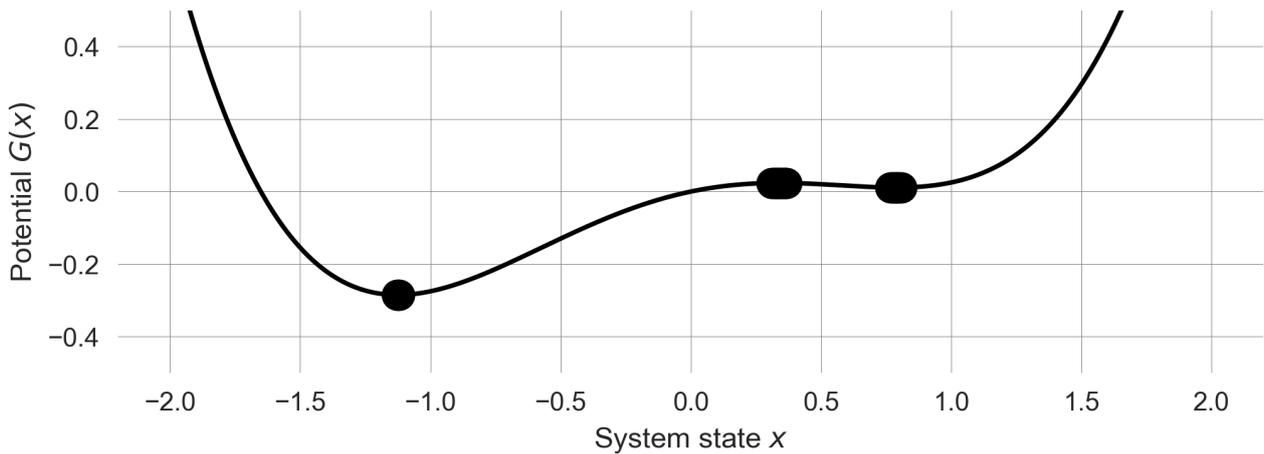
Transformation | Bifurcation diagram

We also visualize the bifurcation diagram (see Tipping Elements) to understand the system's stability and the location of the tipping point.

```python
def plot_bifurcation_tipmod(shape=1.0, timescale=2.0, cextent=[-1.4, 1.4]):
    xe=np.linspace(*cextent, 1001) # equilibrium points
    driver = shape*xe**3 - xe # parameter c
    plt.plot(driver, xe, "--", color='k'); # equilibrim point

    # stability
    def F_(x, shape, timescale): return 1 + (1-3*shape*x**2)/timescale
    cond=np.logical_and(F_(xe, shape, timescale)<1, F_(xe, shape, timescale)>-1)
    plt.plot(driver[cond], xe[cond], ".", c='green')

    plt.xlabel(r'External driver $c$'); plt.ylabel(r'Equilibrium points $x_e$');
    plt.xlim(cextent);
```

```python
plot_bifurcation_tipmod()
```
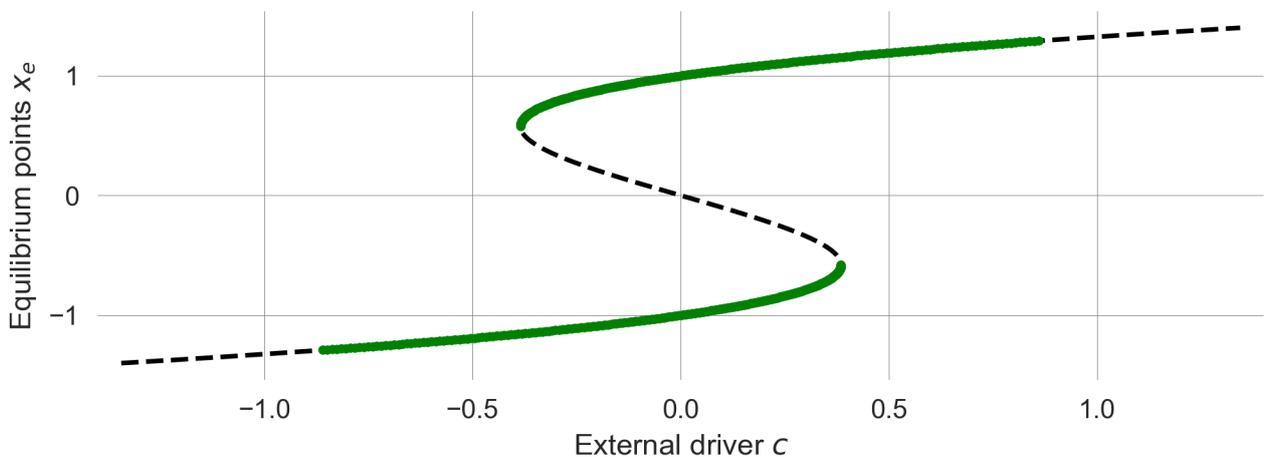
4.2.4 Transformation | Combined analysis

Putting all together shows us how potential, bifurcation diagramm and the noisy trajectories interact.

```python
def plot_tranformation(drive=-0.3, shape=1, timescale=2, noiselevel=0.0,
↪   initalstate=0.5):
    fig = plt.figure(figsize=(10,5))

    ax1 = fig.add_subplot(2,2,1)
    plot_tipmod_potential(drive=drive, shape=shape, timescale=timescale)

    ax2 = fig.add_subplot(2,2,2)
    plot_bifurcation_tipmod(shape=shape, timescale=timescale)
    ax2.plot([drive , drive], [-1.5, 1.5], 'k-') # include driver value
    ax2.set_xlim(-0.5, 0.5)

    ax3 = fig.add_subplot(2,1,2)
    traj = plot_transformation_trajectory(drive=drive, shape=shape,
        timescale=timescale, noiselevel=noiselevel, initalstate=initalstate)

    # include trajectory in the potential
    ax1.scatter(traj, G_tipmod(traj, drive=drive, shape=shape, timescale=timescale),
↪
                alpha=0.5, s=40, c=np.arange(len(traj)), cmap='plasma_r',zorder=10)

    # include trajectory in the bifurcation diagram
    ax2.scatter(np.ones_like(traj)*drive, traj,
                alpha=0.5, s=40, c=np.arange(len(traj)), cmap='plasma_r',zorder=10)

    plt.tight_layout()
```

Now, we can reanalyze our sitation from above. **When there is one fixed point dominating (i.e., having a larger basin of attraction), we observe a noise-induced transition to that fixed point**. The timeseries is shown in the potential and the bifurcation diagram with time going from light to dark colors.

```python
np.random.seed(0); plot_tranformation(drive=-0.3, noiselevel=0.15)
```
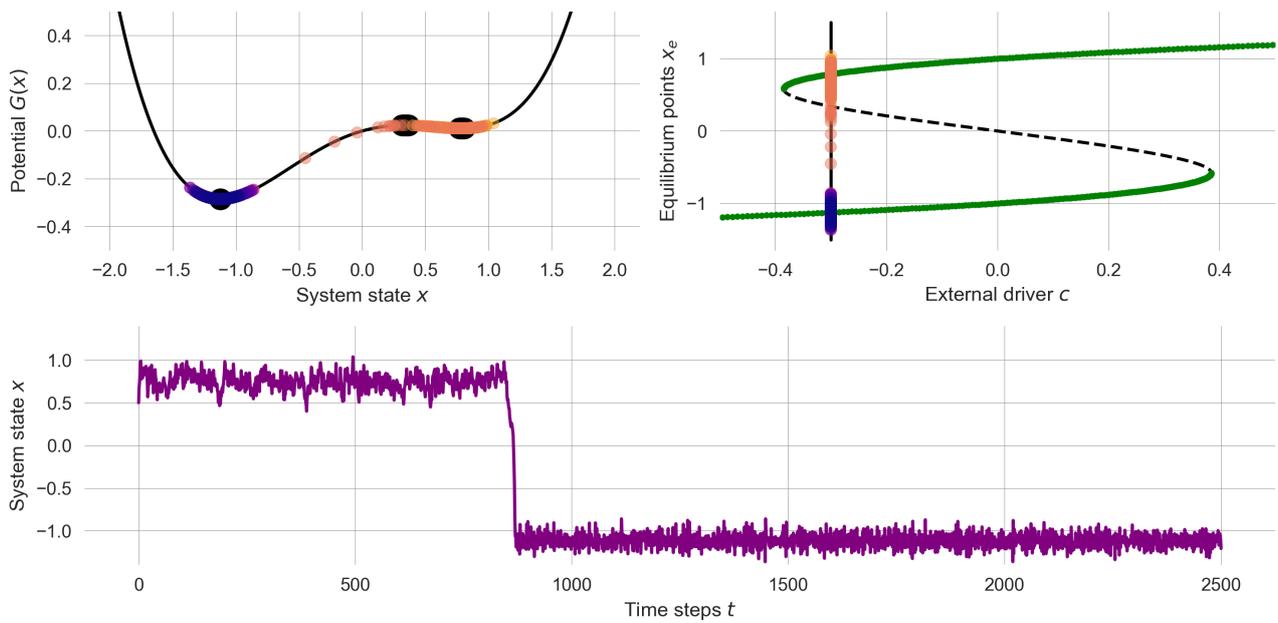
Where there is only one stable fixed point, also the stochastic system will evolve around that state.

```
plot_tranformation(drive=-0.42, noiselevel=0.15)
```
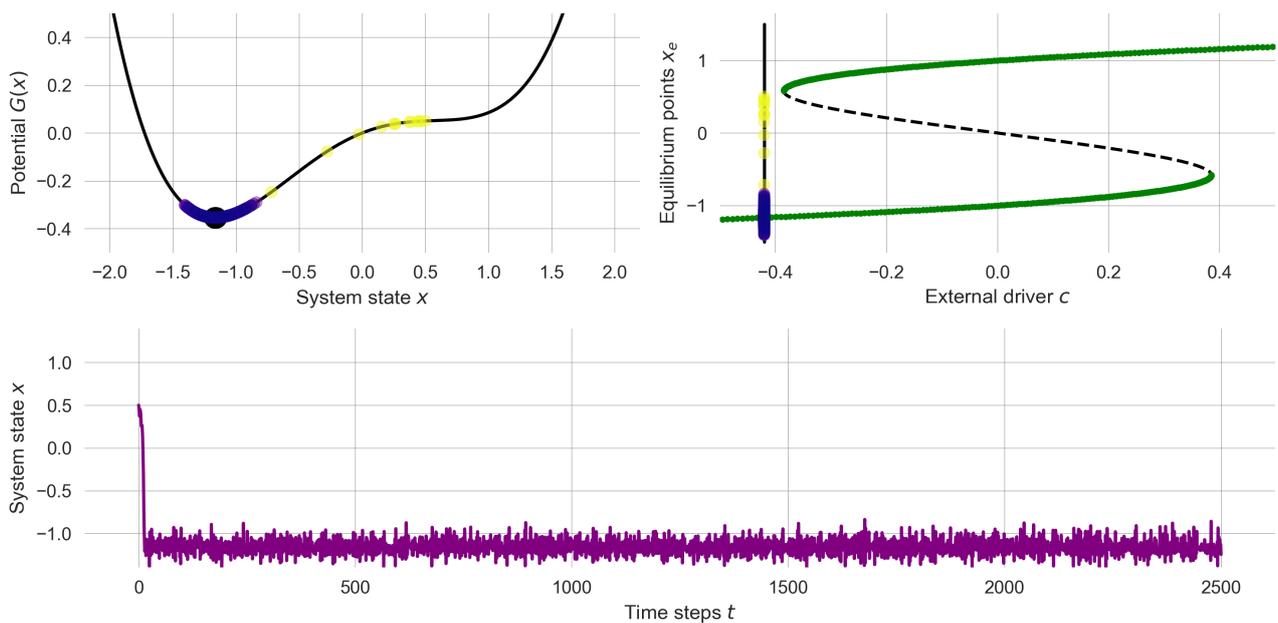

We observe bistable flickering when both fixed points are stable and the noise is not too small and not too large.

```
np.random.seed(0); plot_tranformation(drive=0, noiselevel=0.45)
```
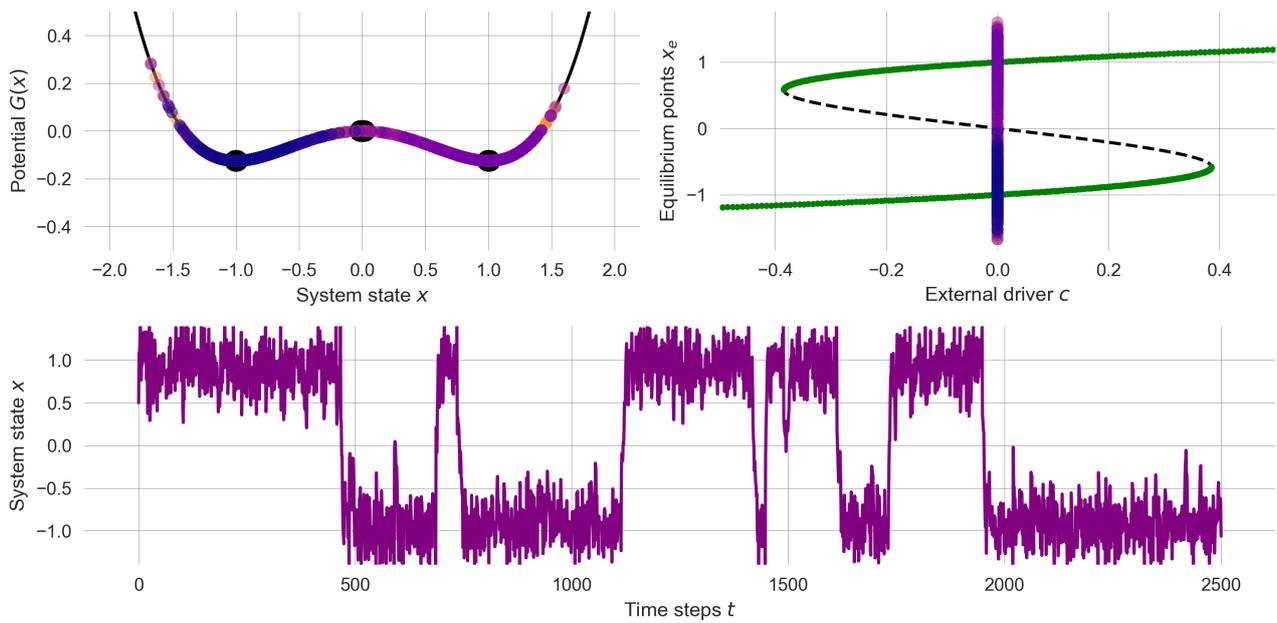
These results show that richness of phenomena that our relativly simple model can explain.

4.2.5 Transformation | Real-world examples

In the **mobility sector**, moving away from system where private cars dominate to a system where public transportation, cycling, and walking are the primary modes of transportation can be seen as an example of transformation resilience. This shift requires a fundamental reorganization of the transportation system, including changes in infrastructure, policies, and social norms. While this transformation is challenging, it can lead to significant benefits, such as reduced traffic congestion, air pollution, and greenhouse gas emissions.

In the **energy sector**, transitioning from fossil fuels to renewable energy sources, such as solar, wind, and hydropower, is another example of transformation resilience. This shift requires a fundamental reorganization of the energy system, including changes in energy production, distribution, and consumption. While this transformation is complex and costly, it can lead to significant benefits, such as reduced greenhouse gas emissions, air pollution, and dependence on finite resources.

In the **agriculture sector**, transitioning from conventional farming practices to regenerative agriculture is another example of transformation resilience. This shift requires a fundamental reorganization of the food system, including changes in farming methods, land use, and food production. While this transformation is challenging, it can lead to significant benefits, such as improved soil health, biodiversity, and food security.

4.3 Quantifying resilience

So far, we **conceptualized multiple facets of resilience** through simple dynamical system models. While these models are crucial for understanding the underlying mechanisms of resilience, it is difficult to apply these models to real-world systems directly.

For instance, critical questions around resilience, tipping elements, and regime shifts of real-world systems is **how resilient is the system?** and **how far away is the system from a tipping point?**. These questions are challenging to answer in empirical systems because we cannot and do not want to trigger the regime shift to find out. Tipping elements are often **hidden** and **uncertain**.

In the following, we will discuss how we can **quantify resilience** and thus **measure whether a system is approaching a tipping point**, based on the conceptual models we discussed above.

4.3.1 Critical slowing down

Critical slowing down describes the phenomenon that the **internal time scale a system** operates on **increases**, when the system **approaches as tipping point**. As a consequence, a system close to a tipping point tends to undergo **larger changes in response to perturbations** and takes longer to recover from them.

How can we capture this phenomenon using a **quantitative model**?

Let's reuse the robustness resilience model, the deterministic subcritical pitch-fork bifurcation, $\Delta x = x^3 - cx$, to illustrate that phenomenon.

```
plot_robustness_potential()
```
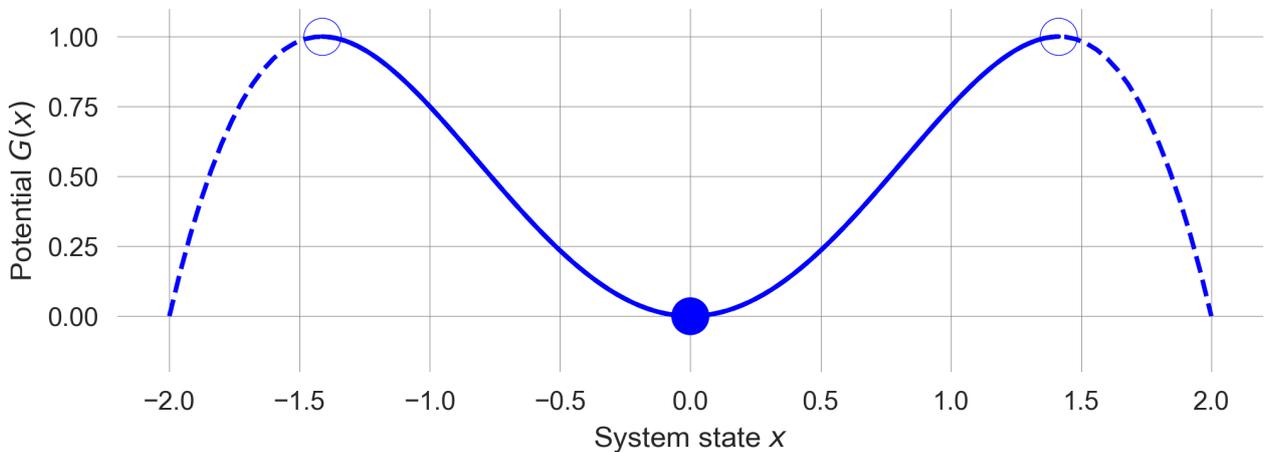

We know, the fixed point at $x_e = 0$ is stable for $c > 0$. Its basin of attraction extends from $-\sqrt{c}$ to $\sqrt{c}$.

We want to simulate how long in takes on average from all points in the basin of attraction to reach the fixed points.

To do so, we have to **define** a notion of **convergence**.

Let's look an exemplary trajectory:

```
c = -1.2; x = 1.0
trajectory = [x]
for _ in range(100):
    x_ = F_robustness_noise(x, c=1.2, n=0) # n=0 noiseless
    trajectory.append(x_)
    x = x_
trajectory = np.array(trajectory)
plt.plot(trajectory);
```
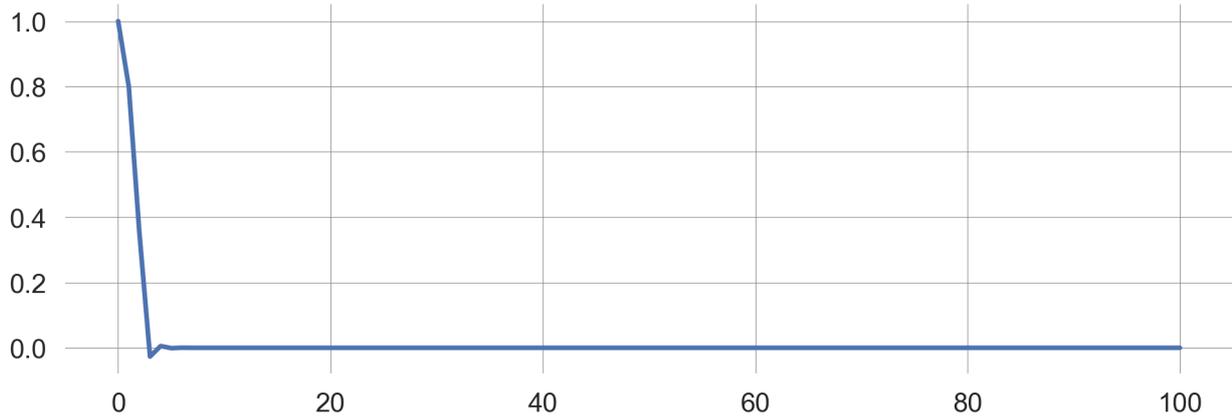
When it reached the equilibirum, it does not change any further.

Investigating the change between timesteps on a logarithmic axis,

```
plt.plot(np.abs(trajectory[0:-1] - trajectory[1:])); plt.yscale('log');
```
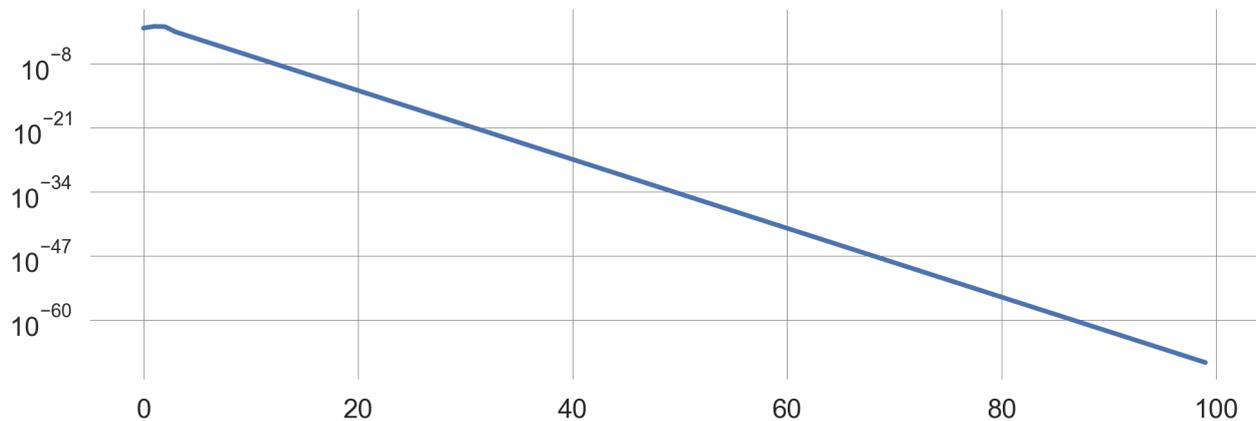

we find, the changes do become smaller.

But it is sufficient for us to set a **threshold tolerance level**, below which we want to consider a trajectory as converged. Let's use 10^{-9}.

```
def simulate_trajectory(xinit, c, threshold=10e-9, maxiter=10000):
    x = xinit
    trajectory = [x]
    for _ in range(maxiter):
        x_ = F_robustness_noise(x, c, n=0)
        trajectory.append(x_)
        if np.abs(x-x_) < threshold:  # <-- HERE
            break
        x = x_
    return np.array(trajectory)
```

Note, we still keep a maximum number of iterations to not get stuck here, should the threshold never be reached.

Now, plotting the simulated trajectory, we observe that it automatically stopped when the threshold was reached.

97

```
plt.plot(simulate_trajectory(xinit=1.0, c=1.2));
```
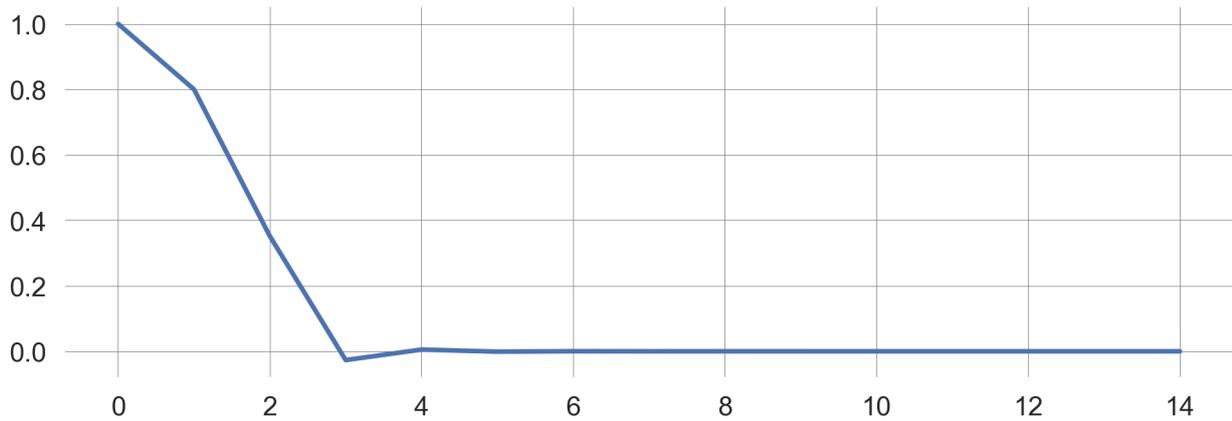

Using this threshold, we can set up our simulation,

```
eps = 10e-9  # The numberical threshold to be use for a small quantity
cvs = np.linspace(0+eps, 0.8, 501)  # The external parameters a to be varried

average_lens = []
for c in cvs:
    xs = np.linspace(-np.sqrt(c)+eps, np.sqrt(c)-eps, 101)
        # starting slightly off the unstable fixed points
    lens = [len(simulate_trajectory(xinit, c, threshold=eps,
                                    maxiter=10000)) for xinit in xs]
    assert max(lens) < 10000
    average_lens.append(np.mean(lens))
```

to demonstrate the phenomenon of **critical slowing down**,

```
plt.plot(cvs, average_lens, '.'); plt.ylabel('<Timesteps to convergence>');
↪   plt.xlabel('External drive $c$');
```
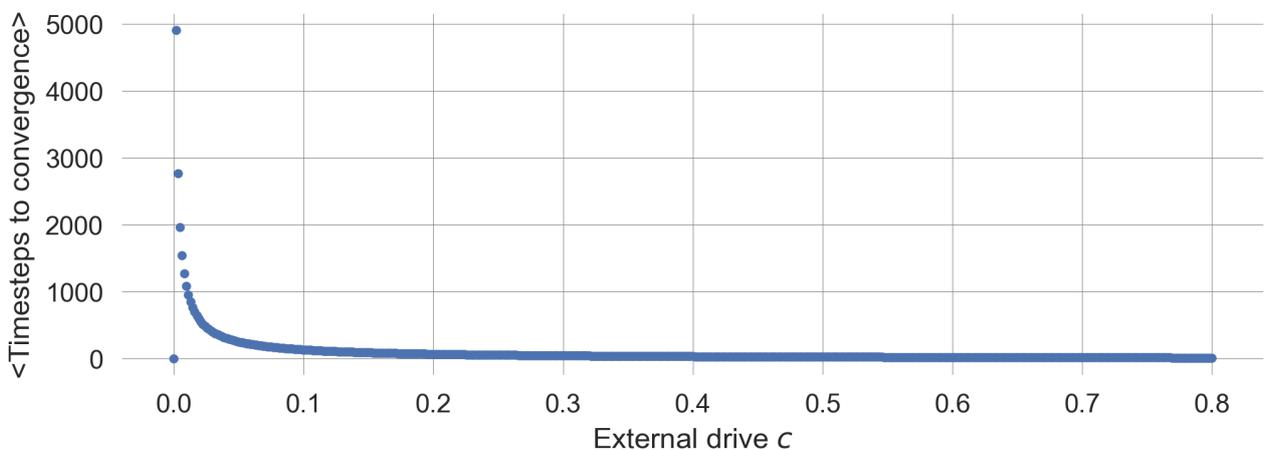

When we approach the tipping point at $c = 0$, the average number of timesteps it takes to converge to the equilibirum increases sharply.

4.3.2 Early-warning signals

Next, we use the phenomenon of critical slowing down as a **sign of resilience loss** to create an early-warning signal. Early-warning signals are important because they allow us to **anticipate critical transitions before they occur**.

Early-warning signals are based on **statistical indicators** of the system behavior. Specifically, we will use the **autocorrelation** of the system's time series. Autocorrelation is the correlation of a signal with a delayed copy of itself as a function of the delay. It measures the degree of similarity between a given time series and a lagged version of itself.

Conceptual model

Let's reuse the our tipping elment model showing alternative stable states,

$$\Delta x = (x - ax^3 + c + n\eta)\frac{1}{\tau},$$

where η represents the noise term with mean zero and n the strength of the stochasticity. As before, τ represents the typical time scale of the system, and thus, inverse strength of the system's change, and a is a parameter that determines the strength of the balancing feedback loop in relation to the reinforcing feedback loop (with unit stength). The parameter c represents the external driver that can push the system over the tipping point.

We create a synthetic time series, along which we slighlty **reduce the resilience** of one equilibirum by changing the drive parameter c **with each iteration step**.

```
xinit=-1.1  # the system's initial condition
iters=1000  # how long to simulate
params=dict(b=0.5,c=0.5,d=0.05) # other parameter values

np.random.seed(0)  # fixing the random seed to make this reproducible
c=0  # initial value of the 'resilience' parameter
trajectory = []  # container to store the trajectory
x = xinit  # re-storing the initial values
for t in range(iters):  # looping through the iterations
    x_ = F_tipmod_noise(x, drive=c, shape=1, timescale=2, noiselevel=0.25)

    # F(x, a=a, **params)  # the ** notation extracts the dict. into the func. as
    ↪  parameters
    if np.abs(x)>3: break  # stop the simulation when x becomes too large
    trajectory.append(x_)  # storing the new state in the container
    x = x_  # the new state of the system `x_` becomes the current state `x`
    c += 0.00038  # we slightly increase c (i.e., reduce the resilience)
```

```
plt.plot(trajectory, 'k'); plt.xlabel('time step t'); plt.ylabel('system state x');
↪  # makes plot nice
```
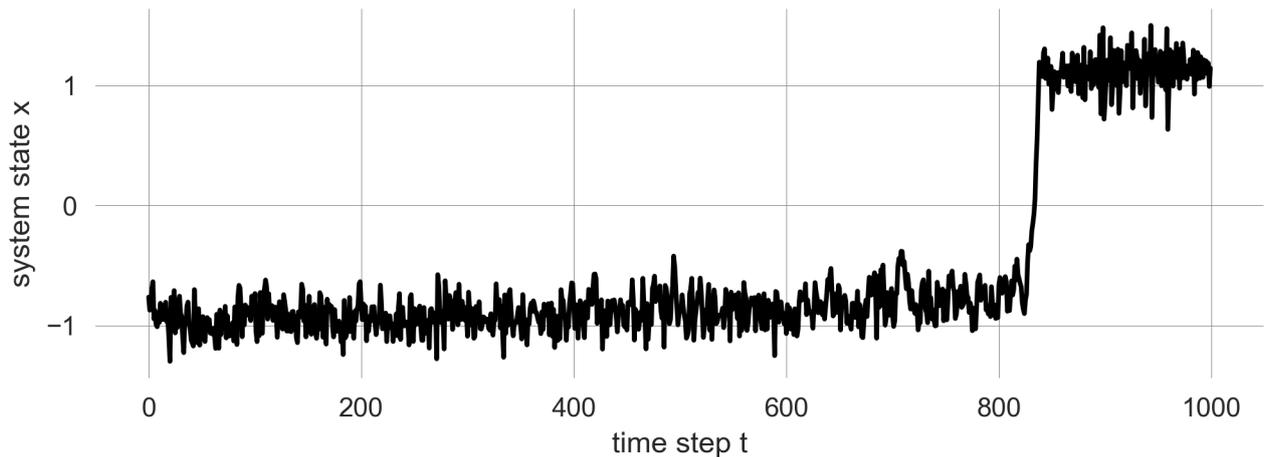
Now imagine, that we do not know the underlying model, but only have the time series. How can we detect the loss of resilience?

Scatter plot

To **visualize the autocorrelation**, (lag-1 temporal autocorrelation or AR(1) to be specific), we create a scatter plot of 200 points of our time series versus the 200 points of our time series at the next time step (lag-1). In this case, 200 is the size of our data **window**.

```
def scatter_autocorrelation(start=0):
    fig, axs = plt.subplots(1,2, figsize=(12,2.8))  # creates the two axes
    axs[0].scatter(trajectory[start:start+200], trajectory[start+1:start+201], s=10,
↪   c='blue');
    axs[0].set_xlabel(r'$x_{t+1}$'); axs[0].set_ylabel(r'$x_{t}$'); # makes first
↪   axis nice

    axs[1].plot(trajectory, 'k');  # plot the time series
    axs[1].fill_betweenx([-1.25, 1.5], [start, start], [start+200, start+200],
↪   color='blue', alpha=0.5) # show window
    axs[1].set_xlabel('time steps'); axs[1].set_ylabel(r'system state $x$');
↪   axs[1].set_xlim(0, 1000) # makes axis nice
```

How do we see the **autocorrelation** in this plot?

```
scatter_autocorrelation(start=0)
```

The autocorrelation is the **correlation** between the time series and a **lagged version** of itself. Thus, it is the correlation between the x-axis and the y-axis of the scatter plot on the left.

Then we **slide** our window of 200 system points **through our time series**. How does the scatter plot change? And what does the mean for the autocorrleation?

```
scatter_autocorrelation(start=310)
```
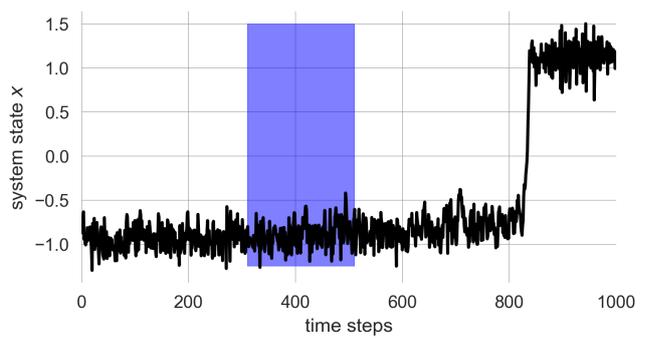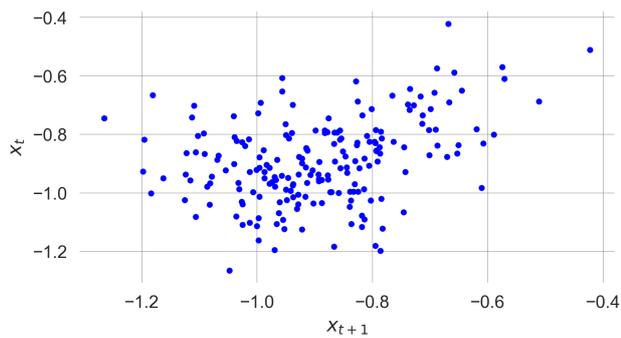
```
scatter_autocorrelation(start=620)
```
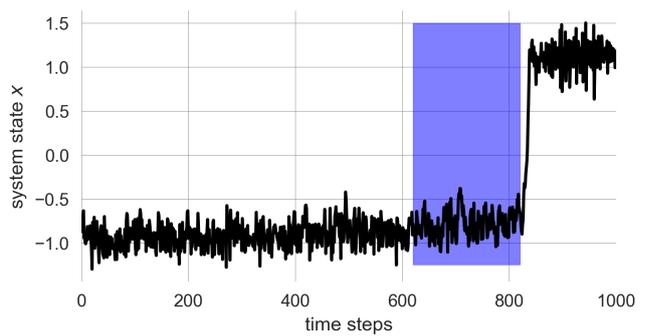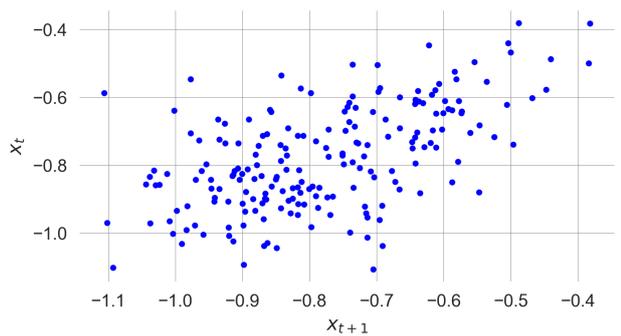

We visually can tell that the autocorrelation increases as we approach the tipping point.

Autocorrelation

To quantify our visual understanding, we finally calculate the lag-1 temporal autocorrelation.

For that, we use the `numpy.corrcoef` function. Thus, the **correlation matrix** between the time series points from index 0 to 200 and from index 1 to 201 is given by,

```
np.corrcoef(trajectory[0:200], trajectory[1:201])
```

```
array([[1.        , 0.20713096],
       [0.20713096, 1.        ]])
```

Thus, we have to **extract one of the off-diagonal elements**,

```
np.corrcoef(trajectory[0:200], trajectory[1:201])[0,1]
```

```
0.2071309613708747
```

Sliding through the time series from the beginning until the 600nd time step,

```
AR1 = [np.corrcoef(trajectory[start:start+200], trajectory[start+1:start+201])[0,1]
       for start in range(0,599)]
```

```
plt.plot(AR1); plt.ylabel('AR(1)'); plt.xlabel('Time step $t$');
```
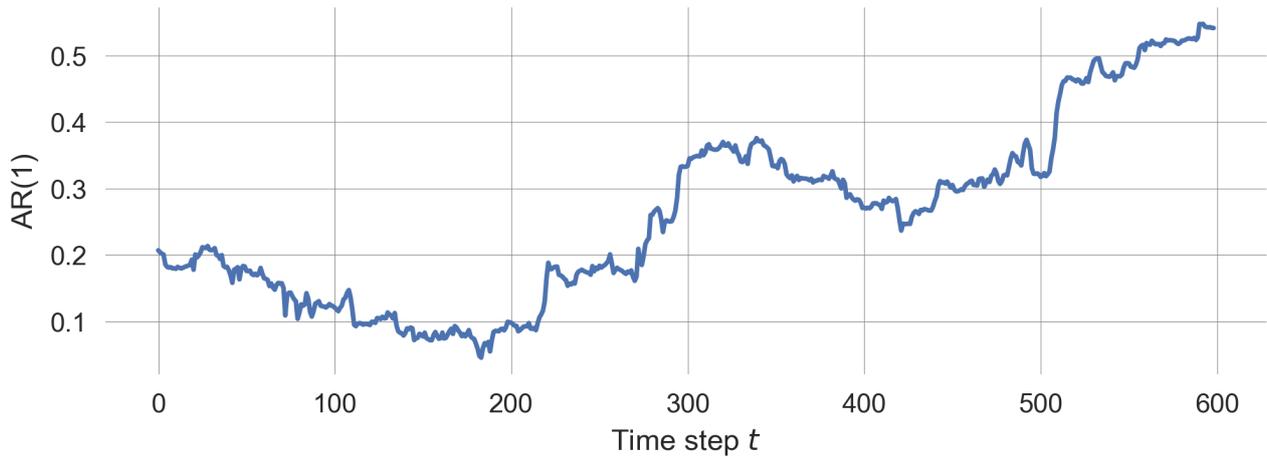

shows a clear rise of the lag-1 autocorrelation when approching the tipping point, **indicating a loss of resilience**.

This method **can be used on time series data only**. It does not require knowledge about the exact systems equation.

4.3.3 Example | Greenland Ice Sheet

A detected critical slowing down of its melt rates suggests that the western Greenland Ice Sheet is close to a tipping point (Boers & Rypdal, 2021).
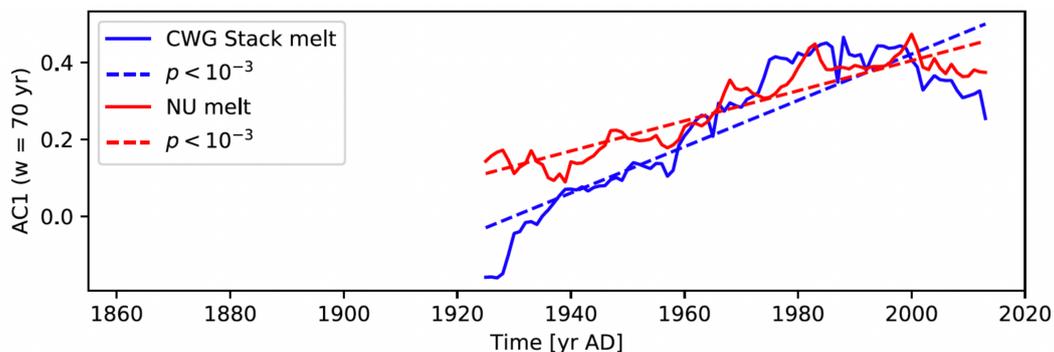
Boers & Rypdal (2021) *Critical slowing down suggests that the western Greenland Ice Sheet is close to a tipping point*

Figure 4.10: Critical slowing down in the Greenland Ice Sheet

4.4 Learning goals revisited

In this chapter, we have explored the concept of resilience in the context of sustainability science and human-environment interactions.

We have studied different types of resilience, including robustness, adaptation, and transformation resilience, and how they can be modeled using dynamic systems theory.

Last, we have examined how resilience can be quantified using early-warning signals based on the phenomenon of critical slowing down as indicators of system stability and resilience. By understanding these concepts and methods, we can better assess the resilience of social-ecological systems and anticipate critical transitions before they occur.

5 State transitions

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

5.1 Motivation | State-transitions models

State-and-transition models are useful tools to explain the causes and consequences of ecosystem change.

Figure 5.1 shows a state-transition model of sandy-loamy alluvails soils in the dry steppe of eastern central Mongolia.
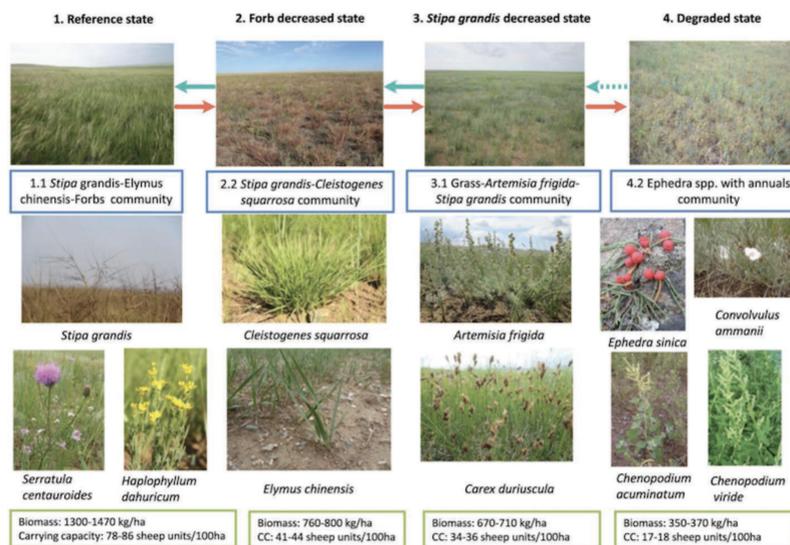
Bestelmeyer et al. (2021) *State-and-transition modelling* in Biggs et al. (2021) *The Routledge Handbook of Research Methods for Social-Ecological Systems*

Figure 5.1: State transitions in the dry steppe of Mongolia

The model describes the dynamics of the vegetation in the region. The vegetation can be in one of four states: a *reference state*, *Forb decreased state*, *Stipa grandis decreased state*, or *Degraded state*. The arrows indicate the possible transitions between the states (Biggs et al., 2021).

Figure 5.2 shows possible state transitions between states of different land cover types in the Brazilian Amazon.

The thickness of the arrows indicates the probability of the transition. The land cover can be in one of five states: *Annual crops*, *Forest*, *Dirty Pasture*, *Clean Pasture*, *Secondary Vegetation*, plus an *Other* state representing all other possible land cover types (Müller-Hansen et al., 2017).

State-and-transition models are **often co-developed with stakeholders** and are used as **heuristic tools** to understand the dynamics of ecosystems. They are also used in **scenario development** to explore possible futures.

Müller-Hansen et al. (2017) *A matrix clustering method to explore patterns of land-cover transitions in satellite-derived maps of the Brazilian Amazon*
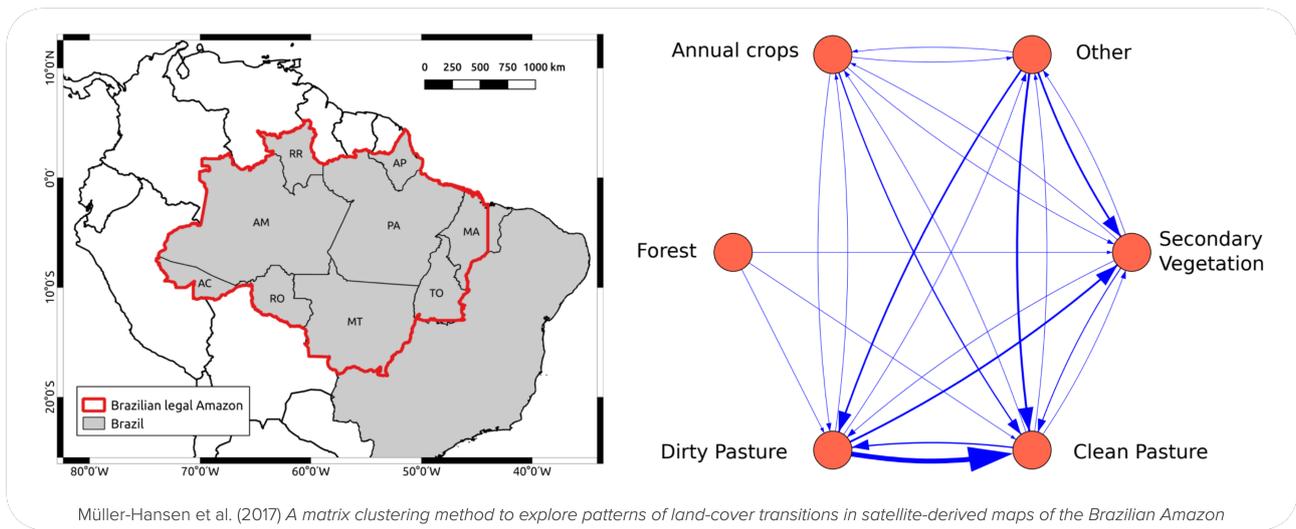
Figure 5.2: Land-cover transitions in Brazilian Amazon

Note, that terms such as *state-and-transition model* and *transition model* have been widely used in the literature without a clear, formal definition (Daniel et al., 2016).

There are **various computational variants** of state-and-transition models. The simplest and most basic model is the one of a **Markov Chain**.

5.1.1 Learning goals

After this lecture, students will be able to:

- **Name** and **explain** the *components of a Markov chain model* and how the model relates to gernal dynamic system models to embed this model in the context of integrated nature-society models.
- **Simulate** and **visualize** *Markov chain models* stochastically, with ensembles, and via its state distribution
- **Compute** the *stationary distribution* of a Markov chain model numerically, analytically and explain the conditions for its existence to understand the long-term behavior of the model.
- **Investigate** the *transient behavior* of a Markov chain model to understand the short-term behavior of the model.
- **Compute** the *typical timesacle* of a Marko chain transition to relate the model to real-world systems.

5.2 Markov chains

Markov chains model systems that transition probabilistically between a finite set of states.

In fact, Markov chains are a very general model that can be applied to all kinds of transitions. For example, a political system might transition between democratic and dictatorial, a market between volatile and stable regimes, or a person between happy, contemplative, anxious, and sad (Page, 2018, Chapter 17).

The movements between states occur according to fixed probabilities. The probability that a country transitions from authoritarian to democratic in a given year might be 5%; the likelihood that a person transitions from anxious to tired within an hour maybe 20%.

Named after Russian mathematician Andrey Markov, the essential element of the model is the **Markov property**. This property states that the probability of transitioning to any particular state **depends solely on the current state** and not on the history of states that preceded it.

More formally, we define a Markov chain by the following elements:

- A discrete set of states $\mathcal{S} = \{S_1, S_2, ..., S_Z\}$.
- A transition matrix $\mathbf{T}$ with transition probabilities $T(i, j)$, for $1 < i, j < Z$, where $T(i, j)$ is the probability of transitioning from state S_i to state S_j.
- A disscrete-time index $t = 0, 1, 2, ...$.
- An initial state distribution $\mathbf{p}_0 = (p_0(S_1), p_0(S_2), ..., p_0(S_Z))$, with $p_t(s)$ denoting the probability or fraction of state $s \in \mathcal{S}$ at time t.

Thus, the transition matrix $\mathbf{T}$ is a square matrix of size $Z \times Z$ with $\sum_{s'} T(s, s') = 1$, where s denotes the current, and s' the next state. Transition probabilites have to sum up to 1. We must go somewhere.

5.2.1 A simple example

Let us consider a simple example of a Markov chain with two states modeling a prosperous and a degraded state of Nature.

- A (discrete) set of states $\mathcal{S} = \{\mathsf{p}, \mathsf{d}\}$
- Transition probabilities $T(\mathsf{p}, \mathsf{p})$, $T(\mathsf{p}, \mathsf{d})$, $T(\mathsf{d}, \mathsf{p})$, $T(\mathsf{d}, \mathsf{d})$, modeled by
  - a collpase probability p_c, and
  - a recovery probability p_r.
- An initial state distribution $p_0(s) = (1, 0)$
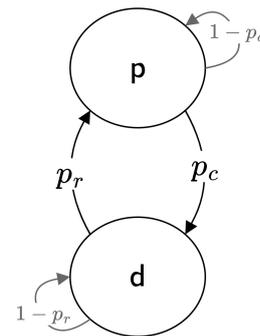
Figure 5.3: A simple Markov chain

5.2.2 Computational model

Let us convert the mathematical into computationa model in Python. We start by importing the necessary libraries and setting up the plotting environment.

```python
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt

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;
```

But how to model the transition?

Transition matrix. We know that a the rows of our transition matrix have to sum up to one, $\sum_{s'} T(s, s') = 1$. Thus, we can simplify the transition matrix by only giving a collpase probability p_c and a recovery probability p_r,

$$\left(\begin{array}{cc} T(\mathsf{p},\mathsf{p}) & T(\mathsf{p},\mathsf{d}) \\ T(\mathsf{d},\mathsf{p}) & T(\mathsf{d},\mathsf{d}) \end{array} \right) = \left(\begin{array}{cc} 1 - p_c & p_c \\ p_r & 1 - p_r \end{array} \right).$$

Lets fix the values for the transition probabilities p_c and p_r, for now,

```
pc = 0.05
pr = 0.01
```

Then we can implement the transition matrix as a two-dimensional numpy array,

```
T = np.array([[1-pc, pc],
              [pr, 1-pr]])
T
```

```
array([[0.95, 0.05],
       [0.01, 0.99]])
```

5.3 Simulations

5.3.1 A single simulation run

Considering the system to be in exaclty one of its states at each time step, we can **simulate** the Markov chain **stochastically** by choosing the next state (using `numpy.random.choice`) according to the transition probabilites as given in the transition matrix (by specifing the probabilities as `p=TransitionMatrix[current_state]`).

```
def simulate_markov_chain(TransitionMatrix, InitialState, NrTimeSteps):
    trajetory = -1*np.ones(NrTimeSteps, dtype=int)
    trajetory[0] = InitialState
    for t in range(1, NrTimeSteps):
        trajetory[t] = np.random.choice([0, 1], # sample next state
                                        p=TransitionMatrix[trajetory[t-1]])
    return trajetory
```

Visualizing the results, we can see how the system evolves over time.

```
np.random.seed(1818)
trajectory = simulate_markov_chain(T, 0, 500)
plt.plot(1-np.array(trajectory), ls='-', marker='.', color='purple');
plt.xlabel('Time steps $t$'); plt.ylabel('System state $s$');
```
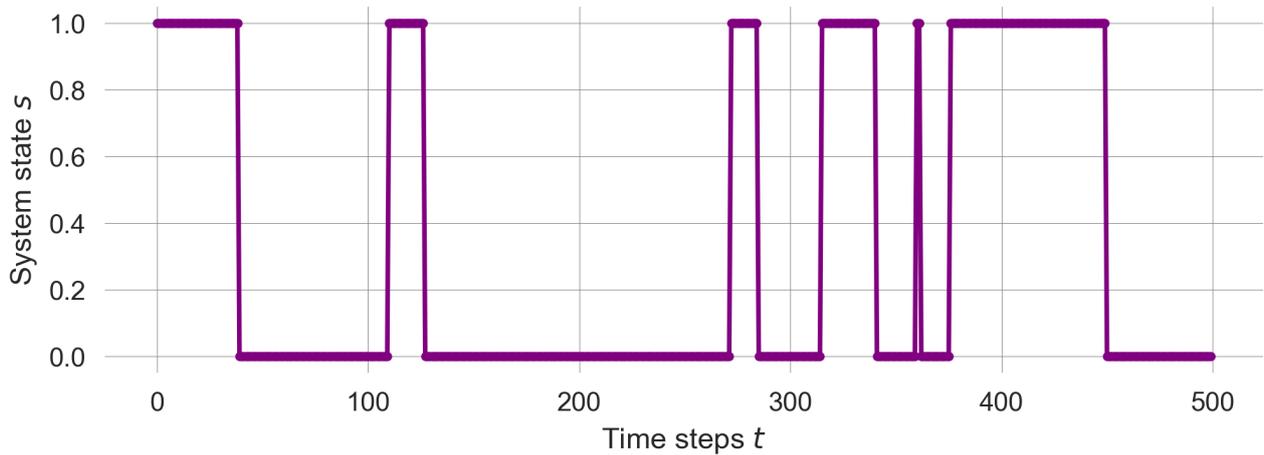
This stochastic simulation of a **single run** has a strong **resemblance** to the previous dynamic system models we introduced. We can interpret the Markovian states as the stable equilibirum points of dynamic system with nonlinear changes and noise (see Figure 5.4 from 02.03-Resilience).
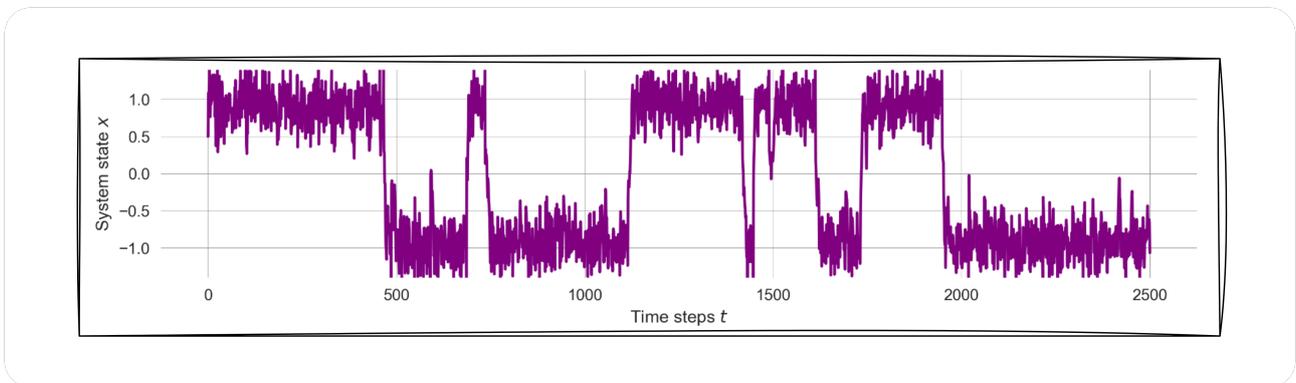
Figure 5.4: Noise induced transitions

However, due to the stochastic nature of the Markov chain, it is hard to judge the system's behavior from a single run. We need to **average over many runs** to get a clearer picture of the system's behavior.

5.3.2 Ensemble simulation

Let's repeat the previous simulation to create an **ensemble of stochastic simulation runs**. Let's assume we want an ensemble of 100 runs.

```
ensemble = []
for _ in range(100):
    state = 0
    trajectory = simulate_markov_chain(T, state, 500)
    ensemble.append(trajectory)
ensemble = np.array(ensemble)
```

It is always a good idea to investigate the object one has just created for consistency, for instance, checking the shape of the ensemble.

```
ensemble.shape
```

```
(100, 500)
```

The first dimension of the ensemble is the number of runs, the second dimension is the number of time steps.

Visualizing the ensemble by takeing the mean over the first dimension (using `ensemble.mean(axis=0)`),

```
ensemble.mean(axis=0).shape
```

```
(500,)
```

we can see how the system evolves over time on average.

```
plt.plot(1-ensemble.mean(0), ls='-', marker='.', color='blue')
plt.ylim(0, 1); plt.xlabel('Time steps $t$'); plt.ylabel('Average system state
↪  $s$');
```
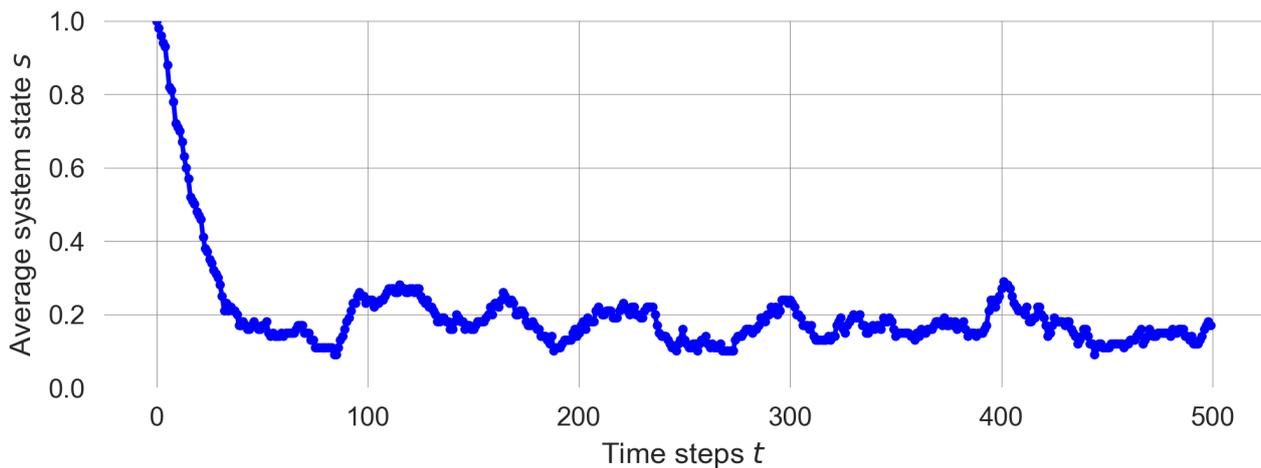

We observe two phases. First a drop. Second, some fluctuations around approx. 0.2.

Thus, after around 50-100 iterations, the system is in approximatly 20 of 100 runs in the the prosperous state.

This is the **statistical equlibirum** or the **long-run stationary distribution** of the Markov chain.

Calculating ensembles is computationlly expensive. **Can we make the computation more efficient?** To do so, we investigate how to update the state distribution of the Markov chain at each time step.

5.3.3 Markov chain update

A Markov chain **update** can be nicely **represented by a matrix operation**. The new state $\mathbf{p}_{t+1}$ equals the old state $\mathbf{p}_t$ applied to the transition matrix $\mathbf{T}$,

$$\mathbf{p}_{t+1} = \mathbf{p}_t \mathbf{T}.$$

We write the state distribution $\mathbf{p}_t$ before the transition matrix $\mathbf{T}$ on the right hand side to indicate the flow of information from left to right. We can also write this as follows,

$$p_{t+1}(s') = \sum_s p_t(s) T(s, s'),$$

where s denotes the current, and s' the next state. The next system state s' depends on the current state s and the transition from s to s' as given by the transition matrix $\mathbf{T}$.

Transposing the transition matrix, we can rewrite the Markov chain update as $p_{t+1}(s') = \sum_s T^{\mathrm{T}}(s', s) p_t(s)$ or

$$\mathbf{p}_{t+1} = \mathbf{T}^{\mathrm{T}} \mathbf{p}_t$$

where $\mathbf{T}^{\mathrm{T}}$ is the transpose of the transition matrix. Rewriting this update as such highlights the fact that Markov chains can be intepreted as a **special kind of dynamic system** with linear changes (see 02.01-Nonlinearity) but with one additional property. The sum of all variables is always one.

5.3.4 State distribution evolution

We use the matrix update to simulate how the state distribution evolves.

```
ps = [1, 0]  # initial state distribution
p_trajectory = [ps] # store the state distribution over time

for i in range(500):
    ps = ps @ T  # matrix update for the state distribution
    p_trajectory.append(ps)
p_trajectory = np.array(p_trajectory)
```

The trajectory of the state distribution has the number of time steps as the first dimension and the number of states as the second dimension.

```
p_trajectory.shape
```

```
(501, 2)
```

Since the state distribution is a probability distribution, the sum of all states at each time step should be one. We can check this by summing over the states at each time step.

```
np.allclose(p_trajectory.sum(axis=-1), 1.0)
```

```
True
```

Visualizing the state distribution evolution together with the ensemble average reveals a close resembles between the two.

```
plt.plot(1-ensemble.mean(0), ls='-', marker='.', color='blue', label='Ensemble
↳  average')
plt.plot(p_trajectory[:, 0], ls='-', marker='.', color='red', label='State
↳  distribution')
plt.xlabel('Time steps $t$'); plt.ylabel('Average system state $s$'); plt.legend();
```
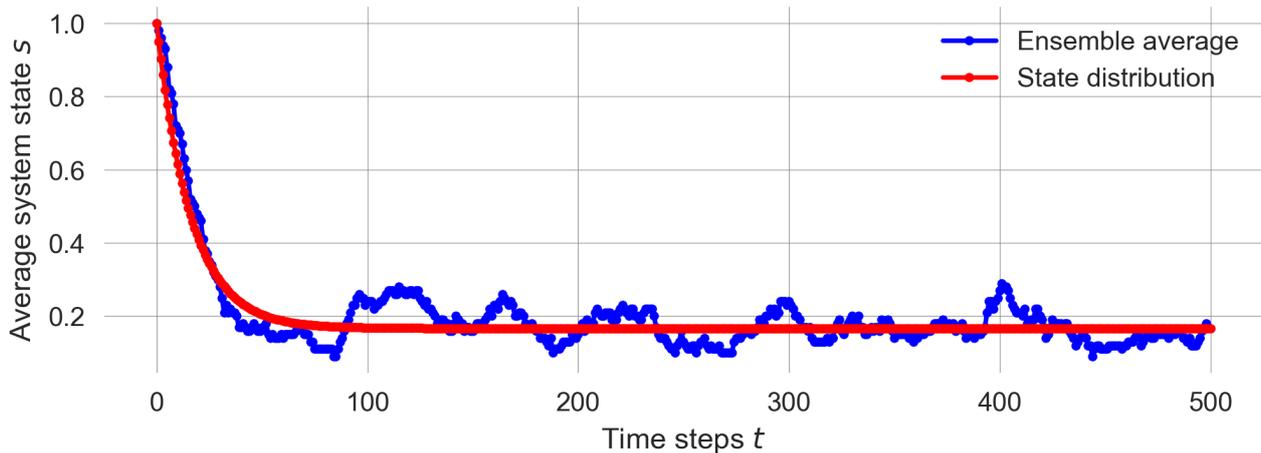

The fluctuations of the ensemble average around the long-run stationary distribution are due to the finite number of runs in the ensemble. The more runs we have, the closer the ensemble average will be to the long-run stationary distribution.

The flat line of the state distribution evolution indicates that the system has reached a **statistical equilibirum**.

For example, a statistical equilibrium in a Markov model of ideology would allow for people to transition between liberal, conservative, and independent, but the proportions of people of each ideology would remain unchanged. When applied to a single entity, a statistical equilibrium means that long-run probability of the entity being in each state does not change. A person could be in a statistical equilibrium in which he is happy 60% of the time and sad 40% of the time. The person's mental state could change from hour to hour, but his long-run distribution across those states does not (Page, 2018).

However, can we **compute the stationary distribution somehow directly**?

5.4 Stationary distribution

After showing how to compute the stationary distribution directly, we will first, compute it **numerically**, second, compute it **symbolically**, and last, we will discuss the conditions for its **existence**.

From the update equation,

$$\mathbf{p}_{t+1} = \mathbf{p}_t \mathbf{T}.$$

we know that the stationary distribution $\mathbf{p}^*$ must satisfy,

$$\mathbf{p}^* = \mathbf{p}^* \mathbf{T}$$

This looks like the defining equation of a (left) **eigenvector** with the eigenvalue 1,

$$1\mathbf{p}^* = \mathbf{p}^* \mathbf{T}$$

5.4.1 Numerical stationary distribution

Fortunately, `numpy` has built-in routine to compute the eigenvectors of a matrix. Since the standard routine `np.linalg.eig` only computes the right eigenvectors, we need to apply the routine to the transposed matrix:

```
eigvv = np.linalg.eig(T.T)
eigvv
```

```
EigResult(eigenvalues=array([0.94, 1.  ]), eigenvectors=array([[-0.70710678, -0.19611614],
       [ 0.70710678, -0.98058068]]))
```

```
eigvv[1][:,1]
```

```
array([-0.19611614, -0.98058068])
```

Normalizing the eigenvector, such that it entries comprise a probability distribution, yields

```
pstar = eigvv[1][:,1] / sum(eigvv[1][:,1])
pstar
```

```
array([0.16666667, 0.83333333])
```

Visualizing the stationary state distribution together with the state distribution evolution and the ensemble average reveals that the calculated stationary distribution fits perfectly to the distribution evolution in (statistical) equilibirum.

```
plt.plot(1-ensemble.mean(0), ls='-', marker='.', color='blue', label='Ensemble
↪   average')
plt.plot(p_trajectory[:, 0], ls='-', marker='.', color='red', label='State
↪   distribution')
plt.plot([0,500], [pstar[0], pstar[0]], '-', color='k', label='Stationary
↪   distribution')
plt.xlabel('Time steps $t$'); plt.ylabel('Average system state $s$'); plt.legend();
```
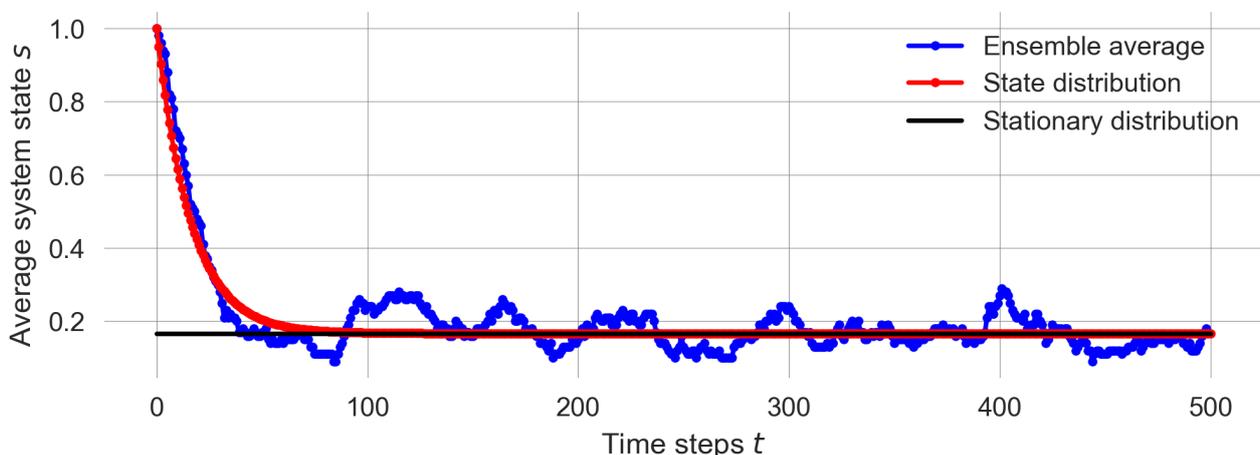
How does this result depend on the model parameters, the collapse probability p_c and the recovery probability p_r?

Can we **compute the stationary distribution analytically**, i.e., can we derive a mathematical equation which says how the stationary distribution depends on the collapse probability p_c and the recovery probability p_r?

5.4.2 Analytical stationary distribution

Fortunately, we can use Python's library for basic symbolic calculations `sympy`, to compute the eigenvectors of the transition matrix symbolically.

```
p_c, p_r = sp.symbols("p_c, p_r")
```

```
T_ = sp.Matrix([[1-p_c, p_c],
                [p_r, 1-p_r]])
T_
```

$$\begin{bmatrix} 1 - p_c & p_c \\ p_r & 1 - p_r \end{bmatrix}$$

Applying the `.eigenvects()` method

```
T_.T.eigenvects()
```

```
[(1,
  1,
  [Matrix([
   [p_r/p_c],
   [      1]])]),
 (-p_c - p_r + 1,
  1,
  [Matrix([
   [-1],
   [ 1]])])]
```

shows us which eigenvector corresponds to the eigenvalue 1. Upon normalizing the eigenvector, we obtain the analytical stationary distribution,

```
pstar_ = T_.T.eigenvects()[0][2][0] # selecting the eigenvector
pstar_ = pstar_ / (pstar_[0] + pstar_[1]) # normalizing the eigenvector
sp.simplify(pstar_)
```

$$\begin{bmatrix} \frac{p_r}{p_c + p_r} \\ \frac{p_c}{p_c + p_r} \end{bmatrix}$$

Thus, only the fraction of p_c and p_r determines the stationary distribution. The stationary distribution remains the same, if you multiply the collapse and recovery probabilities by the same factor. The fraction of the stationary distribution in the prosperous state is proportional to the recovery probability, p_r, i.e., the probability to enter the prosperous state from the degraed state.

And vice versa, the fraction of the stationary distribution in the degraded state is proportional to the collapse probability, p_c, i.e., the probability to enter the degraded state from the prosperous state.

Last, to compare this analyitical solution with the numerical calculation we insert the values for p_c and p_r into the analytical solution.

```
pstar_.subs(p_c, pc).subs(p_r, pr)
```

$$\begin{bmatrix} 0.166666666666667 \\ 0.83333333333333 \end{bmatrix}$$

```
pstar
```

```
array([0.16666667, 0.83333333])
```

We observed the the **statistical equilibrium** in the Markov chain simulation and calculated the long-run stationary distribution both numerically and symbolically. But what are the **conditions that such a unique statistical equilibrium exists?**

5.4.3 Stationary distribution | Existence

Any Markov model with a **finite set of states**, **fixed transition probabilities** between them, the **potential to move from any state to any other** in a series of transitions, and **no fixed cycles** between states converges to a unique equilibrium. These are the conditions of the Perron-Frobenius Theorem (Page, 2018).

> **Perron-Frobenius Theorem**
>
> A Markov process converges to a unique statistical equilibrium provided it satisfies four conditions:
>
> - **Finite set of states**: $\mathcal{S} = S_1, S_2, ..., S_Z$.
> - **Fixed transition rule**: The probabilities of moving between states are fixed, for example, the probability of transitioning from state S_i to state S_j equals $T(S_i, S_j)$ in every period.
> - **Ergodicity** (state accessibility): The system can get from any state to any other through a series of transitions.
> - **Noncyclic**: The system does not produce a deterministic cycle through a sequence of states.

The theorem implies that **if those four assumptions are satisfied**, the *initial state*, *history*, and *interventions that change the state* **cannot change the long-run equilibrium**.

The unique statistical equilibrium implies that long-run distributions of outcomes cannot depend on the initial state or on the path of events. In other words, initial conditions do not matter, and history does not matter in the long run. Nor can interventions that change the state matter. Any one-time change in the state of a system has at most a temporary effect.

For example,

- if nations move between dictatorships and democracies according to fixed probabilities, then interventions that impose or encourage democracies in some countries have no long-term effects.

- If fluctuations in dominant political ideologies satisfy the assumptions, then history cannot influence the long-run distribution over ideologies.
- And if a person's mental state can be represented as a Markov model, then words of encouragement or supportive gestures have no long-run impact.

The **takeaway** from the theorem should not be that history cannot matter but that **if history does matter, one of the model's assumptions must be violated**.

Two assumptions—the finite number of states and no simple cycle—almost always hold.

Ergodicity can be violated. However, in practice, it is often possible to ensure ergodicity by adding a tiny transition probability between states that are not directly connected. This tiny transition probability can justified by our lack of knowledge about the system.

Thus, the **assumption of fixed transition probabilities** between states is the least likely to be valid. When history is important, something must alter the transition probabilities.

For example, take the issue of assisting families in escaping poverty. The forces that create social inequality have proven immune to policy interventions. In Markov models interventions that change families' states—such as special programs for underperforming students or a one-day food drive—can provide temporary boosts. They cannot change the long-run equilibrium. In contrast, interventions that provide resources and training that improve people's ability to keep jobs, and therefore change their probabilities of moving from employed to unemployed, could change long-run outcomes (Page, 2018).

We investigated the long-run behavior of the Markov chain model. But **what about the short-term behavior**?

5.5 Transient behavior

We found that our example system's statistical equilibirum, its stationary distribution, depends only on the fraction of the collapse and recovery probabilities, p_c and p_r.

To investigate the short-term behavior before the evolution of the state distribuion reaches its equilibirum, we create different pairs of collapse and recovery probabilities while keeping their fraction constant.

```
prs = np.array([0.1, 0.03, 0.01, 0.003, 0.001])
pcs = 5 * prs
pcs
```

```
array([0.5  , 0.15 , 0.05 , 0.015, 0.005])
```

We can insert these in the `sympy` matrix as follows,

```
np.array(T_.subs(p_c, pc).subs(p_r, pr), dtype=float)
```

```
array([[0.95, 0.05],
       [0.01, 0.99]])
```

5.5.1 Simulating different transition probabilities

It is convenient to define a function to obtain the time evolution for the state distributions.

```python
def compute_distribution_trajectory(T):  # Transition matrix
    ps = [1, 0]
    p_trajectory = []

    for i in range(500):
        ps = ps @ T
        p_trajectory.append(ps)
    return np.array(p_trajectory)
```

With that function, we simply compute the time evolution of the state distribution for the different values of p_r and p_c.

```python
trajs = []
for pr, pc in zip(prs, pcs):
    Tmat = np.array(T_.subs(p_c, pc).subs(p_r, pr), dtype=float)
    trajs.append(compute_distribution_trajectory(Tmat))
np.array(trajs).shape
```

```
(5, 500, 2)
```

5.5.2 Visualizing distribution trajectories

```python
for i, traj in enumerate(trajs):
    plt.plot(traj[:, 0], ls='-', marker='.', label=prs[i])
plt.plot([0,500], [pstar[0], pstar[0]], '-', color='black')
plt.ylim(0, 1); plt.xlabel('Time steps'); plt.legend(title='Recovery probability');
```
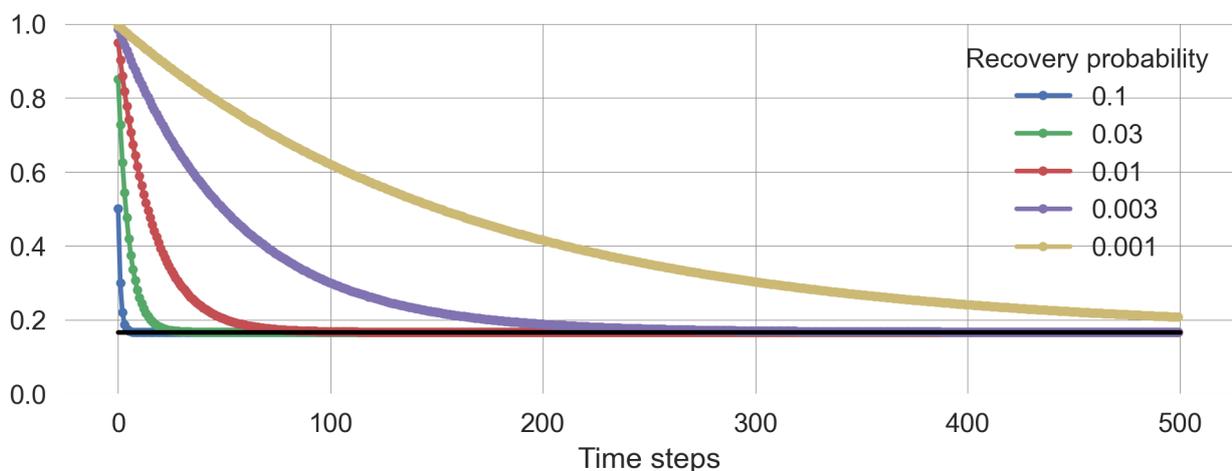

The smaller the transition probabilities, the longer it takes to reach the stationary distribution.

116

5.6 Timescales

Sometimes, it can be interesting or adquate to think about the typical **timescales** of a system. For example, consider the typical timescale it takes a system to tip into another state. **How can we identify the notion of timescale in a Markov chain?**

Let us identify a notion of timescale as the average **number of time steps** spend in a particular state before a transition occurs. How can we calculate it?

We will first compute these timescales numerically, showcasing somewhat more advanced maniumplation using Python. Then we turn to an analytical formula and compare the results.

5.6.1 Numerical computation

To investigate this question nummerically, we re-create a (long) trajectory of states.

```python
state = 0
pc = 0.2
pr = 0.04
T = np.array(T_.subs(p_c, pc).subs(p_r, pr), dtype=float)
trajectory = []
for i in range(500000):
    state = np.random.choice([0,1], p=T[state])
    trajectory.append(state)
trajectory = np.array(trajectory)
```

Looking at the first 100 states

```python
shorttraj = trajectory[:99]
shorttraj
```

```
array([0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

How do we obtain the lengths of the 0 and 1 sequences?

We can subtract the trajectory by itsel with an offset of one time step. For the first 100 time steps, this looks like

```python
shorttraj[0:-1] - shorttraj[1:]
```

```
array([-1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  1,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1, -1,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0])
```

With `np.nonzero` with obtain the 'cutpoints' where the states change,

```
cutpoint = np.nonzero(trajectory[0:-1] - trajectory[1:])
cutpoint
```

```
(array([     0,      19,      26, ..., 499904, 499960, 499985]),)
```

With `np.diff`, we obtain the differences between the cutpoints. These are the lengths of the sequences.

```
lengths = np.diff(cutpoint)
lengths
```

```
array([[19,  7, 22, ..., 10, 56, 25]])
```

The lengths of the prosperous states are in the odd elements of the `lengths` iterable. Taking the average yields

```
length_prosperous = np.mean(lengths[0][1::2])
length_prosperous
```

```
4.982404866992724
```

The lengths of the degraded states are in the even elements of the `lengths` iterable. Taking the average yields

```
length_degraded = np.mean(lengths[0][0::2])
length_degraded
```

```
24.838959799594416
```

5.6.2 Analytical computation

The average number of time steps T it takes until a transition occurs, given a transition probability p, is

$$T = \sum_{n=0}^{\infty} n(1-p)^{n-1}p = \frac{1}{p}.$$

For a sequence of length n it took $n-1$ time steps to remain in state s before the transition. The probability of remaining in state s for $n-1$ time steps is $(1-p)^{n-1}$. At the n'th time step, the transition occurs with probability p.

This is the expected value of a geometric random variable with parameter p. The number of steps T spent in state s before a transition occurs can be thought of as the number of trials until the first success in a sequence of Bernoulli trials, where each trial has a success probability of p of transitioning out of s. In probability theory, the number of trials required to get the first success in such a situation is described by a geometric random variable. It expected value is $\mathbb{E}[T] = 1/p$.

For the prosperous state, we have

```
1/pc
```

```
5.0
```

compared to our numerical estimate

```
length_prosperous
```

```
4.982404866992724
```

For the degraded state, we have

```
1/pr
```

```
25.0
```

compared to our numerical estimate

```
length_degraded
```

```
24.838959799594416
```

Thus, the average number of time steps before a transition occurs equals the inverse transition probability $1/p$. This gives us an indication of the typical timescale of the system.

5.6.3 Example | Regime shifts timescales

Typical collapse and recovery timescales of regime shifts from the Regime Shifts Database have been mapped to the transition probabilities of a Markov chain in this way (Barfuss, Donges, et al., 2024) (Figure 5.5).

Barfuss et al. (2024) *Ecologically-mediated collective action in commons with tipping elements*
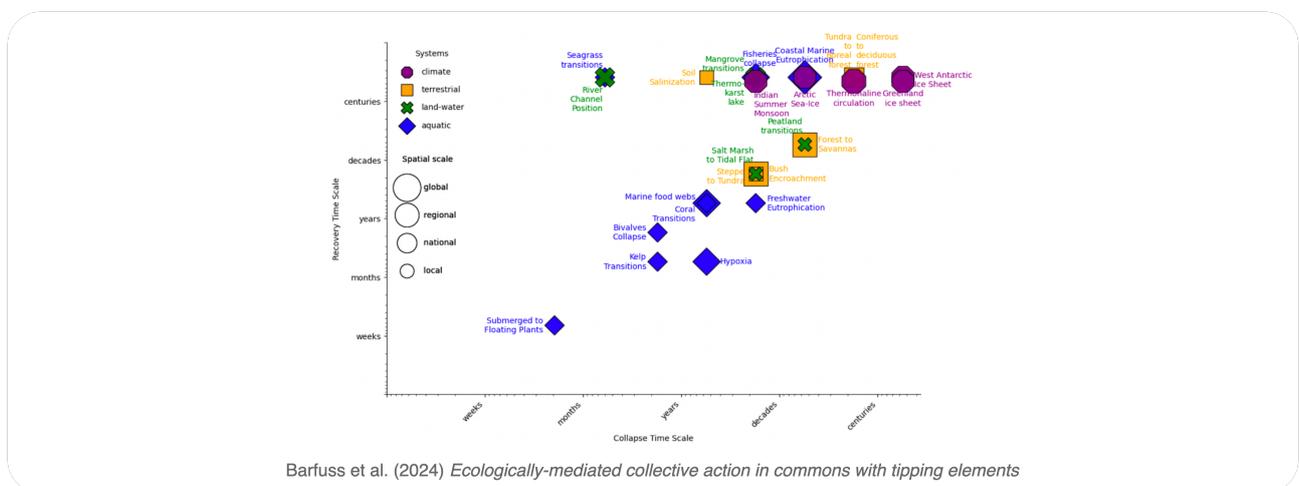
Figure 5.5: Timescales

119

5.7 Learning goals revisited

- We introduced the *components of a Markov chain model* and covered how the model relates to gernal dynamic system models to place this model in the context of integrated nature-society models.
- We simulated and visualized *Markov chain models* in multiple ways: stochastically, with ensembles, and via its state distribution to understand how the model behaves.
- We computed the *stationary distribution* of a Markov chain model numerically, analytically and explain the conditions for its existence to understand the long-term behavior of the model.
- We investigated the *transient behavior* of a Markov chain model to understand the short-term behavior of the model.
- We computed the *typical timesacle* of a Marko chain transition to relate the model to real-world systems.

Part II

Target Equilibria

In this part, we cover target-equilibria models or equilibrium-based models. They operationalize **target knowledge**, which is knowledge about the desired future and the values that indicate which direction to take. It relies on deliberation by different societal actors and is based on values and norms. In sustainability transitions, ways of producing target knowledge include participatory vision, scenario development with a wide range of stakeholders, and the public discourse at large. Target knowledge is strongly associated with **values** and asks **what ought to be?**.
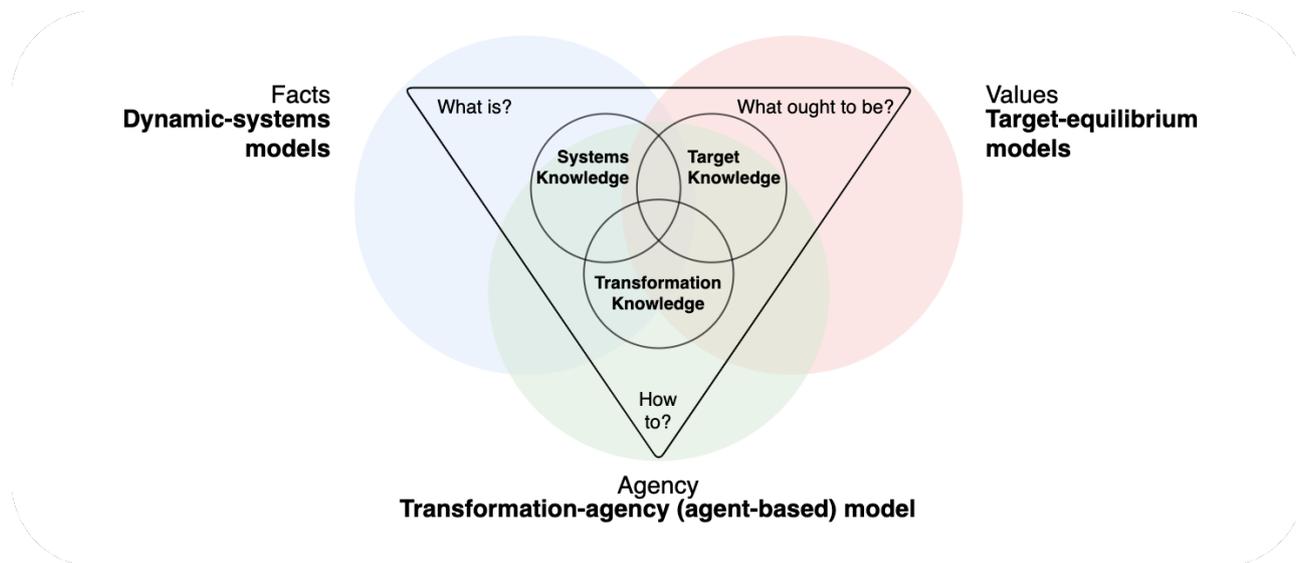
Figure 5.6: Three types of models based on three types of knowledge for transdisciplinary reserach

Target-equilibrium (or *equilibrium-based models applied to sustainability transitions*) are primarily used in economics. The overarching idea of the model type is to find a target equilibrium, a state of the system that is considered desirable. In contrast to the **dynamic systems** models, **target equilibrium** models introduce at least one decision-maker into the model. Given our assumptions about how the world works and that we can precisely specify what we want, we can use optimization techniques to find the **best possible course of action**. Having found the best course of action, we are in a 'target equilibrium.'

The *decision-maker* is sometimes called an *agent* or *actor*. It can be a single individual or a group of individuals, such as a household, a company, a government, or a non-governmental organization. It can be a human, an animal, or a machine. It may even be conceivable that a single human consists of multiple agents. **Thus, when introducing the concept of an agent, we obtain an abstract but flexible modeling tool to represent the agency and decision-making** of a wide range of possible entities.

Specifically, we will cover

- Sequential decisions of a single agent in a dynamic environment in Chapter 03.01
- Strategic interactions of multiple agents in a static environment in Chapter 03.02, and
- Dynamic interactions of multiple agents in a dynamic environment in Chapter 03.03.

6 Sequential Decisions

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

6.1 Motivation | Sequential decision-making under uncertainty

We will introduce the model framework of **Markov Decision Processes (MDPs)** to model sequential decision-making under uncertainty. MDPs are a powerful tool to model decision-making processes in various applications, such as robotics, finance, and environmental management.

MDPs highlight the trade-off between current and future wellbeing in the presence of uncertainty.

- Markov Decision Processes (MDPs) are models for sequential decision-making when outcomes are uncertain.
- They extend Markov Chains by a single **agent**, executing an *action* at each time step, trying to optimize its long-term wellbeing.

6.1.1 Applications in human-environment interactions

MDPs are widely used in environmental management and conservation biology to model the trade-off between current and future wellbeing in the presence of uncertainty (Marescot et al., 2013; Williams, 2009). Application areas cover the whole spectrum of **natural resource ecology, management, and conservation**, including

- **forestry and forest management**
- **fisheries and aquatic management**
- **wildlife and range management**
- **weeds, pest, and disease control**

In ecology, the term *stochastic dynamic programming* (SDP) is often used to refer to both the mathematical model (MDP) and its solution techniques (SDP per see).

6.1.2 Advantages of Markov decision processes

Using MDPs to model human-environment interactions has several advantages:

- **inherently stochastic** - to account for uncertainty
- **nonlinear** - to account for structural changes
- **agency** - to account for *human behavior*
- **future-looking** - to account for the trade-off between short-term and long-term
- **feedback** - between one agent and the environment

In addition to these structural advantages, MDPs can also be solved in a computationally efficient way using a variety of algorithms, such as dynamic programming and reinforcement learning. This makes them also a **scalable** modeling framework. However, as **our focus** lies on **transparent analysis and interpretation**, we will focus on **minimalistic models** and won't cover the computational aspects. But in principle, MDPs can be used to model high-dimensional systems with many states and actions.

6.1.3 Learning goals

After this lecture, students will be able to:

- **Describe** the elements of a Markov Decision Process (MDP) and how they relate to applications in human-environment interactions
- **Simulate** and **visualize** the time-evolution of an MDP
- **Explain** what value functions are, why they are useful, and how to relate to the agent's goal and Bellman equation.
- **Compute** value functions and **visualize** the best policy in simple MDPs

6.2 Markov Decision Processes (MDPs)

```python
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed

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;
```

Graphically, a Markov decision process can be represented by the agent-environment interface in Figure 6.1.
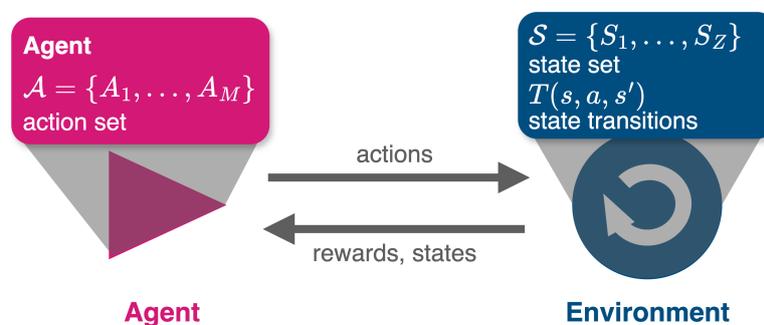
Figure 6.1: Agent-Environment Interface

Formally, we define a Markov Decision process by the following elements:

- A discrete-time variable $t = 0, 1, 2, ...$
- A discrete set of **contexts** or **states** $\mathcal{S} = \{S_1, ..., S_Z\}$.
- A discrete set of **options** or **actions** $\mathcal{A} = \{A_1, ..., A_M\}$.
- A transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$.

    - $T(s, a, s')$ is the transition probability from current state s to the next state s' under action a.

- A **welfare** or **reward** function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$.

    - $R(s, a, s')$ is the current/immediate/short-term reward the agent receives when executing action a in state s and transitioning to state s'.

- The agent's **goal** or **gain** function G, including a discount factor $\gamma \in [0, 1)$, denoting how much the agent cares for future rewards
- The agent's **policy** or **strategy** $x : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$.

6.2.1 Example model overview

We will illustrate the concept of MDPs using a simple example (Barfuss et al., 2018), modeling the trade-off between short-term gains and environmental collapse with long-term consequences for the decision-maker's wellbeing.
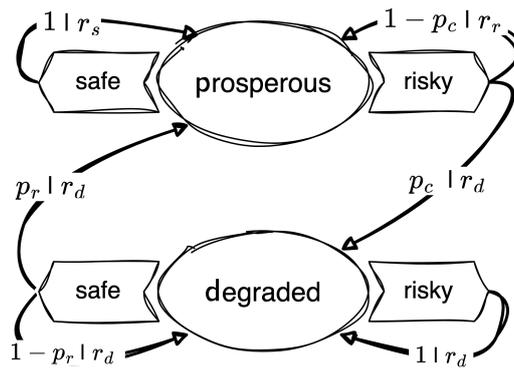
Figure 6.2: Risk-reward dilemma

6.2.2 States and actions

The environment consists of two states, $\mathcal{S} = \{\mathsf{p}, \mathsf{d}\}$, representing a **prosperous** and a **degraded** state of the environment.

```
state_set = ['prosperous', 'degraded']; p=0; d=1
```

We also defined two Python variable `p=0` and `d=1` to serves as readable and memorable indices to represent the environmental contexts.

The agent can choose between two actions, $\mathcal{A} = \{\mathsf{f}, \mathsf{r}\}$, representing a **safe** and a **risky** decision.

```
action_set = ['safe', 'risky']; f=0; r=1
```

Likewise, we define two Python variables, `f=0` and `r=1`, to serve as readable and memorable indices to represent the agent's actions. We represent the *safe* action with the f instead of the s to avoid confusion with the state s.

6.2.3 Transitions | Environmental dynamics

The **environmental dynamics**, i.e., the transitions between environmental state contexts are modeled by two parameters, a collapse probability, p_c, and a recovery probability, p_r.

Let's assume the following default values,

```
pc = 0.05
pr = 0.025
```

We implement the transitions as a three-dimensional array or **tensors**, with dimensions $Z \times M \times Z$, where Z is the number of states and M is the number of actions.

```
T = np.zeros((2,2,2))
```

The cautious action guarantees to remain in the prosperous state, $T(\mathsf{p}, \mathsf{f}, \mathsf{p}) = 1$. Thus, the agent can avoid the risk of environmental collapse by choosing the cautious action, $T(\mathsf{p}, \mathsf{f}, \mathsf{d}) = 0$.

```
T[p,f,d] = 0
T[p,f,p] = 1
```

The risky action risks the collapse to the degraded state, $T(\mathsf{p}, \mathsf{r}, \mathsf{d}) = p_c$, with a collapse probability p_c. Thus, with probability $1 - p_c$, the environment remains prosperous under the risky action, $T(\mathsf{p}, \mathsf{r}, \mathsf{p}) = 1 - p_c$.

```
T[p,r,d] = pc
T[p,r,p] = 1-pc
```

At the degraded state, recovery is only possible through the cautious action, $T(\mathsf{d}, \mathsf{f}, \mathsf{p}) = p_r$, with recovery probability p_r. Thus, with probability $1 - p_r$, the environment remains degraded under the cautious action, $T(\mathsf{d}, \mathsf{f}, \mathsf{d}) = 1 - p_r$.

```
T[d,f,p] = pr
T[d,f,d] = 1-pr
```

Finally, the risky action at the degraded state guarantees a lock-in in the degraded state, $T(\mathsf{d}, \mathsf{r}, \mathsf{d}) = 1$. Thus, the environment cannot recover from the degraded state under the risky action, $T(\mathsf{d}, \mathsf{r}, \mathsf{p}) = 0$.

```
T[d,r,p] = 0
T[d,r,d] = 1
```

Last, we make sure that our transition tensor is normalized, i.e., the sum of all transition probabilities from a state-action pair to all possible next states equals one, $\sum_{s'} T(s, a, s') = 1$.

```
assert np.allclose(T.sum(-1), 1.0)
```

All together, the transition tensor looks as follows:

```
T
```

```
array([[[1.   , 0.   ],
        [0.95 , 0.05 ]],

       [[0.025, 0.975],
        [0.   , 1.   ]]])
```

6.2.4 Rewards | Short-term welfare

The rewards or welfare the agent receives represent the ecosystem services the environment provides. It is modeled by three parameters: a safe reward r_s, a risky reward $r_r > r_s$, and a degraded reward $r_d < r_s$. We assume the following default values,

```
rs = 0.8
rr = 1.0
rd = 0.0
```

As the transition, we implement the rewards as a three-dimensional array or **tensor**, with dimensions $Z \times M \times Z$, where Z is the number of states and M is the number of actions.

```
R = np.zeros((2,2,2))
```

The cautious action at the prosperous state guarantees the safe reward, $R(\mathsf{p}, \mathsf{f}, \mathsf{p}) = r_s$,

```
R[p,f,p] = rs
```

The risky action at the prosperous leads to the risky reward if the environment does not collapse, $R(\mathsf{p}, \mathsf{r}, \mathsf{p}) = r_r$,

```
R[p,r,p] = rr
```

Yet, whenever the environment enters, remains, or leaves the degraded state, it provides only the degraded reward $R(\mathsf{d}, :, :) = R(:, :, \mathsf{d}) = r_d$, where : denotes all possible states and actions.

```
R[d,:,:] = R[:,:,d] = rd
```

Together, the reward tensor looks as follows:

```
R
```

```
array([[[0.8, 0. ],
        [1. , 0. ]],

       [[0. , 0. ],
        [0. , 0. ]]])
```

6.2.5 Policy

For now, we have not contemplated how the agent should behave. Therefore, to understand how transitions, rewards, and policies are related, let us simulate the MDP using a random policy.

We will implement a policy as a two-dimensional array or **tensor**, with dimensions $Z \times M$, where Z is the number of states and M is the number of actions.

```
X = np.random.rand(2,2)  # random values between 0 and 1
```

A policy has to be a probability distribution over actions for each state, $\sum_a X(s,a) = 1$. To ensure this, we normalize the policy tensor,

```
X = X / X.sum(axis=-1, keepdims=True)
```

and test, if the policy is a valid probability distribution by asserting that the sum of all probabilities over actions is equal to one,

```
assert np.allclose(X.sum(-1), 1.0)
```

Together, the policy tensor looks as follows:

```
X
```

```
array([[0.32878922, 0.67121078],
       [0.9813571 , 0.0186429 ]])
```

We convert this logic into a Python function, that returns a random policy for a given number of states and actions,

```
def random_policy(Z=2, M=2):
    X = np.random.rand(Z,M)  # random values
    X = X/X.sum(axis=-1, keepdims=True) # normalize values, such that
    assert np.allclose(X.sum(-1), 1.0) # X is a proper probability distribution
    return X
```

For example, a random policy for our example MDP with two states and two actions looks as follows:

```
np.random.seed(42)
random_policy(M=2, Z=2)
```

```
array([[0.28261752, 0.71738248],
       [0.55010153, 0.44989847]])
```

This completes all definitions required for an MDP. We can now simulate the MDP by iterating over time steps and applying the policy to the current state.

6.3 Simulation

6.3.1 Stochastic simulation

As in the case of Markov chains, we can simulate the MDP by drawing random numbers. We draw the actions according to the policy and the next state according to the transition probabilities. The reward is then a result of the current state, the current action, and the next state. We implement this as a Python function that takes the transition tensor, the reward tensor, the policy tensor, the initial state, and the number of time steps as input arguments.

```python
def simulate_markov_decision_process(TransitionTensor, RewardTensor, Policy,
↪  InitialState, NrTimeSteps):
    state_trajectory = []
    reward_trajectory = []
    state = InitialState

    for t in range(0, NrTimeSteps):
        # Choose random action according to policy:
        action = np.random.choice([f, r], p=Policy[state])
        # Transition to new state:
        state_ = np.random.choice([p, d], p=TransitionTensor[state][action])
        # Record reward:
        reward = RewardTensor[state, action, state_]
        # Update state:
        state = state_
        # Store in trajectories
        state_trajectory.append(state);
        reward_trajectory.append(reward)

    return np.array(state_trajectory), np.array(reward_trajectory)
```

We execute the simulation and visualize the time-evolution of the MDP's environmental state and agent's rewards.

```python
np.random.seed(1818)
state_trajectory, reward_trajectory = simulate_markov_decision_process(T, R, X, 0,
↪  500)

fig, axes = plt.subplots(2,1)
axes[0].plot(1-np.array(state_trajectory), ls='-', marker='.', color='Darkblue')
axes[1].plot(reward_trajectory, color='Red'); axes[0].set_ylabel('Environment');
↪  axes[1].set_ylabel('Rewards'); axes[1].set_xlabel('Time steps');
↪  plt.tight_layout();
```
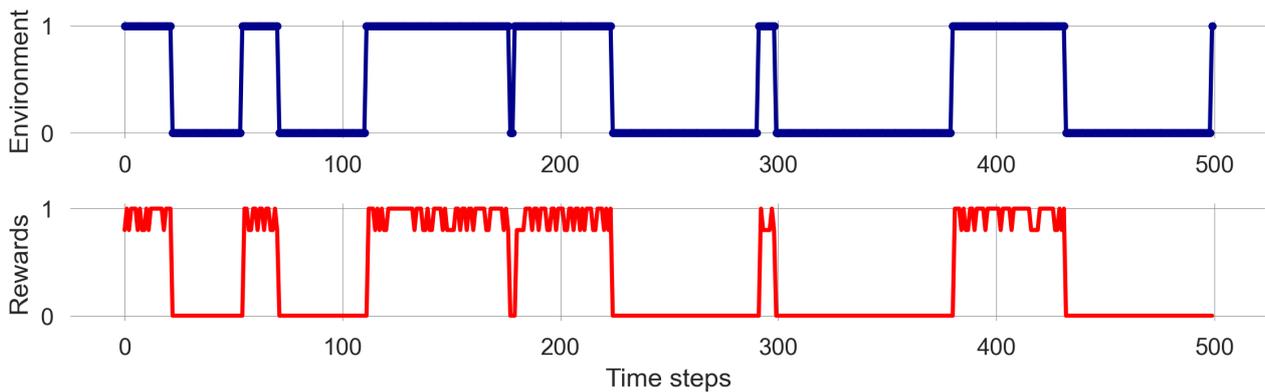
We observe the same stochastic nature of the simulation as with Markov chains. Furthermore, the agent's rewards fluctuate over time, depending on the environmental state and the agent's actions. The agent's rewards are higher in the prosperous state and lower in the degraded state.

Can we make sense of the stochasticity by computing averages over many simulations?

6.3.2 Ensemble simulation

Let's repeat the previous simulation to create an **ensemble of stochastic simulation runs**. Let's assume we want an ensemble of 250 runs.

```python
state_ensemble = []
reward_ensemble = []
for _ in range(250):
    state = 0
    state_trajectory, reward_trajectory =\
        simulate_markov_decision_process(T, R, X, 0, 500)
    state_ensemble.append(state_trajectory)
    reward_ensemble.append(reward_trajectory)
state_ensemble = np.array(state_ensemble)
reward_ensemble = np.array(reward_ensemble)
```

It is always a good idea to investigate the object one has just created for consistency, for instance, checking the shape of the ensemble.

```python
print(state_ensemble.shape, reward_ensemble.shape)
```

```
(250, 500) (250, 500)
```

For each ensembel, the first dimension of the ensemble is the number of runs, the second dimension is the number of time steps.

Visualizing the ensemble by takeing the mean over the first dimension (using `ensemble.mean(axis=0)`),

```python
fig, axes = plt.subplots(2,1)
axes[0].plot(1-state_ensemble.mean(0), ls='-', marker='.', color='Darkblue')
axes[1].plot(reward_ensemble.mean(0), color='Red');
axes[0].set_ylabel('Environment'); axes[1].set_ylabel('Rewards');
axes[0].set_ylim(-0.1,1.1); axes[1].set_ylim(-0.1,1.1); axes[1].set_xlabel('Time
↪  steps');
```

130

```
plt.tight_layout();
```
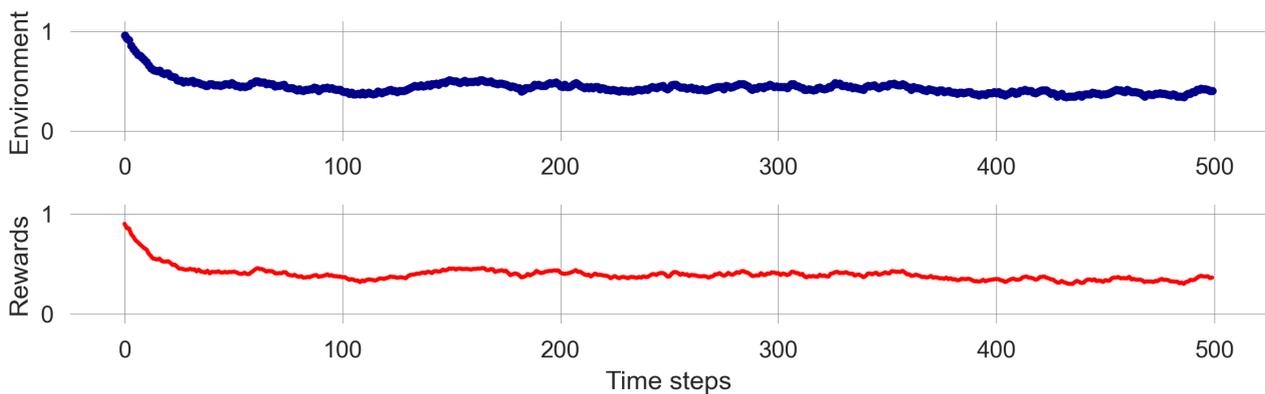

Figure 6.3

Figure 6.3 shows the ensemble average of the environmental state and the agent's rewards over time. The ensemble average is smoother than the individual runs, indicating that the stochasticity averages out over many runs. This observation suggests that we can work with the MDP in the same way as a Markov chain, simulating the time evolution of the state **distribution** directly.

6.3.3 Distribution trajectory

We realize that the MDP's transition tensor can be reduced to a Markov Chain's transition matrix when we fix the agent's policy:

$$T_{\mathbf{x}}(s, s') := \sum_{a \in \mathcal{A}} x(s, a) T(s, a, s')$$

In Python, we use the `einsum` function for that, since it gives us full control over which indices we want to execute the summation:

```
s, a, s_ = 0, 1, 2
Tss = np.einsum(X, [s,a],       # first object with indices
                T, [s,a,s_],    # second object with indices
                [s,s_])         # indices of the output
Tss
```

```
array([[0.96643946, 0.03356054],
       [0.02453393, 0.97546607]])
```

With the effective Markov chain transition matrix, we use the matrix update derived in 02.04-StateTransitions to simulate how the state distribution evolves.

```
ps = [1, 0]
p_trajectory = []
for i in range(500):
    ps = ps @ Tss
    p_trajectory.append(ps)
p_trajectory = np.array(p_trajectory)
```

The trajectory of the state distribution has the number of time steps as the first dimension and the number of states as the second dimension.

```
p_trajectory.shape
```

```
(500, 2)
```

Visualizing the state distribution evolution together with the ensemble average reveals a close resembles between the two.

```
plt.plot(1-state_ensemble.mean(0), ls='-', marker='.', color='Darkblue',
↪  label='Ensemble average')
plt.plot(p_trajectory[:, 0], ls='-', marker='.', color='blue', label='State
↪  distribution')
plt.xlabel('Time steps $t$'); plt.ylabel('Average system state $s$'); plt.legend();
```
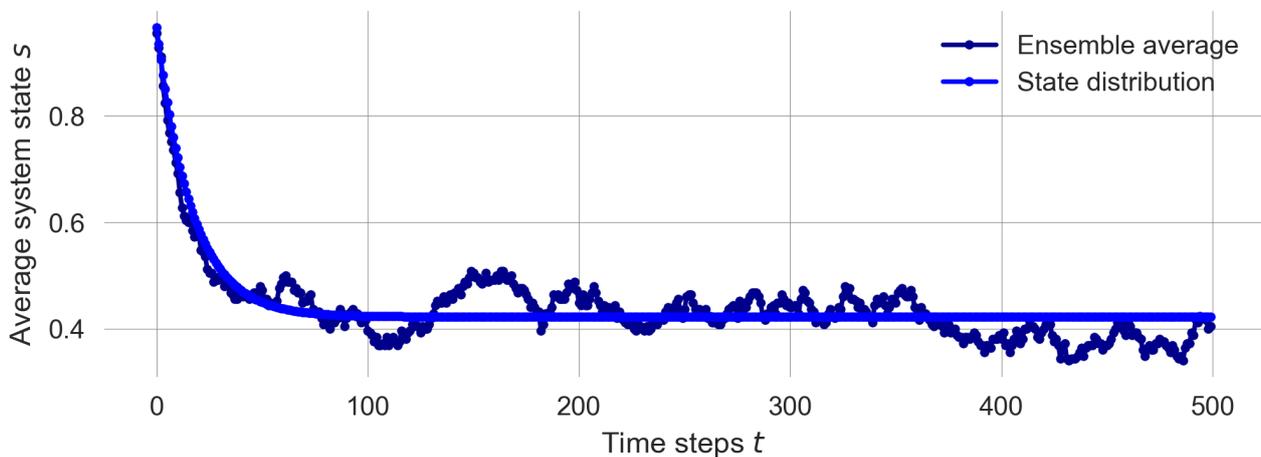

To compute the average reward trajectory over time, we use the same logic as for the state distribution trajectory, $p_t(s)$. We compute the reward distribution by summing over the state dimension, weighted by the state distribution,

$$\langle R_t \rangle_{\mathbf{x}} = \mathbb{E}_{\mathbf{x}}[r_t] = \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} p_t(s) x(s, a) T(s, a, s') R(s, a, s'),$$

where $\mathbb{E}_{\mathbf{x}}[\cdot]$ denotes the expected value of a random variable $\cdot$ given the agent follows policy $\mathbf{x}$.

You see how, in this equation on the right hand side, the information flows from the left to the right. The state distribution $p_t(s)$ is multiplied with the policy $x(s, a)$ to get the probability of taking action a. This probability is then multiplied with the transition probability $T(s, a, s')$ to get the probability of transitioning to state s'. Finally, this probability is multiplied with the reward $R(s, a, s')$ to get the expected reward.

We use the **einsum** function to convert this logic into Python,

```
s, a, s_, t = 0, 1, 2, 3
r = np.einsum(p_trajectory, [t, s],
              X, [s,a],
              R, [s,a,s_],
              T, [s,a,s_],
              [t])  # output only in time dimension
```

We check that the average-reward trajectory is only a one-dimensional array, with the number of timesteps as the first dimension.

```
r.shape
```

```
(500,)
```

Visualizing the average-reward distribution evolution together with the ensemble average reveals a close resemblance between the two.

```
plt.plot(reward_ensemble.mean(0), color='Red', label='Ensemble average');
plt.plot(r, ls='-', marker='.', color='pink', label='Average rewards');
plt.xlabel('Timesteps $t$'); plt.ylabel('Rewards $R$'); plt.legend();
```
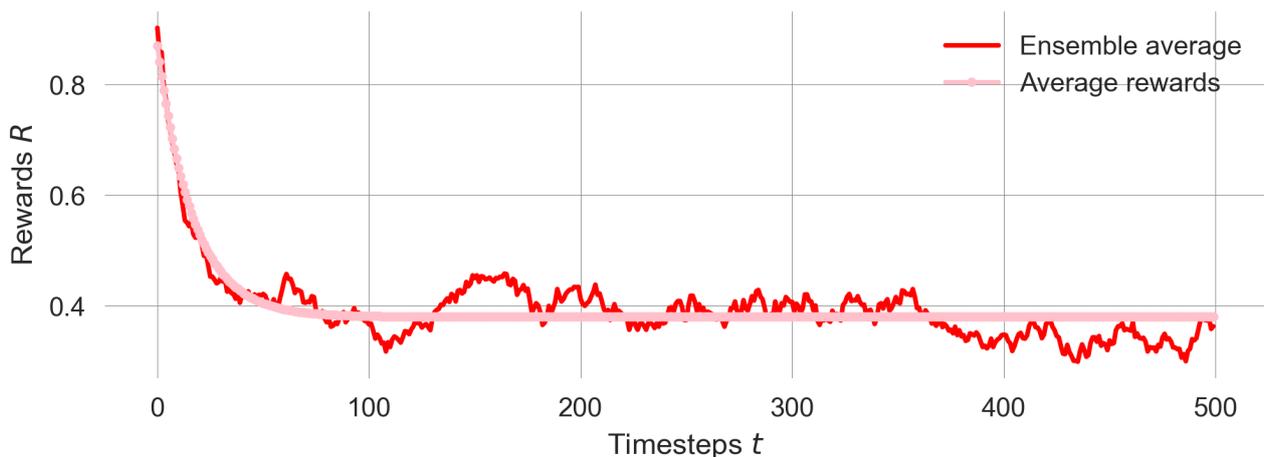

Thus, we can also calculate the stationary distribution of an MDP given a policy **x** in the same way as for a Markov chain.

6.4 Goals and values

In the Markov Decision Process framework, the agent's **purpose** or goal is formalized within the *reward signal*, flowing from the environment to the agent (Sutton & Barto, 2018). At each time step, the agent the reward is represented by a single number $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives over time. This may entail choosing actions that yield less immediate rewards to get more rewards in the future.

Representing the agent's goal by a series of single numbers might seem limiting. However, in practice, it has proven itself flexible and widely applicable. It also aligns well with the unidimensional concepts of **utility** in economics (Schultz et al., 2017) and **fitness** in biological or cultural evolution.

6.4.1 Goal functions

How do we translate our informal definition of the agent's goal as maximizing the total amount of reward into a formal mathematical equation?

Finite-horizon goal. The simplest case for a goal function G_t is to sum up all rewards the agent receives from timestep t onwards until the final time step T,

$$G_t := R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T = \sum_{\tau=t+1}^{T} R_\tau.$$

This definition makes sense only if we have a clearly defined final state, such as the end of a board game, the completion of an individual project, or the end of an individual's life. However, in human-environment interaction in the context of sustainability transitions, we are interested in the long-term future without a clear final state. In these cases, we cannot use the goal definition from above as with $T = \infty$, the sum G_t itself could easily be infinite for multiple reward sequences, which would leave the agent without guidance on which reward sequence yields a higher G_t and, hence, what to do.

For example, on average, our ensemble of stochastic simulations yields a total finite-horizon gain of

```
reward_ensemble.sum(axis=1).mean()
```

```
197.87679999999997
```

Discounted goal. We solve this problem of diverging gains G_t with the concept of **temporal discounting**. We revise our informal definition of the agent's goal: The agent tries to select actions to maximize the sum of discounted future rewards (Sutton & Barto, 2018). The goal function then becomes,

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{\tau=t}^{\infty} \gamma^\tau R_{t+\tau+1},$$

where $\gamma \in [0, 1)$ is the **discount factor**. The discount factor determines how much the agent cares about future rewards. A discount factor of $\gamma = 0$ means that the agent only cares about the immediate reward, while as the discount factor approaches $\gamma \to 1$, the agent takes future rewards into account more strongly and becomes more farsighted.

For example, the discounted gain with a discount factor of $\gamma = 0.9$ of the last ensemble run is

```
np.sum([0.9**t * reward_ensemble[-1, t] for t in range(500)])
```

```
9.254059133343878
```

Temporal discounting is a widely used concept in (environmental) economics, psychology, and neuroscience to model human decision-making. It implies that welfare experienced in the future is worth less to the agent than the same amount of welfare experienced now. This concept is used both as a **normative** and **descriptive** model of decision-making.

One reason for temporal discounting is the **uncertainty** about the future. The future is uncertain, and the agent might not be around to experience future rewards. In fact, γ can be interpreted as the probability that the agent will be around to experience future rewards.

The primary value of temporal discounting and the discount factor for us in our quest to develop integrated system models of human-environment interactions is its ability to **model the trade-off between present and future welfare**. This trade-off is at the heart of many sustainability transitions, such as the trade-off between short-term economic gains and long-term environmental degradation.

For example, let's assume the agent receives a constant reward stream of $R_t = 1$ for all timesteps t. We compare the so-called (net) present value at timestep t for different discount factors γ. We also compute the sum of the discounted rewards for the infinite future, G_t.

```python
for discountfactor in [0.1, 0.5, 0.9, 0.99]:
    summands = [discountfactor**t for t in range(10000)]
    plt.plot(summands, label=discountfactor)

    total_value = np.sum(summands)
    print("Discount factor {dcf:3.2f}: Total {total:5.1f}"\
        .format(dcf=discountfactor, total=total_value))

plt.legend(); plt.ylabel('Present value'); plt.xlabel('Timestep');
plt.xlim(0,100);
```

```
Discount factor 0.10: Total   1.1
Discount factor 0.50: Total   2.0
Discount factor 0.90: Total  10.0
Discount factor 0.99: Total 100.0
```
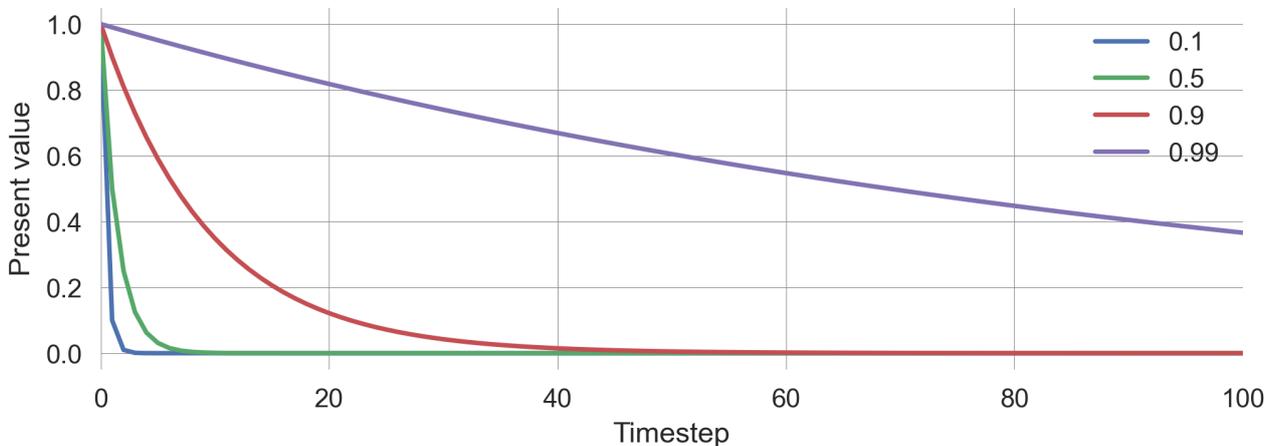

Figure 6.4

Here, we used the Python string method `format` to print the results in a readable way. It can be used to insert variables into a string. The curly brackets `{}` are placeholders for the variables, and the variables are passed to the `format` method as arguments. The colon `:` inside the curly brackets is used to format the output. For example, `:3.2f` formats the number as a floating-point number with three digits before and two digits after the decimal point.

***Normalized* goal**. To account for the fact that the total value depends on the level of discounting, even if the reward stream is constant, we can normalize the goal as follows,

$$G_t = (1-\gamma) \sum_{\tau=t}^{\infty} \gamma^\tau R_{t+\tau+1},$$

where $1-\gamma$ is a normalizing factor and $R_{t+\tau+1}$ is the reward received at time step $t+\tau+1$.

```python
for dcf in [0.1, 0.5, 0.9, 0.99]:
    summands = [dcf**t for t in range(10000)]
    normalizing = 1-dcf
    total_value = normalizing * np.sum(summands)
    print("Discount factor {dcf:3.2f}: Total {total:5.1f}".format(dcf=dcf,
    ↪ total=total_value))
```

```
Discount factor 0.10: Total    1.0
Discount factor 0.50: Total    1.0
Discount factor 0.90: Total    1.0
Discount factor 0.99: Total    1.0
```

With normalization, the discount factor parameter γ expresses how much the agent **cares for the future** without influencing the scale of the total value. That way, the outcomes of different discount factors can be **compared** with each other.

For example, the normalized discounted gain with a discount factor of $\gamma = 0.9$ of the last ensemble run is

```python
dcf = 0.9
(1-dcf) * np.sum([dcf**t * reward_ensemble[-1, t] for t in range(500)])
```

```
0.9254059133343876
```

Bellman equation. Regardless of the goal formulation, the agent's gains G_t at successive time steps relate to each other in an important way:

$$
\begin{align}
G_t &= (1-\gamma) \sum_{\tau=t}^{\infty} \gamma^\tau R_{t+\tau+1} \tag{6.1}\\
&= (1-\gamma)\left(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \cdots\right) \tag{6.2}\\
&= (1-\gamma)\left(R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \cdots)\right) \tag{6.3}\\
&= (1-\gamma)R_{t+1} + \gamma(1-\gamma)(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \cdots) \tag{6.4}\\
&= (1-\gamma)R_{t+1} + \gamma G_{t+1}, \tag{6.5}
\end{align}
$$

The gain G_t is composed of the current short-term reward and the (discounted) value of the future gains. This recursive relationship is known as the **Bellman equation** and is the foundation of many solution methods for MDPs, such as dynamic programming and reinforcement learning.

For example, we can test the Bellman equation by comparing the gain at time step t with the short-term reward at time step t and the gain at time step $t+1$.

```
dcf = 0.9
G0 = (1-dcf) * np.sum([dcf**t * reward_ensemble[-1, t] for t in range(0, 500)])
G1 = (1-dcf) * np.sum([dcf**t * reward_ensemble[-1, t+1] for t in range(0, 499)])

np.allclose((1-dcf) * reward_ensemble[-1, 0] + dcf * G1, G0)
```

```
True
```

Why does that work even though we have a finite time horizon of 500 simulation timesteps here? From Figure 6.4 we observe that for a discount factor $\gamma = 0.9$, the contributions of rewards for timesteps above $t > 100$ are practically zero. So, with a simulation time of 500 timesteps, we are well above the time horizon the agent cares about. This example illustrates not only the power of the Bellman equation. It also shows how a discount factor induces a timescale the agent cares about.

Goals or gains are defined over individual reward streams or trajectories. These may be stochastic beyond the agent's control. Therefore, the agent's course of action should consider the **expected** gains, i.e., the average gains over all possible reward streams, given a policy $\mathbf{x}$.

6.4.2 Value functions

Value functions are defined to be the expected gain G_t for a policy x, given a state or state-action pair. They are helpful in finding a good policy since the best policy will yield the highest value.

Given a policy $\mathbf{x}$, we define the **state value**, $v_{\mathbf{x}}(s)$, as the expected gain, $\mathbb{E}_{\mathbf{x}}[G_t | S_t = s]$, when starting in state s and the following the policy $\mathbf{x}$,

$$v_{\mathbf{x}}(s) := \mathbb{E}_{\mathbf{x}}[G_t | S_t = s] = (1 - \gamma) \mathbb{E}_{\mathbf{x}} \left[\sum_{\tau=t}^{\infty} \gamma^{\tau} R_{t+\tau+1} | S_t = s \right], \quad \text{for all } s \in \mathcal{S},$$

Analogously, we define the **state-action value**, $q_{\mathbf{x}}(s, a)$, as the expected gain when starting in state s and executing action a, and from then on following policy $\mathbf{x}$,

$$q_{\mathbf{x}}(s, a) := \mathbb{E}_X[G(t) | s(t) = s, a(t) = a].$$

$$q_{\mathbf{x}}(s, a) := \mathbb{E}_{\mathbf{x}}[G_t | S_t = s, A_t = a] = (1 - \gamma) \mathbb{E}_{\mathbf{x}} \left[\sum_{\tau=t}^{\infty} \gamma^{\tau} R_{t+\tau+1} | S_t = s, A_t = a \right], \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}.$$

How is that useful?

1) **State values let us compare strategies.** A strategy $\mathbf{x}$ is better than a strategy $\mathbf{y}$ iff for all states s: $v_{\mathbf{x}}(s) > v_{\mathbf{y}}(s)$.

2) **The best strategy yields the highest value**. At least one strategy is always better than or equal to all other strategies. That is an *optimal strategy* $\mathbf{x}_*$ with the *optimal state value* $v_*(s) := \max_{\mathbf{x}} v_{\mathbf{x}}(s), \forall s$.

3) **Highest state-action values indicate the best action**. If we knew the *optimal state-action value*, $q_*(s, a) := \max_{\mathbf{x}} q_{\mathbf{x}}(s, a), \forall s, a$, we can simply assign nonzero probability at each state s only to actions which yield maximum value, $\max_{\tilde{a}} q_*(s, \tilde{a})$.

The **beauty** of state(-action) **values**, in general, and optimal state(-action) values, in particular, is that they **encapsulate all relevant information about future environmental dynamics** with all inherent stochasticity **into short-term actionable numbers**. *Relevant* means relevant to the agent regarding its goal function. State-action values represent the short-term consequences of actions in each state regarding the long-term goal. Optimal state-action values allow for selecting the best actions, irrespective of knowing potential successor states and their values or any details about environmental dynamics. Having such values would save the agent enormous cognitive computational demands every time it must make a decision.

The only problem we are left with is, how to compute a policy's state(-action) values?

6.4.3 Bellman equation

We convert the recursive relationship of the goal function (**?@eq-bellman1**) to state values,

$$v_{\mathbf{x}}(s) = \mathbb{E}_{\mathbf{x}}[G_t | S_t = s] \tag{6.6}$$

$$= \mathbb{E}_{\mathbf{x}}[(1-\gamma)R_{t+1} + \gamma G_{t+1} | S_t = s] \tag{6.7}$$

$$= (1-\gamma)\mathbb{E}_{\mathbf{x}}[R_{t+1} | S_t = s] + \gamma \mathbb{E}_{\mathbf{x}}[G_{t+1} | S_{t+1} = s'] \tag{6.8}$$

$$= (1-\gamma)R_{\mathbf{x}}(s) + \gamma \sum_{s'} T_{\mathbf{x}}(s, s')v_{\mathbf{x}}(s'), \tag{6.9}$$

where $R_{\mathbf{x}}(s)$ is the expected reward in state s under policy $\mathbf{x}$ and $T_{\mathbf{x}}(s, s')$ is the expected transition probability from state s to state s' under policy $\mathbf{x}$.

The expected state reward $R_{\mathbf{x}}(s)$ is given by

$$R_{\mathbf{x}}(s) = \sum_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} x(s, a)T(s, a, s')R(s, a, s'),$$

which can be neatly translated into Python using the `numpy.einsum` method.

```
s, a, s_ = 0, 1, 2  # defining indices for convenicence
Rs = np.einsum(X, [s, a], T, [s, a, s_], R, [s, a, s_], [s]); Rs
```

```
array([0.90068162, 0.        ])
```

The recursive equation is called the *Bellman equation* in honor of Richard Bellman and his pioneering work (Bellman 1957). The recursive relationship is exploited in several algorithmic ways to compute the values or even approximate the optimal state values. In recent years, it became possible to approximate optimal state values with deep neural networks, a technique known as *deep reinforcement learning* (Mnih et al., 2015), allowing for solving high-dimensional MDPs with many - even infinitely many - states and actions. This is a fascinating field of research, which we will not cover in this course. I recommend the interested reader to start from the excellent (introduction to reinforcement learning by Sutton & Barto, 2018).

Our focus lies on a **transparent way of modeling** human-environment interactions. We use MDPs as a framework to improve our conceptual understanding of decision-making under uncertainty. Specifically, we exemplify that with the trade-off between short-term and long-term welfare.

Using an MDP framework, our models are formulated in a way that - in principle - can scale to high-dimensional systems. The trade-off is, however, the computational cost of solving high-dimensional

MDPs. The more complex and "realistic" a model, the less we can understand how the outcome depends on the model's specifications.

As these model specifications are often highly uncertain in the context of sustainability and global change (Polasky et al., 2011), it is very **likely that we end up with an optimal policy for a wrong model** that is not useful for decision-making. It might even be harmful, conveying a false sense of optimality. This problem gets worse with the complexity of the model. The more model parameters we have to specify as the input to the model, the more sources of possible but unconscious uncertainty there is.

Therefore, we will focus on minimalistic models but take a radical stance to account for parameter uncertainty to keep the analysis and interpretation transparent. Thus, in the following, we derive an analytical expression how to compute the state values for a given policy.

We write the Bellman equation in matrix form,

$$\mathbf{v_x} = (1 - \gamma)\mathbf{R_x} + \gamma\underline{\mathbf{T}}_\mathbf{x}\mathbf{v_x}$$

where $\mathbf{R_x}$ is the vector of expected state rewards $R_\mathbf{x}(s)$, $\underline{\mathbf{T}}_\mathbf{x}$ is the transition matrix, and $\mathbf{v_x}$ is the vector of state values under policy $\mathbf{x}$. Thus, $\mathbf{v_x}$ and $\mathbf{R_x}$ are vectors of dimension Z, i.e., the number of states, and $\underline{\mathbf{T}}_\mathbf{x}$ is the transition matrix of dimension $Z \times Z$.

We can solve this equation for $\mathbf{v_x}$,

$$\mathbf{v_x} = (1 - \gamma)\mathbf{R_x} + \gamma\underline{\mathbf{T}}_\mathbf{x}\mathbf{v_x} \tag{6.10}$$
$$\mathbf{v_x} - \gamma\underline{\mathbf{T}}_\mathbf{x}\mathbf{v_x} = (1 - \gamma)\mathbf{R_x} \tag{6.11}$$
$$(\mathbb{1}_Z - \gamma\underline{\mathbf{T}}_\mathbf{x})\mathbf{v_x} = (1 - \gamma)\mathbf{R_x} \tag{6.12}$$
$$(\mathbb{1}_Z - \gamma\underline{\mathbf{T}}_\mathbf{x})^{-1}(\mathbb{1}_Z - \gamma\underline{\mathbf{T}}_\mathbf{x})\mathbf{v_x} = (1 - \gamma)(\mathbb{1}_Z - \gamma\underline{\mathbf{T}}_\mathbf{x})^{-1}\mathbf{R_x} \tag{6.13}$$
$$\mathbf{v_x} = (1 - \gamma)(\mathbb{1}_Z - \gamma\underline{\mathbf{T}}_\mathbf{x})^{-1}\mathbf{R_x}, \tag{6.14}$$
$$\tag{6.15}$$

where $\mathbb{1}_Z$ is the identity matrix of dimension Z.

Thus, to compute state value, we must invert a $Z \times Z$-matrix, which is computationly infeasable for large MDPs. For low-dimensional models, however, it works perfectly fine and can even be executed analytically.

In Python, an identity matrix can be created with the `eye` function from the `numpy` package.

```
np.eye(2)
```

```
array([[1., 0.],
       [0., 1.]])
```

We define a function to compute the state values given a policy, a transition tensor, a reward tensor, and a discount factor. The function returns a vector of state values. We use the `inv` function from the `numpy.linalg` package to invert the matrix.

```
def compute_statevalues(
    policy_Xsa, transitions_Tsas, rewards_Rsas, discountfactor):
    s, a, s_ = 0, 1, 2  # defining indices for convenicence
    Tss = np.einsum(policy_Xsa, [s, a], transitions_Tsas, [s, a, s_], [s,s_])
    Rs = np.einsum(policy_Xsa, [s, a], transitions_Tsas, [s, a, s_],
                   rewards_Rsas, [s, a, s_], [s])
    inv = np.linalg.inv((np.eye(2) - discountfactor*Tss))
    Vs = (1-discountfactor) * np.einsum(inv, [s,s_], Rs, [s_], [s])
    return Vs
```

```
Vs = compute_statevalues(X, T, R, 0.9); Vs
```

```
array([0.7220388 , 0.13059414])
```

Thus, in contrast to the expected state rewards, the long-term value of the degraded state is above the immediate reward of the degraded state, $r_d = 0$.

```
Rs
```

```
array([0.90068162, 0.        ])
```

In the state value $v_{\mathbf{x}}(\mathsf{d})$ of the degraded state, the agent anticipates the recovery of the environment and the return to the prosperous state. Likewise, the agent anticipates the collapse of the environment and the loss of the prosperous state in the state value of the prosperous state. Hence, $v_{\mathbf{x}}(\mathsf{p})$ is smaller than the expected reward of the prosperous state, $R_{\mathbf{x}}(\mathsf{p})$. The expected state rewards only consider the immediate possible transitions, while the state values also account for the long-term consequences of these transitions.

How do these values depend on the discount factor γ?

We define an array of linearly spaced values of different discount factors,

```
discountfactors = np.linspace(0.001, 0.9999, 301)
```

We then compute the state value for each discount-factor value using a list comprehension,

```
values = np.array([compute_statevalues(X, T, R, dcf) for dcf in discountfactors])
values.shape
```

```
(301, 2)
```

We plot the state values along the discount factors on the x-axis. We also include the expected state rewards, which are independent of the discount factor.

```
plt.plot(discountfactors, Rs[0]*np.ones_like(discountfactors), label='$R(p)$',
↪    c='green', ls='--');
plt.plot(discountfactors, values[:, 0], label='$v(p)$', c='green');
plt.plot(discountfactors, values[:, 1], label='$v(d)$', c='brown');
plt.plot(discountfactors, Rs[1]*np.ones_like(discountfactors), label='$R(d)$',
↪    c='brown', ls='--');
plt.legend(); plt.xlabel('Discount factor'); plt.ylabel('Value')
```

```
Text(0, 0.5, 'Value')
```
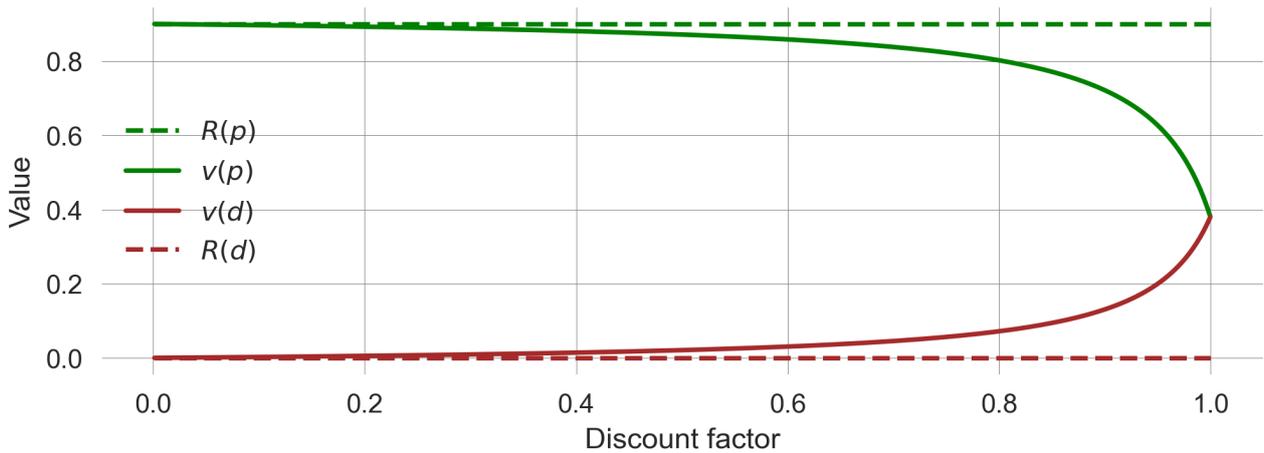

When the discount factor is close to zero, $\gamma = 0$, the values equal the average immediate rewards.

When the discount factor $\gamma \to 1$, the state values for the prosperous and the degraded state approach each other.

Last, the state values change more for large $\gamma > 0.85$ than for lower γ.

So far, we investigated how to compute the state values for a given policy and use a random policy as an example. To eventually answer what the agent should do, we must compare multiple policies and find the best one.

6.5 Optimal policies

The key question of our example model is, when is it better to play safe, and when is it better to be risky? From our model definition, we can easily see that, in the degraded state, it is always better to play safe as this is the only way to recover to the more rewarding, prosperous state. But what about the prosperous state?

6.5.1 Numerical computation

We define two policies, a **safe** policy, $\mathbf{x}_{\text{safe}}$, where the agent always chooses the safe action and a **risky** policy, $\mathbf{x}_{\text{risky}}$, where the agent always chooses the risky action in the prosperous state.

```
Xsafe = np.array([[1,0],[1,0]])
Xrisk = np.array([[0,1],[1,0]])
```

For each of these policies, we compute the state values with our `compute_statevalues` function,

```
V_safe = np.array([compute_statevalues(Xsafe, T, R, dcf) for dcf in
↪  discountfactors])
V_risk = np.array([compute_statevalues(Xrisk, T, R, dcf) for dcf in
↪  discountfactors])
```

and plot these values for each policy and each state as

141

```
plt.plot(discountfactors, V_safe[:, 0], label='$v_{safe}(p)$', color='blue');
plt.plot(discountfactors, V_safe[:, 1], label='$v_{safe}(d)$', color='blue',
↪   lw=0.4);
plt.plot(discountfactors, V_risk[:, 0], label='$v_{risk}(p)$', color='red');
plt.plot(discountfactors, V_risk[:, 1], label='$v_{risk}(d)$', color='red', lw=0.4);
plt.legend(); plt.xlabel('Discount factor'); plt.ylabel('Value');
```
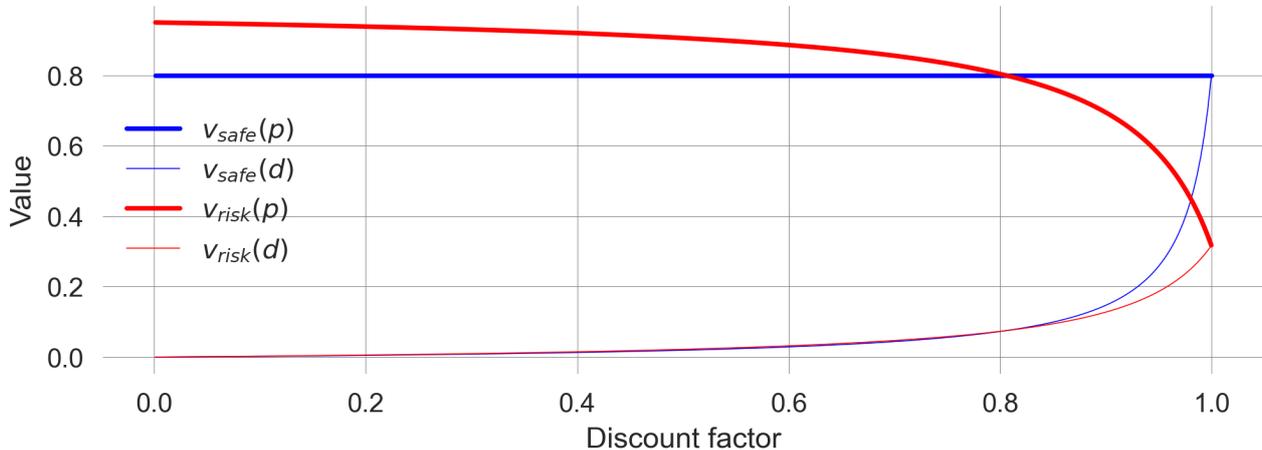

We find a **critical discount factor** $\hat{\gamma}$, where the optimal policy changes. Below $\hat{\gamma}$, the agent acts optimally by choosing the risky policy. Above the critical discount factor, $\hat{\gamma}$, the agent acts optimally by choosing the safe policy.

Hence, when the agent cares enough about the future, it is better to be safe than sorry, even if this means giving up immediate, short-term welfare ($r_s < r_r$).

But how does this result depend on the other parameters, p_c, p_r, r_s, r_r, r_d?

This investigates how the optimal policy depends on all parameters of the model; we first **define general transition and reward functions** that return a transition and reward tensor, given our model parameters. We make these functions general by passing the most general datatype to the respective `numpy.arrays`, i.e., `dtype=object`. This allows us to store arbitrary Python objects in the arrays, such as float numbers or symbolic expressions.

```
def get_transitions(pc, pr):
    c=0; r=1; p=0; d=1  # for reference we define these as function-local variables
    T = np.zeros((2,2,2), dtype=object)
    T[p,c,d] = 0              # Cautious action guarantees prosperous state
    T[p,c,p] = 1    #
    T[p,r,d] = pc;           # Risky action risks collapse
    T[p,r,p] = 1-T[p,r,d]    # ... but collapse may not happen
    T[d,c,p] = pr            # Recovery only possible with cautious action
    T[d,c,d] = 1-T[d,c,p]    # ... but recovery might not happen
    T[d,r,p] = 0             # Risky action remains at degraded state
    T[d,r,d] = 1
    return T
```

```
def get_rewards(rs, rr=1, rd=0):
    c=0; r=1; p=0; d=1  # for reference we define these as function-local variables
    R = np.zeros((2,2,2), dtype=object)
```

```
    R[p,c,p] = rs            # The cautious action at the prosperous state
↪ guarantees the safe reward
    R[p,r,p] = rr            # The risky action can yield the risky reward if the
↪ environment remains at p
    R[d,:,:] = R[:,:,d] = rd # Otherwise, the agent receives rd
    return R
```

Now, we can create transition and reward tensors flexibly. As we want to perform numerical computations, we specify the data type of the arrays to be float numbers.

```
T = get_transitions(0.04, 0.1).astype(float)
T
```

```
array([[[1.  , 0.  ],
        [0.96, 0.04]],

       [[0.1 , 0.9 ],
        [0.  , 1.  ]]])
```

```
R = get_rewards(0.7).astype(float)
R
```

```
array([[[0.7, 0. ],
        [1. , 0. ]],

       [[0. , 0. ],
        [0. , 0. ]]])
```

Let's assume we want to know how the critical discount factor $\hat{\gamma}$ depends on the collapse probability p_c for a given recovery probability $p_r = 0.01$ and safe reward $r_s = 0.8$, a risky reward $r_r = 1.0$ and a degraded reward $r_d = 0.0$. We define these quantities as

```
pr = 0.01
rs = 0.5
rr = 1.0
rd = 0.0
```

and let the discount factor and collapse probabilities run from almost zero to almost one with a resolution of 301 elements,

```
discountfactors = np.linspace(0.0001, 0.9999, 301)
collapseprobabilities = np.linspace(0.0001, 0.9999, 301)
```

We will go through each combination of discount factors and collapse probabilities, compute the state values for both policies, compare them, and store the result in a *data container*. We prepare this *data container* by

```
risky_optimal_data_container = np.zeros((discountfactors.size,
↪   collapseprobabilities.size, 2))
```

Now, we are ready to execute our simulation. We loop through each discount factor and for each discount factor through each collapse probability, obtain our new transition matrix, compute the state values, and store them in our data container. The Jupyter cell magic %%time shows us how long it took to execute that cell.

```
%%time
for i, dcf in enumerate(discountfactors):
    for j, pc in enumerate(collapseprobabilities):
        T = get_transitions(pc, pr).astype(float)
        R = get_rewards(rs, rr, rd).astype(float)
        Vs_risk = compute_statevalues(Xrisk, T, R, dcf)
        Vs_safe = compute_statevalues(Xsafe, T, R, dcf)
        risky_optimal_data_container[i, j, :] = Vs_risk > Vs_safe
```

```
CPU times: user 10.9 s, sys: 1.94 s, total: 12.8 s
Wall time: 8.7 s
```

We noticeably have to wait for the result!

```
plt.subplot(131); plt.xticks([]); plt.yticks([]);
plt.subplot(133); plt.xticks([]); plt.yticks([]);
plt.subplot(132) # just to center the plot in the middle

plt.pcolormesh(collapseprobabilities, discountfactors,
               risky_optimal_data_container[:,:,0], cmap='bwr')
plt.ylabel('Discount factor'); plt.xlabel('Collapse probabiliy');
```
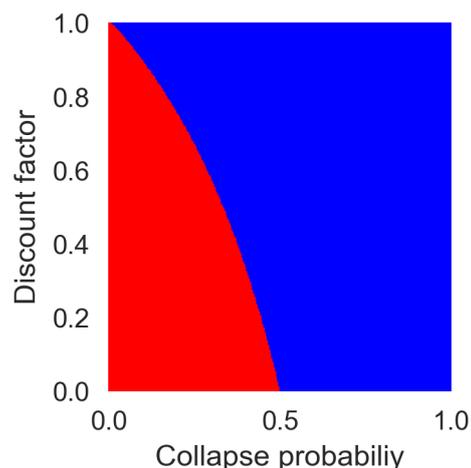

The higher the collapse probability, the lower the critical discount factor. When the collapse is more likely, less future care is required to evaluate the safe policy as optimal. When the collapse probability is zero ($p_c = 0$), the critical discount factor is one ($\hat{\gamma} = 1$), and the agent should always choose the risky policy, as the environment cannot be destroyed.

When the discount factor is zero ($\gamma = 0$), the critical collapse probability is a half $\hat{p}_c = 0.5$. If an environmental collapse under the risky action is more likely $p_c > \hat{p}_c$, the agent should always choose the safe policy and vice versa. But where does the value 0.5 come from? Intuitively, it is the ratio between the safe and the risky reward, r_s/r_r.

But how can we be sure? And wouldn't it be great, if we could speed up the computation time somehow?

The solution to both questions lies in a symbolic computation of the critical parameter values $\hat{\gamma}, \hat{p}_c, \hat{p}_r, \hat{r}_s, \hat{r}_r, \hat{r}_d$.

6.5.2 Symbolic computation

We define symbolic expressions for our model parameters and obtain the corresponding transition and reward tensors,

```
pc, pr = sp.symbols("p_c, p_r")
T = sp.Array(get_transitions(pc, pr))
T
```

$$\left[\begin{bmatrix} 1 & 0 \\ 1-p_c & p_c \end{bmatrix} \begin{bmatrix} p_r & 1-p_r \\ 0 & 1 \end{bmatrix} \right]$$

```
rs, rr, rd = sp.symbols("r_s r_r r_d")
R = sp.Array(get_rewards(rs, rr, rd))
R
```

$$\left[\begin{bmatrix} r_s & r_d \\ r_r & r_d \end{bmatrix} \begin{bmatrix} r_d & r_d \\ r_d & r_d \end{bmatrix} \right]$$

As before, we also define a risky and a safe policy, now as symbolic variables,

```
Xsafe = sp.Array([[1,0],[1,0]])
Xrisk = sp.Array([[0,1],[1,0]])
```

and also the discount factor as a symbolic variable

```
dcf = sp.symbols("gamma")
dcf
```

γ

Luckily, we only have to change our `compute_statevalues` slightly, (since the `np.einsum` function also works with Sympy expressions)

```
def symbolic_statevalues(policy_Xsa, transitions_Tsas, rewards_Rsas,
↪   discountfactor=dcf):
    s, a, s_ = 0, 1, 2  # defining indices for convenicence
    Tss = sp.Matrix(np.einsum(policy_Xsa, [s, a], transitions_Tsas, [s, a, s_],
↪   [s,s_]))
    Rs = sp.Array(np.einsum(policy_Xsa, [s, a], transitions_Tsas, [s, a, s_],
↪   rewards_Rsas, [s, a, s_], [s]))
```

```
    inv = (sp.eye(2) - discountfactor*Tss).inv(); inv.simplify()  # sp.simplify()
↪   often helps
    Vs = (1-discountfactor) * sp.Matrix(np.einsum(inv, [s,s_], Rs, [s_], [s]));
↪   Vs.simplify()
    return Vs
```

The symbolic expressions of the state values for the risky policy are

```
symbolic_statevalues(Xrisk, T, R)
```

$$\begin{bmatrix} \frac{\gamma p_c p_r r_d - \gamma p_c p_r r_r + \gamma p_c r_r + \gamma p_r r_r - \gamma r_r + p_c r_d - p_c r_r + r_r}{\gamma p_c + \gamma p_r - \gamma + 1} \\ \frac{\gamma p_c p_r r_d - \gamma p_c p_r r_r + \gamma p_c r_d + \gamma p_r r_r - \gamma r_d + r_d}{\gamma p_c + \gamma p_r - \gamma + 1} \end{bmatrix}$$

and for the safe policy, are

```
symbolic_statevalues(Xsafe, T, R)
```

$$\begin{bmatrix} r_s \\ \frac{\gamma p_r r_s - \gamma r_d + r_d}{\gamma p_r - \gamma + 1} \end{bmatrix}$$

To check whether the risky policy is optimal, we subtract the value of the safe policy from the risky policy's value at the prosperous state 0.

```
risky_optimal = sp.simplify(symbolic_statevalues(Xrisk, T, R)[0])\
    - sp.simplify(symbolic_statevalues(Xsafe, T, R)[0])
sp.simplify(risky_optimal)
```

$$\frac{\gamma p_c p_r r_d - \gamma p_c p_r r_r + \gamma p_c r_r + \gamma p_r r_r - \gamma r_r + p_c r_d - p_c r_r + r_r - r_s \left(\gamma p_c + \gamma p_r - \gamma + 1\right)}{\gamma p_c + \gamma p_r - \gamma + 1}$$

We can solve this equation for any variable. For example, to check the critical collapse probability for an entirely myopic agent with zero care for the future, we solve the equation for the collapse probability p_c and substitute the discount factor $\gamma = 0$.

```
sp.solve(risky_optimal, pc)[0].subs(dcf, 0)
```

$$\frac{-r_r + r_s}{r_d - r_r}$$

Thus, **our intuition about the ratio between r_s and r_r was not entirely correct**. In fact, we can simplify the three reward parameters r_r, r_s, and r_d. As it is irrelevant to the agent's decision whether all rewards are multiplied by a factor or all rewards are added by a constant, we can set $r_d = 0$ and $r_s = 1$ without loss of generality.

Setting the degraded reward to zero, $r_d = 0$, and the risky reward to one, $r_r = 1$, improves the transparency and interpretability of the model.

```
sp.solve(risky_optimal.subs(rr, 1).subs(rd, 0), pc)[0].subs(dcf, 0)
```

$$1 - r_s$$

Thus, the critical collapse probability $\hat{p}_c$ for $\gamma = 0$ is given by the $\hat{p}_c = 1 - r_s$.

By using symbolic calculations, we improve the transparency and interpretability of our model.

How can we speed up the computation time with `sympy`?

6.5.3 Efficient computation

To create a plot as above, it is an excellent strategy to **convert this symbolic expression into a numeric function**. In `sympy`, this is done with the `sympy.lambdify` function, (called as `sp.lambdify((<symbolic parameters>), <symbolic expression to be turned into a numeric function>)`

```
risky_optimal_func = sp.lambdify((pc,pr,dcf,rs,rr,rd), risky_optimal)
```

For example, we can now execute `risky_optimal_func` for $p_c = 0.2$, $p_r = 0.01$, $\gamma = 0.9$, $r_s = 0.5$, $r_r = 1.0$, and $r_d = 0.0$ as

```
risky_optimal_func(0.2, 0.01, 0.9, 0.5, 1.0, 0.0)
```

```
-0.19826989619377183
```

and learn that the risky policy is *not* optimal in this case.

However, the big advantage of a *lambdified* function is that we can apply it in **vectorized form**. This means the parameters don't have to be single numbers. They can be vectors or even larger tensors. See, for example,

```
gams = np.linspace(0.0001, 0.9999, 9)
risky_optimal_func(0.2, 0.01, gams, 0.5, 1.0, 0.0)
```

```
array([ 0.299984  ,  0.27779382,  0.25014334,  0.21473437,  0.1677686 ,
        0.10248474,  0.00556964, -0.15331544, -0.46154209])
```

Thus, to recreate our example from above, where we wanted to know how the critical discount factor $\hat{\gamma}$ depends on the collapse probability p_c for given other parameters, we can now use the `risky_optimal_func` directly in vectorized form.

However, if we simply put two vectors (of the same dimension) inside the function, we only get

```
discountfactors = np.linspace(0.0001, 0.9999, 9)
collapseprobabilities = np.linspace(0.0001, 0.9999, 9)
risky_optimal_func(collapseprobabilities, 0.01, discountfactors, 0.5, 1.0, 0.0)
```

```
array([ 0.49989999,  0.3595776 ,  0.19241376,  0.01072705, -0.16556291,
       -0.31475139, -0.42146066, -0.48138804, -0.499999  ])
```

147

we only get one vector (of the same dimension) out. This is, beacuse vectorization groups changes along a dimension together.

This means we need to separate the variation in the discount factors (in `discountfactors`) and the variation in the collapse probabilities (in `collapseprobabilities`) into different dimensions. Luckily, we don't have to do that manually. The `numpy` method `numpy.meshrid` exactly fits this purpose. For example,

```
discountfactors = [0.8, 0.9]
collapseprobabilities = [0.1, 0.2, 0.3]
np.meshgrid(discountfactors, collapseprobabilities)
```

```
[array([[0.8, 0.9],
        [0.8, 0.9],
        [0.8, 0.9]]),
 array([[0.1, 0.1],
        [0.2, 0.2],
        [0.3, 0.3]])]
```

In practise, we can use a meshgrid as follows,

```
pr_ = 0.01
rs_ = 0.5
rr_ = 1.0
rd_ = 0.0
discountfactors = np.linspace(0.0001, 0.9999, 301)
collapseprobabilities = np.linspace(0.0001, 0.9999, 301)
DCFs, PCs = np.meshgrid(discountfactors, collapseprobabilities)
```

```
risky_optimal_func(PCs, pr_, DCFs, rs_, rr_, rd_)
```

```
array([[ 0.49989999,  0.49989966,  0.49989932, ...,  0.49398743,
         0.49251766,  0.49009707],
       [ 0.49656699,  0.49655555,  0.49654403, ...,  0.32758543,
         0.29387422,  0.24378043],
       [ 0.49323399,  0.49321152,  0.4931889 , ...,  0.20822659,
         0.1607447 ,  0.09481031],
       ...,
       [-0.49323534, -0.49325773, -0.49328013, ..., -0.4998874 ,
        -0.49990965, -0.4999319 ],
       [-0.49656768, -0.49657908, -0.49659048, ..., -0.49994305,
        -0.49995431, -0.49996556],
       [-0.49990001, -0.49990034, -0.49990068, ..., -0.49999835,
        -0.49999867, -0.499999  ]])
```

Notice how quickly that was compared to our previous calculations!

To time how long the cell execution takes more precisely, we can use the `%%timeit` cell magic command:

```
%%timeit
risky_optimal_func(PCs, pr_, DCFs, rs_, rr_, rd_)
```

```
1.92 ms ± 396  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

It executes the cell multiple times and presents us with a short summary statistic. Compare the average runtime of the cell with the numerical computation. It is around 5000 times faster!

Thus, we can summarize the lambdified `sympy` expression into a `plot_parameter_space` function:

```
def plot_parameter_space(safe_reward=0.5, risky_reward=1.0, degraded_reward=0.0,
↪  recov_prop=0.05):
    plt.subplot(131); plt.xticks([]); plt.yticks([]);
    plt.subplot(133); plt.xticks([]); plt.yticks([]);
    plt.subplot(132) # just to center the plot in the middle

    resolution=251

    X = np.linspace(0.0001, 0.9999, resolution)
    Y = np.linspace(0.0001, 0.9999, resolution)
    XX, YY = np.meshgrid(X, Y)

    ro = risky_optimal_func(XX, recov_prop, YY, safe_reward, risky_reward,
↪  degraded_reward)
    plt.pcolormesh(XX, YY, ro, cmap='bwr', vmin=-0.1, vmax=0.1)
    plt.ylabel('Discount factor'); plt.xlabel('Collapse leverage');
```

```
plot_parameter_space()
```
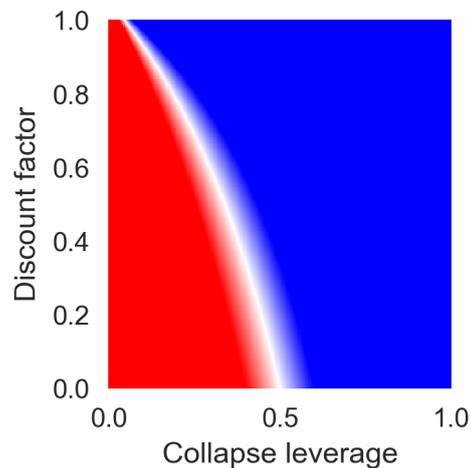

When working with this Jupyter Notebook directly, we can interactively explore the parameter space of the model.

For an agent that does not discount the future at all, i.e., with $\gamma \to 1$, the critical collapse leverage yields,

```
sp.simplify(sp.solve(risky_optimal.subs(rr,1).subs(rd, 0), pc)[0].subs(dcf, 1))
```

$$\frac{p_r\,(1 - r_s)}{p_r + r_s}$$

Thus, if there is zero recovery probability $p_r = 0$, the safe policy is optimal regardless of the relative reward $0 < r_s < 1$.

```
plot_parameter_space(recov_prop=0.0)
```
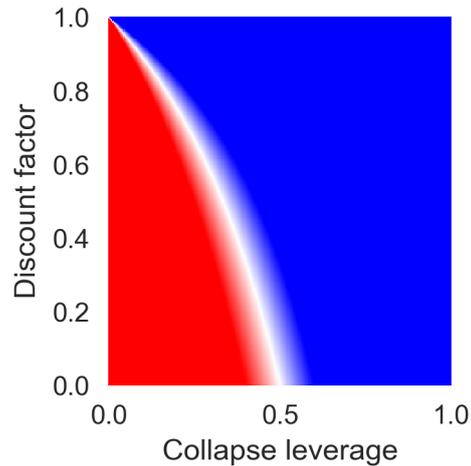

If $p_r > 0$, then it depends on the relative reward r_s whether the safe or the risky policy is optimal for a fully farsighted agent.

Taken together, by using symbolic computation from the sympy package, we improve interpretability, transparancey and computational efficiency of our model.

6.6 Learning goals revisited

- We **introduced** the elements of a Markov Decision Process (MDP) and discussed how they relate to applications in human-environment interactions
- We **simulateed** and **visualized** the time-evolution of an MDP.
- We **covered** what value functions are, why they are usful and how to realte to the agent's goal and Bellman equation.
- We **computed** value functions *in serveral ways* and **visualized** how the best policy depends on other model parameters.

7 Strategic Interactions

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

7.1 Motivation | Collective action for sustainability

Consider the following questions:

- Do you think climate change is a significant problem the world needs to address?
- Do you think the world has been trying?
- Do you think the world has succeeded?

Scott Barrett asked these questions at the beginning of a talk, which I can highly recommend watching. The talk is called Climate Change Diplomacy: a Most Dangerous Game and is given at the London School of Economics.

The typical answers to these questions raise the point of **why it is so difficult to succeed in stopping climate change despite recognizing the problem and trying to solve it.**

The outcome depends on all! Carbon dioxide (CO2) is the most prevalent greenhouse gas driving global climate change. CO2 from different sources (fossil fuels, burned biomass, land ecosystems, oceans) is being added to Earth's atmosphere from various locations over the globe. Then, it mixes relatively fast in the atmosphere, i.e., the consequences for a region do not depend on how much that region emits but on the overall emissions.

NASA Scientific Visualization Studio | https://svs.gsfc.nasa.gov/5110
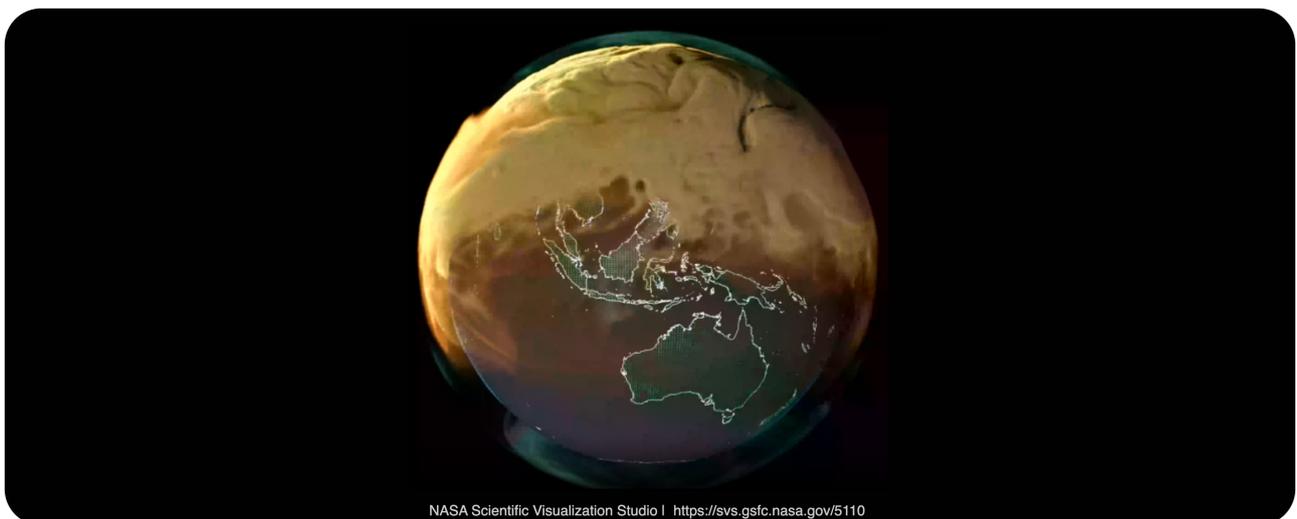
Figure 7.1: This visualization from NASA shows the CO2 added to Earth's atmosphere over 2021, split into four major contributors: fossil fuels in orange, burning biomass in red, land ecosystems in green, and the ocean in blue. The dots on the surface also show how green land ecosystems and the ocean in blue absorb atmospheric carbon dioxide. Though the land and oceans are each carbon sinks in a global sense, individual locations can be sources at different times.

To stabilize the climate, (net) emissions have to go to zero.
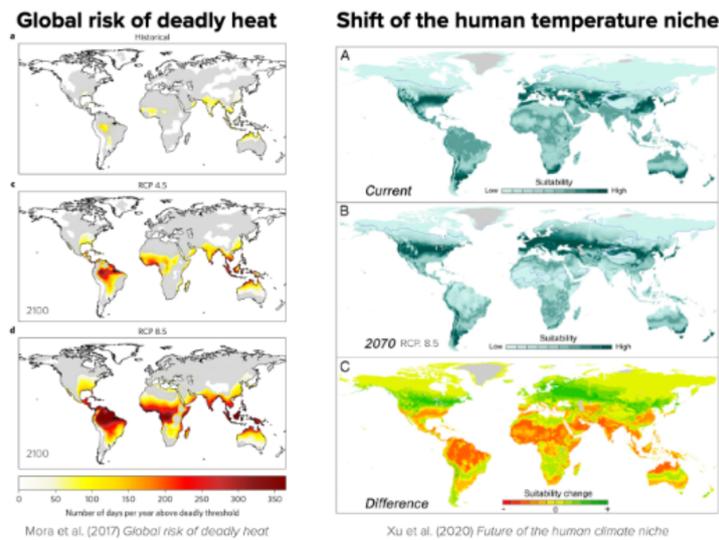
Figure 7.2: Linear damages of climate change

While the climate is the most discussed example, the collective action problem extends to the whole **planetary commons**.

7.1.1 Advantages of game theory

In this lecture, we introduce the basics of mathematical game theory to uncover the underlying mechanisms of strategic interactions. We will see how the behavior of individuals can lead to collective outcomes that are not in the interest of any individual. We will also discuss possible mechanisms and variations of the situation that help to overcome these challenges and will acknowledge the limitations of these variations.

A **mathematical game** describes an action situation where an **outcome** relevant to an individual **depends on** at least one **other actor**. This is why we speak of **inter**actions instead of only actions of a single-agent action situation. The **strategic** aspect comes into play when the actors are aware of the interdependence and can anticipate the actions of others.

7.1.2 Learning goals

After this lecture, students will be able to:

- Apply game theory to model multi-agent action situations
- Resolve games by finding Nash equilibria
- Describe the dimensions of a social dilemma
- Explain two special kinds of games: *agreement games* and *threshold public goods*, and how they relate to the dimensions of a social dilemma.

7.2 Game theory

Life is a game. At least in theory.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
from ipywidgets import interact, fixed

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;
```

Game theory in itself is diverse. Here, we focus on **normal-form games** with the following elements.

- A finate set of N agents $\mathcal{I} = \{2, \dots, N\}$ participating in an interaction.
- For each agent $i \in \mathcal{I}$, a discrete set of **options** or **actions** $\mathcal{A}^i = \{A_1^i, \dots, A_M^i\}$.

  - Let's denote the joint action set by $\mathcal{A} = \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$.
  - An action profile $a = (a^1, \dots, a^N) \in \mathcal{A}$ is a joint action of all agents.

- For each agent $i \in \mathcal{I}$, a **welfare**, **reward** or **payoff** function $R^i : \mathcal{A} \to \mathbb{R}$.

  - $R^i(a)$ is the welfare agent i receives when all agents chose $a = (a^1, \dots, a^N) \in \mathcal{A}$

- The agent's **policy** or **strategy** $x^i : \mathcal{A}^i \to [0, 1]$.

  - $x^i(a^i)$ is the probability agent i chooses action a^i.
  - A strategy is called *pure* if it chooses actions deterministically. If it is not *pure*, it is called *mixed* instead.

7.2.1 Let's play

- You are given two choices: **abate** climate change or continue to **pollute** the atmosphere.
- You gain 100 Euros (of averated damages) for each person that chooses *abate*.
- If you choose *abate*, you must pay 250 Euros.

What would you choose?

Would you have chosen differently if the action had been labeled **red** and **blue**?

The agents or actors in this game are all participants in this questionnaire. The actions are *abate* and *pollute*. The payoff is a given in (hypothetical) money. The strategies are the deterministic choice of either *abate* or *pollute*.

7.2.2 Mathematical model

Let us model the reward functions of this normal-form game in general terms. We have N agents, the players of the game, each with two actions A or P, the choices they can make.

Each abating actor brings a **benefit** b (of averted damages) to all actors at an **individual cost** c.

From the perspective of a focal agent, let N_A be the number of all other actors abating. The rewards are then

- for a polluting actor: $R_P(\mathbf{a}) = N_A b$
- for an abating actor: $R_A(\mathbf{a}) = (N_A + 1)b - c$

We visualize these payoffs as a function of the number of other abating actors.

```python
def plot_payoffs(N, b, c, ax=None):
    Na_other = np.arange(0, N)
    bA = (Na_other+1)*b - c
    bP = Na_other*b

    _, a = plt.subplots() if ax is None else (None, ax)
    a.plot(Na_other, bA, '.-', label='Abate')
    a.plot(Na_other, bP, '.-', label='Pollute')
    a.legend(); a.set_xlabel('Number of other actors abating');
↪   a.set_ylabel('Payoff')
```

```python
plot_payoffs(N=100, b=1, c=50)
```
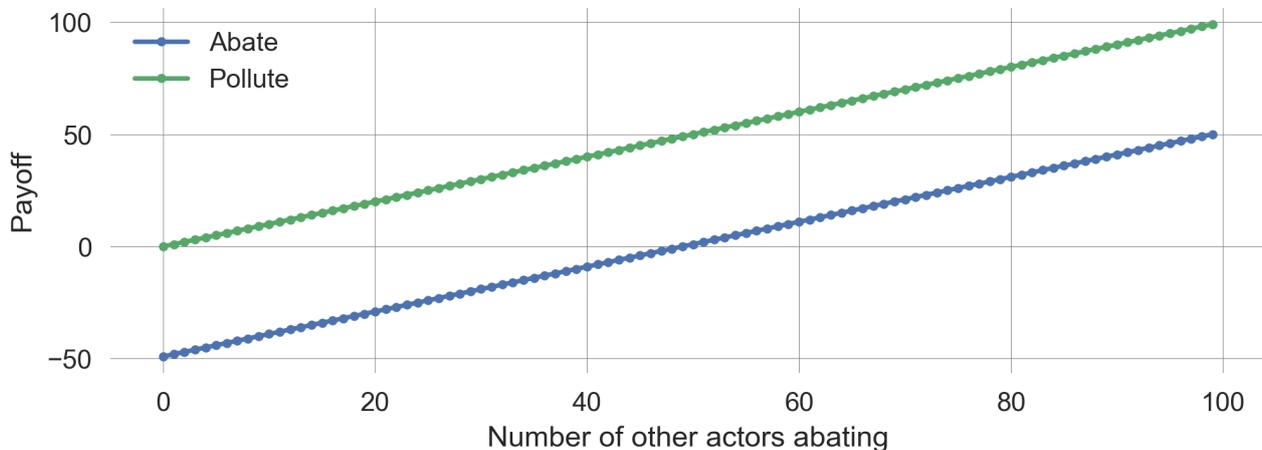

We observe that the reward of *polluting* is always higher than that of *abating*, regardless of the number of other actors abating.

Thus, regardless of what the others do, every individual is incentivized to choose *pollute*. Hence, for every individual i, *pollute* is a **dominant strategy**.

Definition | **Dominant strategy**

Let $x = (x^i, x^{-i})$ be a joint strategy, where x^i is actor i's strategy, and x^{-i} is the joint strategy of all other actors.

Actor i has a *dominant strategy*, x_D^i, iff

$$R^i(x_D^i, x^{-i}) \geq R^i(\tilde{x}^i, \tilde{x}^{-i})$$

for all possible other strategies $\tilde{x}^i, \tilde{x}^{-i}$.

Thus, when all actors choose *pollute*, no actor has an incentive to deviate from this strategy. This is called a **Nash equilibrium**.

7.2.3 Nash equilibrum

Definition

Let $x = (x^i, x^{-i})$ be a joint strategy, where x^i is agent i's strategy and x^{-i} is the joint strategy of all other agents.

A joint strategy x_* is a Nash-equilibrium when no agent can benefit from changing its strategy unilaterally,

$$R^i(x_*^i, x_*^{-i}) \geq R^i(\tilde{x}^i, x_*^{-i})$$

for all agents i and all other strategies $\tilde{X}^i$.

In 1950, John Nash showed (in a one-pager) that such an equilibrium always exists for games with *any number of (finite) actors with a finite number of actions* and *any type of payoffs* (beyond zero-sum games).

DeeDive | Nash's equilibrium produces the same solution as von Neumann and Morgenstern's minimax in the two-player zero-sum game. But while von Neumann and Morgenstern had struggled to extend the minimax solution beyond two-player zero-sum games in their 600-page book, Nash's solution could be extended to any other case! I recommend the following blog post Time for Some Game Theory - by Lionel Page for an intuitive introduction to game theory.

Interpretation

There is much confusion about **how to interpret a Nash equilibrium**, considering the question of how actors would be able to play a Nash equilibrium in a one-shot interaction. We will briefly discuss two interpretations: the *rationalistic* and the *learning* interpretation.

In the **rationalistic interpretation**, rational players would mentally simulate the various ways the game could unfold and choose a Nash equilibrium. However, when faced with non-trivial strategic situations for the first time, people typically fail to play a Nash equilibrium of the game. When there is more than one equilibrium, this interpretation cannot say which one an actor would choose (This is also known as the **equilibrium selection problem**).

In the **learning interpretation**, actors learn to play a Nash equilibrium through experience from repeated interactions over time. Learning here is understood in the broadest sense possible. It could be through imitation, trial and error, or even (cultural or genetic) evolution (Hoffman & Yoeli, 2022). However, we will not explicitly model the learning process in this lecture and the whole part on *target equilibria*. This will be the main topic of the last part of this course.

Movie time

The movie *A Beautiful Mind* portrays John Nash, the inventor of the Nash equilibrium. The Bar Scene is the moment in the film where Nash experiences the revelation of his equilibrium concept.

Is the solution of the game Nash advocates for in the clip a Nash equilibrium?

7.2.4 Social dilemma

In fact, the bar scene from *A Beautiful Mind* is a good example of a social dilemma.

A social dilemma is a situation where all actors have an incentive to behave selfishly.

However, everyone would be better off if everyone would behave cooperatively.

The game of abating and polluting is also a social dilemma for some values of the parameters b and c.

```
plot_payoffs(N=3, b=0.55, c=1.5)
```
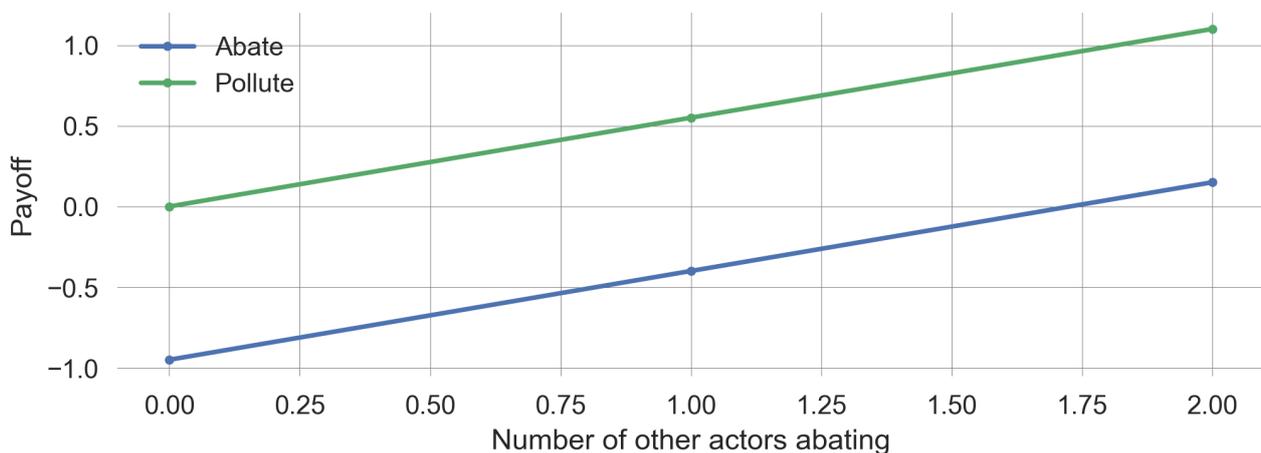

As long as the cost of abating is greater than the benefit, $b < c$, the unique Nash equilibrium is that all actors pollute.

However, when the benefit of abating times the number of actors is higher than the cost $c < bN$, all actors would be better off if all actors abate.

When both conditions are met, $b < c < bN$, the game is a social dilemma.

Note, when $c > bN$, all actors *polluting* is the unique Nash equilibrium and the social optimum. Everyone is better off when all actors pollute.

This simple model helps to explain why situations with many actors N can be more prone to be social dilemmas. When abating benefits everyone, independent of how many actors are involved, having many actors makes the situation likely to be a social dilemma. Independent of how large the cost c may be, we simply have to increase N, such that $c < bN$. The condition that the benefits b are independent of N refers to the **public good** nature of the benefits. The benefits of having an intact and healthy planet with maintenance and regulating ecosystem services serve everyone, regardless of how many. They are so-called non-rivalrous. Other services of Nature are rivalrous, e.g., the fish in the ocean.

7.3 Dimensions of a social dilemma

Let us dissect a social dilemma along two dimensions:

- the **greed** G to exploit others, and
- the **fear** F of being exploited by others.

For two actors that must face the decision between *abate* or *pollute*, we can summarize the payoffs in a matrix,

	Abate	Pollute
Abate	$1 \mid 1$	$-1 - F \mid +1 + G$
Pollute	$+1 + G \mid -1 - F$	$-1 \mid -1$

.

Depending on whether the greed G and fear F are positive or negative, we can distinguish four types of games Figure 10.10.
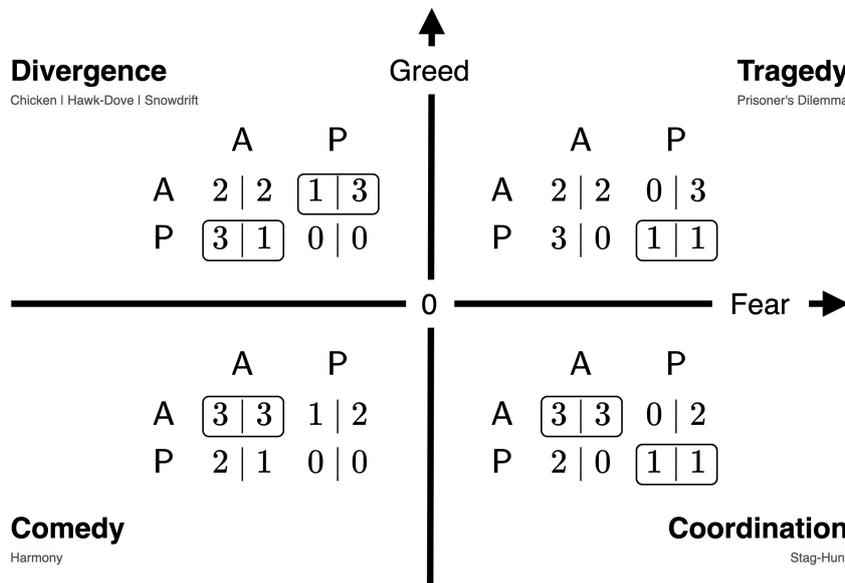
Figure 7.3: Dimensions of a social dilemma with ordinal payoffs and Nash equilibira shown in boxes.

In Figure 10.10, the payoff values are ordinal, meaning that only their order, $3 > 2 > 1 > 0$, is considered of relevance.

Case 1 | Tragedy ($G > 0, F > 0$).

When actors are greedy to exploit others and fear being exploited by others, the game is a tragedy. Regardless of what others do, each agent has an incentive to pollute. Thus, the **Nash equilibrium** is that **all actors pollute**. The tragedy is that all actors would be better off if all actors abate. Another common name for this situation is the *Prisoner's dilemma*.

Case 2 | Divergence ($G > 0, F < 0$).

When actors are greedy to exploit others but do not fear being exploited by others, the actors are in a situation of divergence. When enough other actors *pollute*, individual incentives regard *abate* better than *pollute*. Thus, in the two-actor case, both **(abate, pollute)** and **(pollute, abate)** are **Nash equilibria**, where the polluting actor receives more than the abating one (as long as $F > -(G + 2)$).

This is a situation of **inequality emerging** despite both actors being identical. It also induces a **first-mover advantage**, as the first actor to choose *pollute* will receive more reward than the *abating* actor. We call this situation **divergence** since the collective remains divided and only partial sustainability is achieved. Other popular names are *chicken*, *hawk-dove*, *snow-drift* describing different stories around the situation.

Case 3 | Coordination ($G < 0, F > 0$**).**

When actors are not greedy to exploit others but fear being exploited by others, they are in a situation of coordination. What is better for an individual mirrors what the others do. Thus, both **(abate, abate)** and **(pollute, pollute)** are **Nash equilibria**. In both equilibria, both agents are equally well off, but it depends on which of the two equilibria the actors coordinate. The agents are better off in *(abate, abate)* than in *(pollute, pollute)*. However, coordination may still be difficult to achieve, e.g., because of anonymity, a lack of communication, or false beliefs. Nevertheless, **turning a tragedy into a coordination game is a common mechanism to resolve the social dilemma**. Another popular name for this situation is the *stag-hunt* game.

Case 4 | Comedy ($G < 0, F < 0$**).**

When there is neither greed to exploit others nor fear to be exploited by others, the actors are in a situation of comedy. Regardless of what others do, each agent has an incentive to abate. Thus, the joint strategy **(abate, abate)** is the only **Nash equilibrium**. Since individual and collective interests point to the same solution, we call this the **comedy** of the commons (Ostrom et al., 2002). The *Harmony* game is another common name for this situation.

7.3.1 Limitations

We assumed that the **actors** were **anonymous**. However, actors are often not anonymous, especially in the governance of local commons. They know each other and can communicate and reciprocate (Anderies & Janssen, 2016; Nowak, 2006; Ostrom et al., 2002). This can help overcome the social dilemma. We will discuss and model this in the last part of the course.

We also did not discuss any mechanisms that let one or more of these games (or incentive regimes) emerge. We will discuss two such broad mechanisms in the remainder of this chapter: **international agreements** and **threshold public goods**.

7.4 International Agreements

Let us revisit our climate commons dilemma from above. Each abating actor brings a **benefit** $b = 100$ EUR (of averted damages) to all actors at an **individual cost** $c = 250$ EUR. One way to resolve this social dilemma could be through an agreement. The **actors could agree to abate**. In the following, we will model this additional game layer, highlighting its potential and limitations.

7.4.1 Let's play

We assume that **an agreement has already been negotiated**. If you sign the agreement, you must choose *abate*. However, the agreement comes only into force if there are at least three signatories.

You gain 100 Euros for each person who chooses *abate*. If you choose *abate*, you have to pay 250 Euros

You have to make three choices:

1) whether you **sign** or **not sign** the agreement
2) what you choose if there are not enough signatories: **abate** or **pollute**
3) what you choose if you did not sign the agreement: **abate** or **pollute**

7.4.2 Agreement participation game

Generally, in this model, the agreement mandates that all signatories *abate* if at least k^* actors sign the agreement. Then, at

- Stage 1: Every actor chooses whether or not to sign the agreement. At
- Stage 2: The signatories choose jointly whether to abate or pollute. Finally, at
- Stage 3: The non-signatories choose independently whether to abate or pollute.

7.4.3 Self-enforcing agreements

An international environmental agreement (IEA) must be **self-enforcing**, i.e., a Nash equilibrium in the game-theoretic sense, as there are no global enforcement mechanisms.

- No signatory can gain by withdrawing unilaterally
- No non-signatory can gain by joining
- Thus, there is **no incentive to re-negotiate**

7.4.4 Critical participation level k^*

The crucial question is, what is the critical participation level k^*, such that the agreement is self-enforcing?

Suppose there are k signatories.

- In a tragedy dilemma, non-signatories will defect in Stage 3.

- Signatories have an incentive to abate if $kb - c \geq 0$.

- Rearranging yields: if $k \geq c/b$, signatories abate.

- Let k^0 be the smallest integer greater than or equal to c/b.

- Suppose there are k^0 signatories.

- No non-signatory would want to join the agreement.

- Thus, the critical participation level is

$$\frac{c}{b} + 1 \geq k^* \geq \frac{c}{b}$$

```python
def plot_agreement_payoffs(N, b, c, ax=None):
    kstar = int(np.ceil(c/b)); print(kstar)

    Ns_other = np.arange(0, N)
    bS = ((Ns_other+1)*b - c < 0)*0 + ((Ns_other+1)*b - c >= 0)*((np.arange(N)+1)*b
↪    - c)
    bN = ((Ns_other)*b - c < 0)*0 + ((Ns_other)*b - c >= 0)*np.arange(N)*b

    _, a = plt.subplots() if ax is None else (None, ax)
    a.plot(Ns_other, bS, '.-', label='Signatories')
    a.plot(Ns_other, bN, '.-', label='Non-signatories')
    a.legend(); a.set_xlabel('Number of other signatories'); a.set_ylabel('Payoff')
```

7.4.5 Agreements turn tragedy into divergence

Full participation in an agreement is difficult (only possible if costs are astronomical). This holds for many generalizations (e.g., non-linear payoff functions/cost functions).

```python
fig, axs = plt.subplots(1,2, figsize=(11,3))
plot_agreement_payoffs(8, 1, 3.5, ax=axs[0]); axs[0].set_ylim(-3,8)
plot_payoffs(8, 1, 3.5, ax=axs[1]); axs[1].set_ylim(-3,8);
```

4
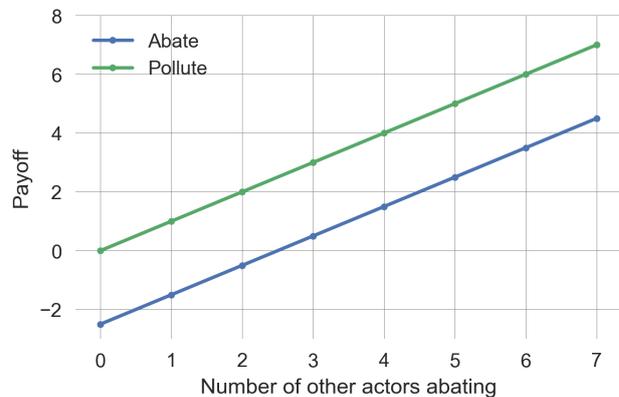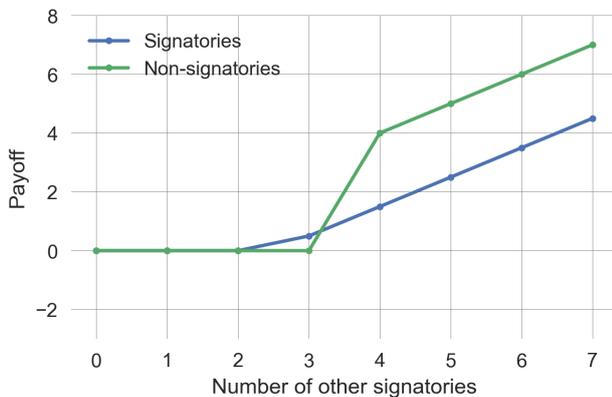
7.5 Threshold Public Goods

Is the tragedy dilemma the game we are playing?

Exceeding 1.5°C global warming could trigger multiple climate tipping points
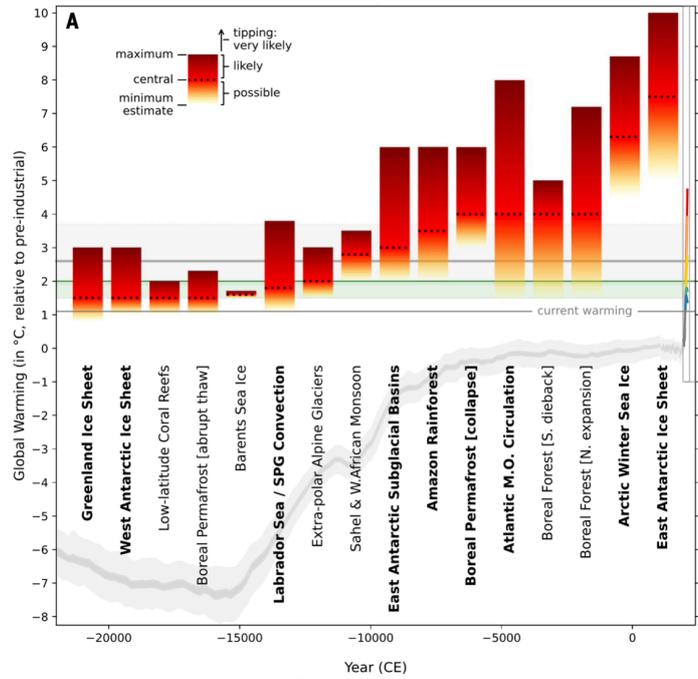
Armstrong-McKay (2022) *Exceeding 1.5°C global warming could trigger multiple climate tipping points*

Figure 7.4: Climate tipping risks

Let's play again

- You are given two choices: **abate** climate change or continue to **pollute** the atmosphere.
- You gain 100 Euros (of averated damages) for each person that chooses *abate*.
- If you choose *abate*, you must pay 250 Euros.
- If not all choose to *abate*, the climate will tip, and everybody will lose 350 Euros.

What would you choose?

7.5.1 Threshold dilemma game

In general, we can model a threshold public goods game as follows:

- There are N-actors.
- Each actor can contribute an amount c (by *abating*) to the public good, or they contribute nothing and *pollute*.
- Each contributed unit brings a benefit b_u (of averted linear damages) to all actors.
- If the collective does not contribute at least a critical threshold amount T_{crit}, all actors experience a *catastrophic* impact m of non-linear tipping damages.

How does the threshold dilemma map onto the tragedy dilemma? For simplicity, let's assume actors have two actions:

- Contributing a fair amount to avert the collapse, $c = T_{\text{crit}}/N$
- Contributing nothing.

The unit benefit b_u relates to a benefit b from abating from the tragedy dilemma by

$$b = b_u c = b_u T_{\text{crit}}/N$$

```
def plot_threshold_payoff(N, bu, Tc, m, ax=None):
    c=Tc/N; b=bu*c

    Na_other = np.arange(0, N)
    bA = (Na_other+1)*b - c - m*(Na_other+1<N)
    bP = Na_other*b - m*(Na_other<N)

    _, a = plt.subplots() if ax is None else (None, ax)
    a.plot(Na_other, bA, '.-', label='Abate')
    a.plot(Na_other, bP, '.-', label='Pollute')
    a.legend(); a.set_xlabel('Number of other actors abating');
↪   a.set_ylabel('Payoff')
```

Contribution thresholds turn tragedy into coordination challenge (Figure 7.5).

```
plot_threshold_payoff(5, 0.5, 50, 6);
```
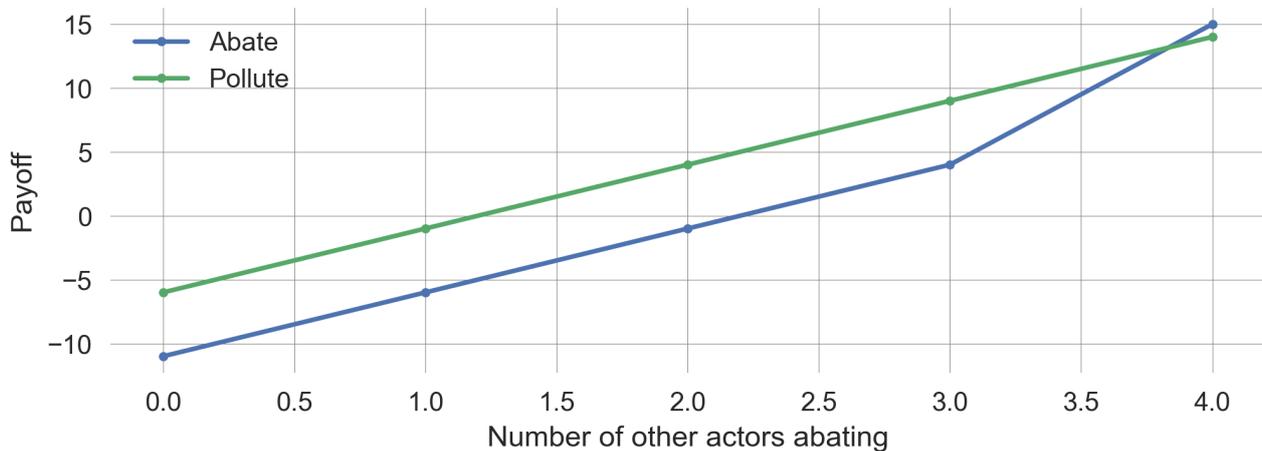

Figure 7.5: Payoffs from the thresholds game.

Nations are good at solving coordination challenges via international conferences and agreements. The **Montreal Protocol** on Substances that Deplete the Ozone Layer is the landmark multilateral environmental agreement that regulates the production and consumption of nearly 100 artificial chemicals referred to as ozone-depleting substances (ODS). When released into the atmosphere, those chemicals damage the stratospheric ozone layer, Earth's protective shield that protects humans and the environment from harmful levels of ultraviolet radiation from the sun. Adopted on 16 September 1987, the Protocol is, to date, one of the rare treaties to achieve universal ratification. [Source | UNEP]

Conditions for coordination

Under what conditions on the parameters does a threshold turn a tragedy dilemma into a coordination challenge?

The benefit of a cooperating actor choosing to abate when all others choose to abate is

$$R^i(\mathsf{A}, \mathsf{A}) = Nb_u\frac{T_c}{N} - \frac{T_c}{N}$$

```
def bAA(N, bu, Tc, m): return N*bu*Tc/N - Tc/N
```

The benefit of a *polluter*, when all others choose *abate* is

$$R^i(\mathsf{P}, \mathsf{A}) = (N-1)b_u\frac{T_c}{N} - m$$

```
def bPA(N, bu, Tc, m): return (N-1)*bu*Tc/N - m
```

When all other actors *pollute*, there is no incentive to abate (because the collapse happens anyway). But when all other actors *abate*, there is an incentive to *abate* if

$$R^i(\mathsf{A}, \mathsf{A}) > R^i(\mathsf{P}, \mathsf{A}).$$

Rearranging yields,

$$m^* = \frac{1-b_u}{N}T_c^*.$$

```
# Parameters
bu=0.5; N = 5

# Varying parameters
ms = np.linspace(0, 10, 201);
Ts = np.linspace(0, 100, 251);
TT, MM = np.meshgrid(Ts, ms)

# Get the plot nice
cmap = plt.colormaps['PiYG'];
norm = BoundaryNorm([0,0.5,1], ncolors=cmap.N, clip=True)

# do the plot
cb = plt.pcolormesh(TT, MM, bAA(N, bu, TT, MM)>bPA(N, bu, TT, MM), cmap=cmap,
↪   norm=norm)

# make the plot nice (again)
cbar = plt.colorbar(cb, ticks=[0.25, 0.75]);
cbar.ax.set_yticklabels(['Tragedy', 'Coordination'])

plt.xlabel('Threshold'); plt.ylabel('Impact');
plt.plot(Ts, Ts/N*(1-bu));
```
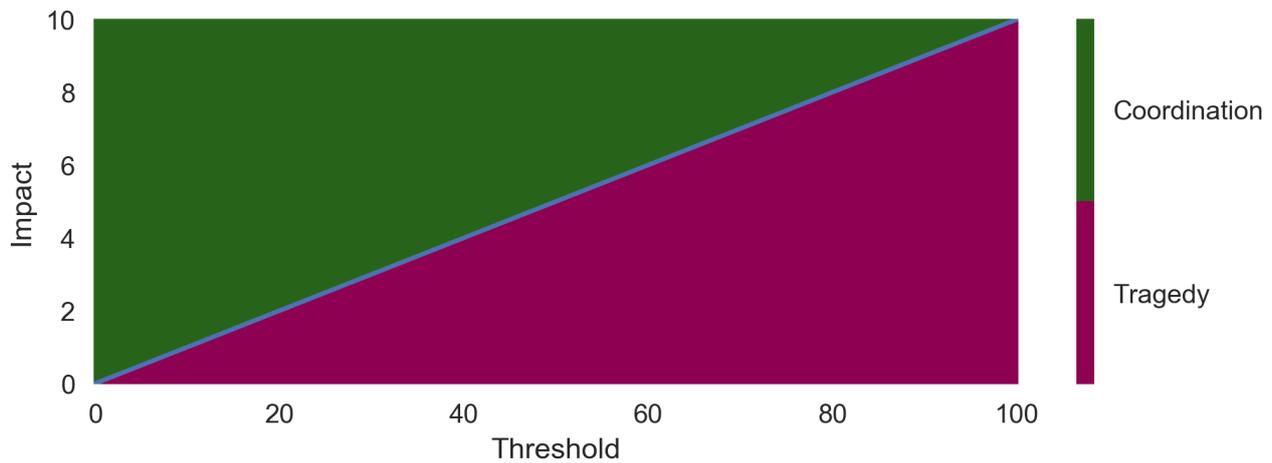
7.5.2 Uncertainty

What if there is uncertainty? We will consider two kinds of uncertainty:

- Impact uncertainty: We don't know exactly how bad it is going to be
- Threshold uncertainty: We don't know exactly where the threshold lies

Impact uncertainty

Assume the impact m is uncertain.

The impact m is distributed according to a probability distribution.

Expected value theory underlying game theory suggests that decision-makers will only care about the expected value.

Thus, as long as $m_{\text{certain}} = \mathbb{E}[m_{\text{uncertain}}]$, the model remains unchanged.

Here, we consider a very simple distribution of $Pr(m)$. There is a $1/3$ chance of a small impact $m_{\text{certain}} - \Delta m$, a $1/3$ chance of a medium impact $m = m_{\text{certain}}$, and a $1/3$ chance of a large impact $m = m_{\text{certain}} + \Delta m$. Obviously, the expected value is m_{certain}.

```python
def plot_uncertainimpacts_payoffs(N, bu, Tc, m, delta_m=2, ax=None):
    c=Tc/N; b=bu*c;

    Na_other_fair = np.arange(0, N)

    bPl = Na_other_fair*b - (m-delta_m)*(Na_other_fair<N)
    bPc = Na_other_fair*b - m*(Na_other_fair<N)
    bPh = Na_other_fair*b - (m+delta_m)*(Na_other_fair<N)

    bAl = (Na_other_fair+1)*b - c - (m-2)*(Na_other_fair+1<N)
    bAc = (Na_other_fair+1)*b - c - m*(Na_other_fair+1<N)
    bAh = (Na_other_fair+1)*b - c - (m+2)*(Na_other_fair+1<N)

    bP = np.mean([bPl, bPc, bPh], axis=0)
    bA = np.mean([bAl, bAc, bAh], axis=0)

    _, a = plt.subplots() if ax is None else (None, ax)
```

```
    col1 = plt.rcParams['axes.prop_cycle'].by_key()['color'][0]
    col2 = plt.rcParams['axes.prop_cycle'].by_key()['color'][1]

    a.plot(Na_other_fair, bAc, '.-', color=col1, alpha=0.5)
    a.plot(Na_other_fair, bAl, '.--', color=col1, alpha=0.5)
    a.plot(Na_other_fair, bAh, '.-.', color=col1, alpha=0.5)

    a.plot(Na_other_fair, bPc, '.-', color=col2, alpha=0.5)
    a.plot(Na_other_fair, bPl, '.--', color=col2, alpha=0.5)
    a.plot(Na_other_fair, bPh, '.-.', color=col2, alpha=0.5)

    a.plot(Na_other_fair, bA, '.-', label='Average abate', color=col1)
    a.plot(Na_other_fair, bP, '.-', label='Average pollute', color=col2)

    a.set_xlabel('Number of other actors abating'); a.set_ylabel('Payoffs')
    a.legend()
```

```
plot_uncertainimpacts_payoffs(N=5, bu=0.5, Tc=50, m=6, delta_m=4)
```
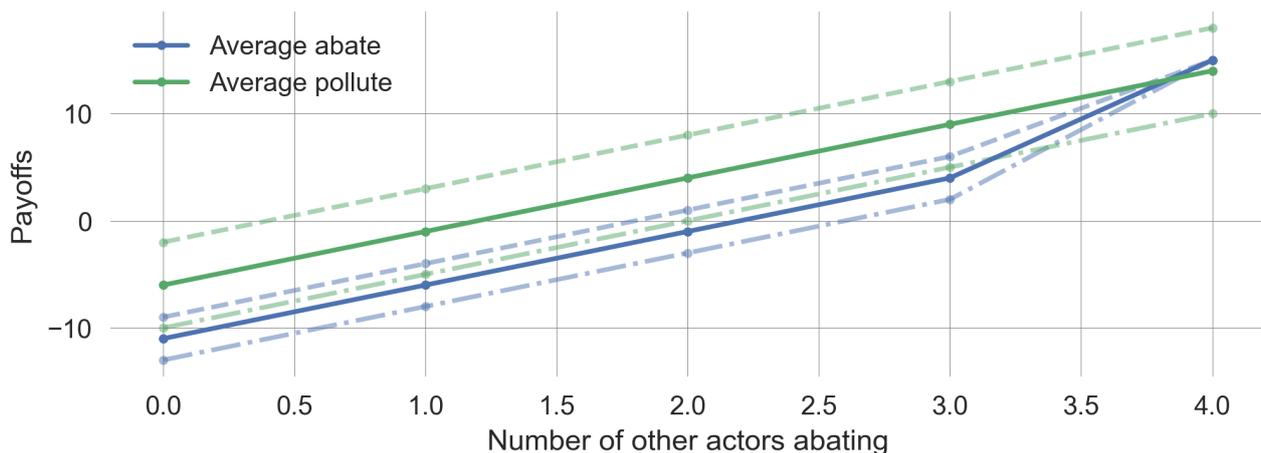

Thus, impact uncertainty does not change the outcome of our model.

Threshold uncertainty

Empirically, there is considerable uncertainty about the critical threshold T_{crit} (Figure 7.4).

Suppose, the threshold T_{crit} is uncertain:

- with $p = 1/3$ it is at $T_{\text{crit,low}} = T_{\text{crit,certain}} - 10$
- with $p = 1/3$ it is at $T_{\text{crit,mid}} = T_{\text{crit,certain}}$
- with $p = 1/3$ it is at $T_{\text{crit,high}} = T_{\text{crit,certain}} + 10$

Thus, expected value of the threshold remains the same.

```
def plot_uncertainthresholds_payoff(N, bu, Tc, m, ax=None):
    c=Tc/N; b=bu*c
```

```
    Na_other_fair = np.arange(0, N)

    bD = Na_other_fair*b - m*(Na_other_fair<N)

    bC = (Na_other_fair+1)*b - c - m*(Na_other_fair+1<N)
    bL = (Na_other_fair+1)*b - c - m*(Na_other_fair+1<N)
    bH = (Na_other_fair+1)*b - c - m*(Na_other_fair<N)

    bA = np.mean([bC, bL, bH], axis=0)

    _, a = plt.subplots() if ax is None else (None, ax)
    a.plot(Na_other_fair, bA, '.-', label='Average abate')
    a.plot(Na_other_fair, bD, '.-', label='Pollute')
    a.plot(Na_other_fair, bC, '.-', label='Abate - Center threshold', color='grey')
    a.plot(Na_other_fair, bL, '.--', label='Abate - Low threshold', color='black')
    a.plot(Na_other_fair, bH, '.-.', label='Abate - High threshold',
↪   color='lightgrey')

    a.legend(); a.set_xlabel('Number of other actors abating');
↪   a.set_ylabel('Payoff')
```

Threshold uncertainty reverts the coordination challenge back to a tragedy

```
plot_uncertainthresholds_payoff(N=5, bu=0.5, Tc=50, m=6)
```
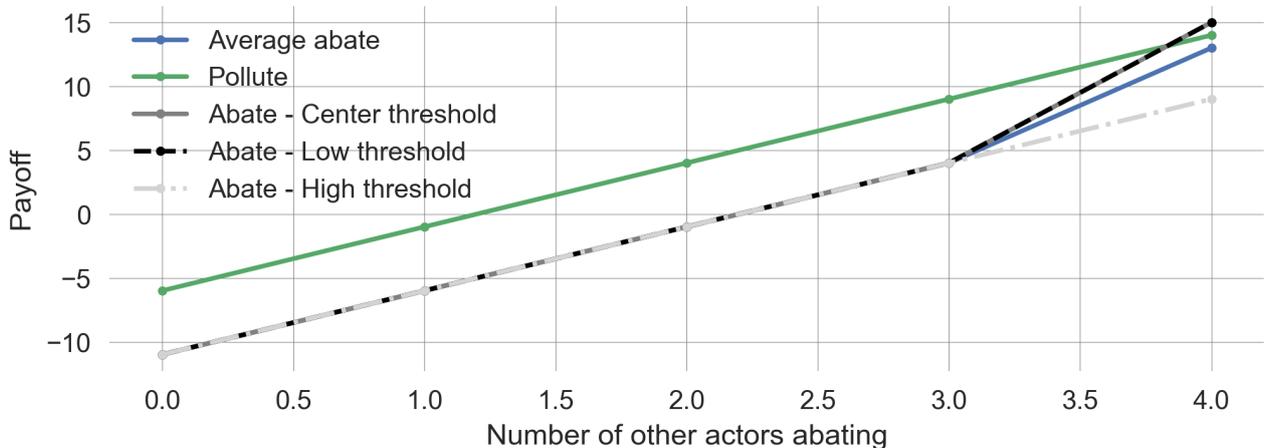

Threshold uncertainty raises the cost of averting collapse. Essentially, this is **because low thresholds cannot compensate for high thresholds**. High thresholds cause collapse, making actors worse off than those with medium thresholds. However, low thresholds cannot make actors better off than medium thresholds.

Threshold dilemmas summary

Contribution thresholds can turn tragedy into a coordination challenge - given a sufficiently severe collapse impact and sufficiently low threshold.

Uncertainty is important: Impact uncertainty has no effect - But threshold uncertainty can revert the coordination challenge back to tragedy.

Experimental evidence

Our model allows us to make **predictions** about how actors behave in threshold public goods games across the four (two by two) treatments of uncertainty:

1) **Impact uncertainty has no effect**. Certainty and Impact uncertainty should yield the same behavior (abating), just as Threshold uncertainty and Threshold+Impact uncertainty should yield the same behavior (polluting).
2) **Threshold uncertainty has an effect**. The certainty and impact uncertainty treatment should yield a different behavior than the treatments with threshold uncertainty.

Both hypotheses are confirmed by the experimental evidence (Figure 7.6) (Barrett & Dannenberg, 2012).

Barrett & Dannenberg (2012) *Climate negotiations under scientific uncertainty*
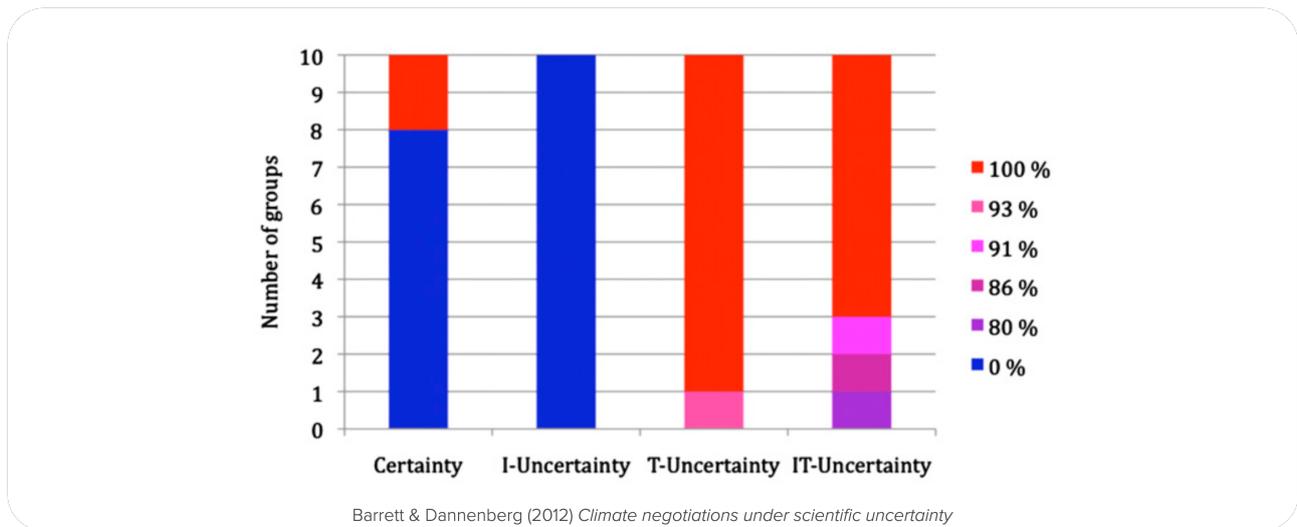
Figure 7.6: Experimental evidence on threshold public goods games

7.6 Learning goals revisited

In this chapter,

we applied game theory to model multi-agent action situations.

We resolved games by defining and finding Nash equilibria.

We described and analyzed the dimensions of a social dilemma.

We introduced and analyzed two special kinds of games: *agreement games* and *threshold public goods.*

- Agreement games can turn tragedies into divergence dilemmas.
- Threshold public goods can turn tragedies into coordination challenges - given enough certainty about where the threshold lies.

7.6.1 Overall limitations

Our equilibrium game-theoretic model left it **unclear where the strategies come from** or from which process they arise.

The models in this chapter did **not explicitly consider environmental dynamics**.

The **consequences** for the actors were assumed to be **experienced immediately**.

7.6.2 Bibliographic remarks

The fear and greed dimensions of a social dilemma are inspired by (Macy & Flache, 2002).

The name of the four social dilemma types is inspired by (Ostrom et al., 2002), who talks about the *drama of the commons*. Commons may sometimes be a *tragedy*, sometimes a *comedy*, often something in between.

International environmental agreement games are coined by (Barrett, 1994, 2005).

8 Dynamic Interactions

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

8.1 Motivation | Futures and environments

Prototypical models did not explicitly consider future environmental consequences of strategic interactions (Figure 8.1).
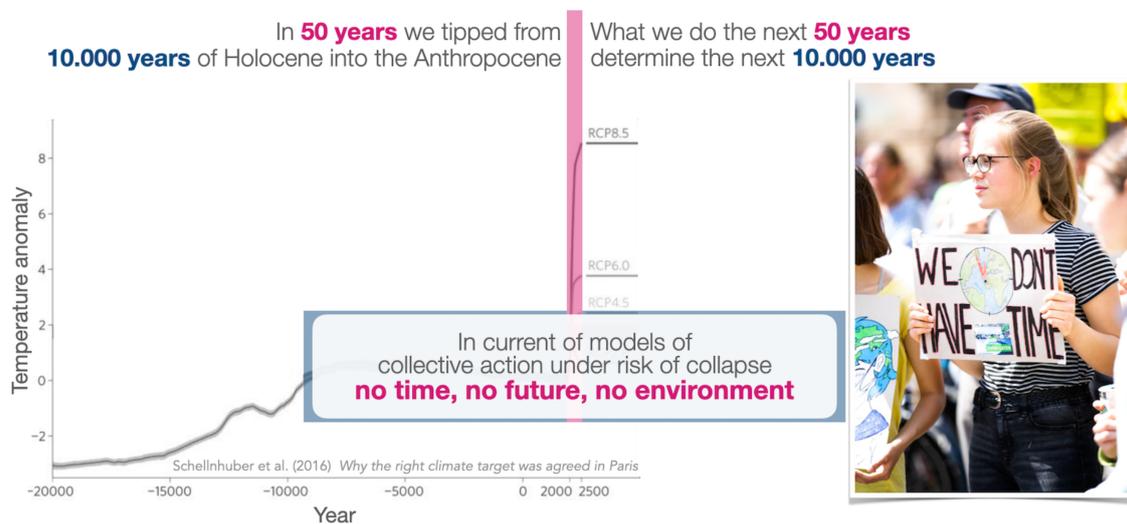
Figure 8.1: Environmental dynamics are relevant to consider

However, many real-world scenarios involve strategic interactions with environmental consequences. For example, the tragedy of the commons, agreements, and threshold public goods can be extended to include ecological consequences. In this lecture, we will introduce **dynamic games**, particularly *stochastic* or *Markov games*, to model strategic interactions with environmental consequences.

Stochastic games integrate *Markov chains*, *Markov decision processes*, and *game theory* to model strategic interactions in dynamic environments. They are particularly useful for modeling human-environment interactions, where the environment is affected by human actions and, in turn, affects human behavior.

8.1.1 Advantages of dynamic games

Using **dynamic games**, particularly *stochastic games* to model strategic interactions with environmental consequences has several advantages:

- **inherently stochastic** - to account for uncertainty

- **nonlinear** - to account for structural changes
- **agency** - to account for human behavior
- **interactions** - to account for strategic interactions
- **future-looking** - to account for the trade-off between short-term and long-term
- **feedback** - between multiple agents and the environment

Stochastic games also have structural benefits that make them compatible with numerical computer modeling due to their discrete action and state sets, as well as their advancement in discrete time intervals.

8.1.2 Learning goals

After this lecture, students will be able to:

- Describe the elements of a stochastic game
- Apply stochastic games to model human-environment interactions
- Analyze the strategic interactions in stochastic games

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

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;
```

8.2 Dynamic games | Strategic interactions with environmental consequences
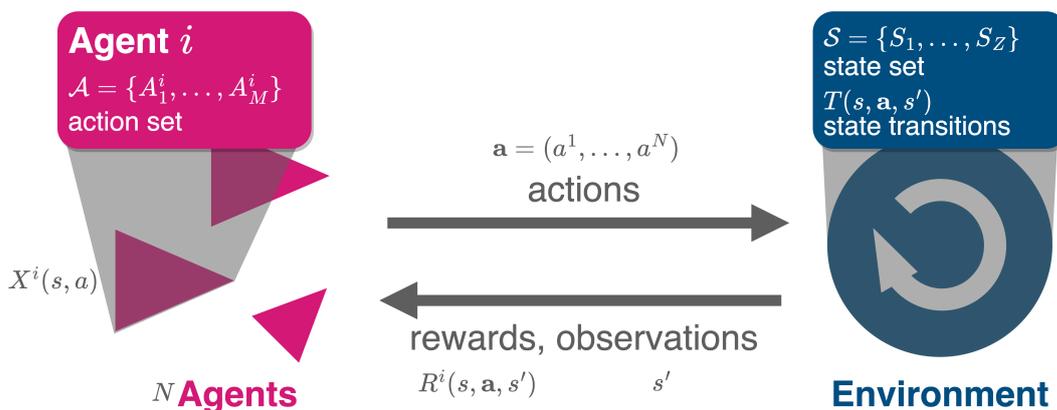
Figure 8.2: Multiagent-Environment Interface

Here, we focus on environments with a discrete state set in discrete time. These specifications are commonly called **stochastic** or **Markov games**. They consist of the following elements:

- A discrete set of environmental **contexts** or **states** $\mathcal{S} = \{S_1, \ldots, S_Z\}$.

  - We denote an environmental state by $s \in \mathcal{S}$.

- A finate set of N agents $\mathcal{J} = \{2, \ldots, N\}$ participating in an interaction.
- For each agent $i \in \mathcal{J}$, a discrete set of **options** or **actions** $\mathcal{A}^i = \{A_1^i, \ldots, A_M^i\}$.

  - Let's denote the joint action set by $\mathcal{A} = \mathcal{A}^1 \times \cdots \times A^N$.
  - An action profile $\mathbf{a} = (a^1, \ldots, a^N) \in \mathcal{A}$ is a joint action of all agents.

Time t advances in discrete steps $t = 0, 1, 2, \ldots$, and agents choose their actions simultaneously.

- We denote the state at time t by s_t and the joint action by $\mathbf{a}_t$.

The transitions tensor $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ defines the **environmental dynamics**.

- $T(s, \mathbf{a}, s')$ is the probability of transitioning from state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$ given the joint action $\mathbf{a}$.
- Thus, $\sum_{s'} T(s, \mathbf{a}, s') = 1$ most hold for all $s \in \mathcal{S}$ and $\mathbf{a} \in \mathcal{A}$.

The reward tensor $\mathbf{R} : \mathcal{J} \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ defines the agents' **short-term welfare**, **utility**, **rewards**, or **payoffs**.

- $R^i(s, \mathbf{a}, s')$ is the reward agent i receives for transitioning from state s to s' given the joint action $\mathbf{a}$.
- We assume that it is each agent i's goal to maximize their expected discounted sum of future rewards, $G^i = \sum_{t=0}^{\infty} (\gamma^i)^t R^i(s_t, \mathbf{a}_t, s_{t+1})$, where $\gamma^i \in [0,1)$ is the discount factor.

We assume that agents can condition their probabilities of choosing actions on the current state s_t, yielding so-called Markov **policies** or **strategies**, $\mathbf{x} : \mathcal{J} \times \mathcal{S} \times \mathcal{A}^i \to [0,1]$.

- $x^i(s, a)$ is the probability agent i chooses action a in state s.

8.3 Application | Ecological public good

We apply the stochastic game framework to integrate the fact that we are embedded in a shared environment and care about the future to some extent. We do so by considering a public good game with ecological consequences (Figure 8.3), which allows us to answer how the strategic incentives depend on their level of care for future rewards.

8.3.1 States, agents and actions

The environment consists of two states, $\mathcal{S} = \{\mathsf{p}, \mathsf{d}\}$, representing a **prosperous** and a **degraded** state of the environment.

```
state_set = ['prosperous,' 'degraded']; p=0; g=1; Z=2
```

We also defined two Python variable `p=0` and `g=1` to serves as readable and memorable indices to represent the environmental contexts.

There are 2 identical agents. In each state $s \in \mathcal{S}$, each agent $i \in \{1, 2\}$ can choose within their action set between either **cooperation** or **defection**, $\mathcal{A}^i = \{\mathsf{c}, \mathsf{d}\}$.
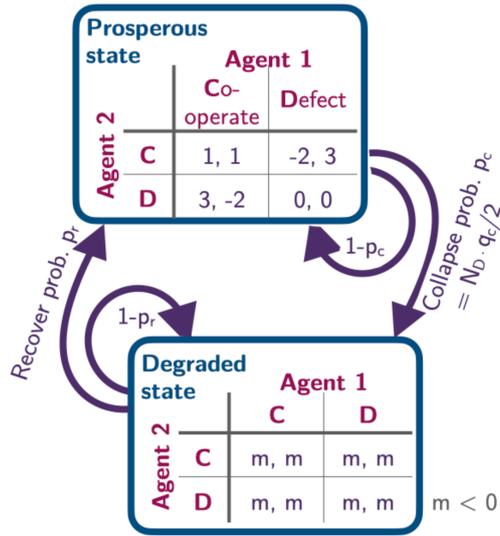
Figure 8.3: Ecological public good collective decision-making environment

```
action_sets = ['cooperate', 'defect']; c=0; d=1; M=2
```

Likewise, we define two Python variables, `c=0` and `d=1`, to serve as readable and memorable indices to represent the agents' actions.

We denote the number of cooperating agents by N_c. The number of defecting agents is $N_d = N - N_c$.

8.3.2 Transitions | Environmental dynamics

We represent the **environmental dynamics**, i.e., the transitions between environmental state contexts, in a $Z \times M \times M \times Z$ tensor, where Z is the number of states and M is the number of actions. In this representation,

- the first dimension corresponds to the **current state**,
- the second to the action profile of the **first agent**,
- the third to the action profile of the **other agent**, and
- the fourth and last dimension corresponds to the **next state**.

```
TransitionTensor = np.zeros((Z, M, M, Z), dtype=object)
```

The *environmental dynamics* are then governed by two parameters: a collapse leverage, q_c, and a recovery probability, p_r.

```
qc, pr = sp.symbols('q_c p_r')
```

A collapse from the prosperous to the degraded state occurs with a transition probability

$$T(\mathbf{p}, \mathbf{a}, \mathbf{g}) = \frac{N_d}{N} q_c.$$

```
TransitionTensor[p, c, c, g] = 0   # no agent defects
TransitionTensor[p, d, c, g] = qc/2  # one agent defects
TransitionTensor[p, c, d, g] = qc/2  # other agent defects
TransitionTensor[p, d, d, g] = qc   # all agents defect
```

Thus, if all agents defect, the environment collapses with probability q_c. The collapse leverage indicates how much impact a defecting agent exerts on the environment. The environment remains within the prosperous state with probability, $T(\mathsf{p}, \mathbf{a}, \mathsf{p}) = 1 - \frac{N_d}{N} q_c$.

```
TransitionTensor[p, : , :, p] = 1 - TransitionTensor[p, : , :, g]
```

In the degraded state, we set the recovery to occur with probability,

$$T(\mathsf{g}, \mathbf{a}, \mathsf{p}) = p_r,$$

independent of the agents' actions.

```
TransitionTensor[g, :, :, p] = pr
```

The probability that the environment remains in the degraded state is then, $T(\mathsf{g}, \mathbf{a}, \mathsf{g}) = 1 - p_r$.

```
TransitionTensor[g, :, :, g] = 1-pr
```

Together, our transition tensor is then given by

```
sp.Array(TransitionTensor)
```

$$
\left[
\begin{bmatrix}
\begin{bmatrix} 1 & 0 \\ 1 - \frac{q_c}{2} & \frac{q_c}{2} \end{bmatrix} & \begin{bmatrix} 1 - \frac{q_c}{2} & \frac{q_c}{2} \\ 1 - q_c & q_c \end{bmatrix} \\
\begin{bmatrix} p_r & 1 - p_r \\ p_r & 1 - p_r \end{bmatrix} & \begin{bmatrix} p_r & 1 - p_r \\ p_r & 1 - p_r \end{bmatrix}
\end{bmatrix}
\right]
$$

Last, we make sure that our transition tensor is normalized, i.e., the sum of all transition probabilities from a state-joint-action pair to all possible next states equals one, $\sum_{s'} T(s, \mathbf{a}, s') = 1$.

```
TransitionTensor.sum(-1)
```

```
array([[[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]]], dtype=object)
```

8.3.3 Rewards | Short-term welfare
```

```
RewardTensor = np.zeros((2, Z, M, M, Z), dtype=object)
```

Rewards in the prosperous state follow the standard public goods game with a synergy factor r and a cost of cooperation c.

```
r, cost = sp.symbols('r c')
```

The rewards for cooperating and defecting agents are then given by

$$R^i(\mathsf{p}, a^i, a^{-i}, \mathsf{p}) = \begin{cases} \frac{rc(N_c+1)}{N} - c, & \text{if } a^i = \mathsf{c}, \\ \frac{rcN_c}{N}, & \text{if } a^i = \mathsf{d}. \end{cases}$$

```
RewardTensor[:, p, c, c, p] = r*cost - cost
RewardTensor[:, p, d, d, p] = 0
RewardTensor[0, p, c, d, p] = RewardTensor[1, p, d, c, p] = r*cost/2 - cost
RewardTensor[0, p, d, c, p] = RewardTensor[1, p, c, d, p] = r*cost/2
```

When a state transition involves the degraded state, g, the agents only receive an environmental collapse impact m:

```
m = sp.symbols('m')
```

$$R^i(\mathsf{p}, \mathbf{a}, \mathsf{g}) = R^i(\mathsf{g}, \mathbf{a}, \mathsf{g}) = R^i(\mathsf{g}, \mathbf{a}, \mathsf{p}) = m, \quad \text{for all } \mathbf{a}, i.$$

```
RewardTensor[:, p, :, :, g] = RewardTensor[:, g, :, :, g] = RewardTensor[:, g, :, :,
↪  p] = m
```

Together, our reward tensor is then given by

```
sp.Array(RewardTensor)
```

$$\left[\left[\left[\begin{bmatrix} cr - c & m \\ \frac{cr}{2} - c & m \end{bmatrix} \begin{bmatrix} \frac{cr}{2} & m \\ 0 & m \end{bmatrix}\right] \left[\begin{bmatrix} m & m \\ m & m \end{bmatrix} \begin{bmatrix} m & m \\ m & m \end{bmatrix}\right]\right] \left[\left[\begin{bmatrix} cr - c & m \\ \frac{cr}{2} & m \end{bmatrix} \begin{bmatrix} \frac{cr}{2} - c & m \\ 0 & m \end{bmatrix}\right] \left[\begin{bmatrix} m & m \\ m & m \end{bmatrix} \begin{bmatrix} m & m \\ m & m \end{bmatrix}\right]\right]\right]$$

8.3.4 DeepDive | Subsituting parameter values

In this chapter, we defined the transition and reward tensors as general `numpy` arrays with data types `object`, which we filled with symbolic expressions from `sympy`. To manipulate and substitute these expressions, we can use the `sympy.subs` method, however, not directly on the `numpy` array. Instead, we define a helper function `substitute_in_array` that takes a `numpy` array and a dictionary of substitutions and returns a new array with the substitutions applied.

```
def substitute_in_array(array, subs_dict):
    result = array.copy()
    for index,_ in np.ndenumerate(array):
        if isinstance(array[index], sp.Basic):
            result[index] = array[index].subs(subs_dict)
    return result
```

To make this work, it seems to be of critical importance that the subsitution dictionary is given as a dictionary in the form of `{<symbol_variable>: <subsitution>, ...}` and *not* as `dict(<symbol_variable>=<subsitution>, ...)`. For example,

```
substitute_in_array(TransitionTensor, {pr:0.01, qc:0.2}).astype(float)
```

```
array([[[[1.  , 0.  ],
         [0.9 , 0.1 ]],

        [[0.9 , 0.1 ],
         [0.8 , 0.2 ]]],


       [[[0.01, 0.99],
         [0.01, 0.99]],

        [[0.01, 0.99],
         [0.01, 0.99]]]])
```

8.3.5 Policies | Strategic choices

The **crucial question** is **whether** or not the agents should **cooperate** or defect in the **prosperous state**, p, under the assumption that agents are **anonymous** - and how to answer depends on the parameter conditions q_c, p_r, γ, r, c, and m.

Anonymity means that agents do not consider the game's history, i.e., behave according to a Markov policy. We analyze the two extreme cases: An agent can either cooperate or defect in the prosperous state, p.

Generally, a single agent's policy is represented by a $Z \times M$ tensor. The cooperative policy is then given by

```
Xsa_coo =  0.5 * np.ones((Z, M))
Xsa_coo[p, c] = 1
Xsa_coo[p, d] = 0
```

The defective policy is given by

```
Xsa_def =  0.5 * np.ones((Z, M))
Xsa_def[p, c] = 0
Xsa_def[p, d] = 1
```

To obtain the incentive regimes, we need to calculate the long-term values of the four policy combinations:

- mutual cooperation (cc), i.e. both agents cooperate,
- unilateral defection (dc), i.e. one agent defects, and the other cooperates,
- unilateral cooperation (cd), i.e. one agent cooperates, and the other defects, and
- mutual defection (dd), i.e. both agents defect.

They can also be represented by a *meta-game* payoff matrix, where the rows represent the focal agent's policies, and the columns represent the opponent's policies,

	c	d
c	v_{cc}	v_{cd}
d	v_{dc}	v_{dd}

We summarize these respecitve joint policies in four $N \times Z \times M$ tensors,

```python
Xisa_cc = np.array([Xsa_coo, Xsa_coo])
Xisa_cd = np.array([Xsa_coo, Xsa_def])
Xisa_dc = np.array([Xsa_def, Xsa_coo])
Xisa_dd = np.array([Xsa_def, Xsa_def])
```

8.3.6 State values | Long-term welbeing

Long-term values are defined precisely like in the single-agent case (03.01-SequentialDecisions) except that they now depend on the **joint policy x** and each agent i holds their own values.

Given a joint policy **x**, we define the **state value** for agent i, $v_{\mathbf{x}}^i(s)$, as the expected gain, $\mathbb{E}_{\mathbf{x}}[G_t^i | S_t = s]$, when starting in state s and the following the policy **x**,

$$v_{\mathbf{x}}^i(s) := \mathbb{E}_{\mathbf{x}}[G_t^i | S_t = s] = (1 - \gamma^i)\mathbb{E}_{\mathbf{x}}\left[\sum_{\tau=t}^{\infty} (\gamma^i)^{\tau} R_{t+\tau+1}^i | S_t = s\right], \quad \text{for all } s \in \mathcal{S},$$

They are also computable like in the single-agent case (03.01-SequentialDecisions) by solving the **Bellman equations** in matrix form,

$$\mathbf{v}_{\mathbf{x}}^i = (1 - \gamma^i)(\mathbf{1}_Z - \gamma^i \underline{\mathbf{T}}_{\mathbf{x}})^{-1}\mathbf{R}_{\mathbf{x}}^i.$$

Before we solve this equation, we focus on computing the effective **transition matrix $\underline{\mathbf{T}}_{\mathbf{x}}$** and the **average reward $\mathbf{R}_{\mathbf{x}}^i$**, given a joint policy **x**.

The **transition matrix $\underline{\mathbf{T}}_{\mathbf{x}}$** is a $Z \times Z$ matrix, where the element $T_{\mathbf{x}}(s, s')$ is the probability of transitioning from state s to s' under the joint policy **x**. It is computed as

$$T_{\mathbf{x}}(s, s') = \sum_{a^1 \in \mathcal{A}^1} \dots \sum_{a^N \in \mathcal{A}^N} x^1(s, a^1) \dots x^N(s, a^N) T(s, a^1, \dots, a^N, s').$$

For $N = 2$, we implement in Python as follows:

```python
def compute_symbolic_TransitionMatrix_Tss(policy_Xisa,
                                          transitions_Tsas):
    s, a, b, s_ = 0, 1, 2, 3  # defining indices for convenience

    Tss = sp.Matrix(np.einsum(policy_Xisa[0], [s, a],
                              policy_Xisa[1], [s, b],
                              transitions_Tsas, [s, a, b, s_],
                              [s,s_]))
    return sp.simplify(Tss)
```

For example, the transition matrix for the joint policy $\mathbf{x} = (\mathsf{d}, \mathsf{c})$ is then given by

```
compute_symbolic_TransitionMatrix_Tss(Xisa_dc, TransitionTensor)
```

$$\begin{bmatrix} 1.0 - 0.5q_c & 0.5q_c \\ 1.0p_r & 1.0 - 1.0p_r \end{bmatrix}$$

The **average reward** $\mathbf{R}_{\mathbf{x}}^i$ is a $N \times Z$-matrix, where the element $R_{\mathbf{x}}^i(s)$ is the expected reward agent i receives in state s under the joint policy $\mathbf{x}$. It is computed as

$$R_{\mathbf{x}}^i(s) = \sum_{a^1 \in \mathcal{A}^1} \cdots \sum_{a^N \in \mathcal{A}^N} x^1(s, a^1) \ldots x^N(s, a^N) T(s, a^1, \ldots, a^N, s') R^i(s, a^1, \ldots, a^N, s').$$

For $N = 2$, we implement in Python as follows:

```python
def compute_symbolic_AverageReward_Ris(policy_Xisa,
                                        transitions_Tsas,
                                        rewards_Risas):
    i, s, a, b, s_ = 0, 1, 2, 3, 4 # defining indices for convenience
    Ris = sp.Array(np.einsum(policy_Xisa[0], [s, a],
                             policy_Xisa[1], [s, b],
                             transitions_Tsas, [s, a, b, s_],
                             rewards_Risas, [i, s, a, b, s_],
                             [i, s]))
    return sp.simplify(Ris)
```

For example, the average reward under the joint policy $\mathbf{x} = (\mathsf{d}, \mathsf{c})$ is then given by

```
compute_symbolic_AverageReward_Ris(Xisa_dc, TransitionTensor, RewardTensor)
```

$$\begin{bmatrix} -\frac{cr(0.5q_c - 1.0)}{2} + 0.5mq_c & 1.0m \\ -\frac{c(0.5q_c - 1.0)(r-2)}{2} + 0.5mq_c & 1.0m \end{bmatrix}$$

With the transition matrix $\underline{\mathbf{T}}_{\mathbf{x}}$ and the average reward $\mathbf{R}_{\mathbf{x}}^i$, we can now solve the Bellman equation for the state values $\mathbf{v}_{\mathbf{x}}^i$. For convenience, we assume a homogeneous discount factor $\gamma^i = \gamma$ for all agents i.

```
dcf = sp.symbols('gamma')
def compute_symbolic_statevalues(policy_Xisa,
                                 transitions_Tsas,
                                 rewards_Risas,
                                 discountfactor=dcf):
    i, s, a, s_ = 0, 1, 2, 3  # defining indices for convenience

    Tss = compute_symbolic_TransitionMatrix_Tss(
        policy_Xisa, transitions_Tsas)
    Ris = compute_symbolic_AverageReward_Ris(
        policy_Xisa, transitions_Tsas, rewards_Risas)

    inv = (sp.eye(2) - discountfactor*Tss).inv();
    inv.simplify()  # sp.simplify() often helps

    Vis = (1-discountfactor) *\
        sp.Matrix(np.einsum(inv, [s,s_], Ris, [i, s_], [i, s]));

    return sp.simplify(Vis)
```

With the help of the function `compute_symbolic_statevalues`, we can now compute the state values for all four joint policies.

Mutual cooperation

The state value of the prosperous state, p, for the joint policy (c, c) is given by

```
statevalues_Vis_cc = compute_symbolic_statevalues(
    Xisa_cc, TransitionTensor, RewardTensor)
```

```
Vcc_p = statevalues_Vis_cc[0, p]
Vcc_g = statevalues_Vis_cc[0, g]
Vcc_p
```

$1.0c\,(r-1)$

Unilateral defection

The state value of the prosperous state, p, for the joint policy (d, c) is given by

```
statevalues_Vis_dc = compute_symbolic_statevalues(
    Xisa_dc, TransitionTensor, RewardTensor)
```

```
Vdc_p = statevalues_Vis_dc[0, p]
Vdc_g = statevalues_Vis_dc[0, g]
Vdc_p
```

$$\frac{-0.5c\gamma p_r q_c r + 1.0c\gamma p_r r + 0.5c\gamma q_c r - 1.0c\gamma r - 0.5cq_c r + 1.0cr + 1.0\gamma mp_r q_c + 1.0mq_c}{2.0\gamma p_r + 1.0\gamma q_c - 2.0\gamma + 2.0}$$

178

Unilateral cooperation

The state value of the prosperous state, p, for the joint policy (c, d) is given by

```
statevalues_Vis_cd = compute_symbolic_statevalues(
    Xisa_cd, TransitionTensor, RewardTensor)
```

```
Vcd_p = statevalues_Vis_cd[0, p]
Vcd_g = statevalues_Vis_cd[0, g]
Vcd_p
```

$$\frac{-0.5c\gamma p_r q_c r + 1.0c\gamma p_r q_c + 1.0c\gamma p_r r - 2.0c\gamma p_r + 0.5c\gamma q_c r - 1.0c\gamma q_c - 1.0c\gamma r + 2.0c\gamma - 0.5cq_c r + 1.0cq_c + 1.0cr -}{2.0\gamma p_r + 1.0\gamma q_c - 2.0\gamma + 2.0}$$

Mutual defection

The state value of the prosperous state, p, for the joint policy (d, d) is given by

```
statevalues_Vis_dd = compute_symbolic_statevalues(
    Xisa_dd, TransitionTensor, RewardTensor)
```

```
Vdd_p = statevalues_Vis_dd[0, p]
Vdd_g = statevalues_Vis_dd[0, g]
Vdd_p
```

$$\frac{1.0mq_c\left(\gamma p_r + 1\right)}{\gamma p_r + \gamma q_c - \gamma + 1}$$

8.3.7 Critical curves

Finally, we can compute the critical conditions on the model parameters where the agents' incentives change. The three conditions are:

- **Dilemma**: The agents are indifferent between all cooperating and all defecting, $v_{cc} = v_{dd}$,
- **Greed**: Each individual agent is indifferent between cooperating and defecting, given all others cooperate, $v_{cc} = v_{dc}$, and
- **Fear**: Each individual agent is indifferent between cooperating and defecting, given all agents defect, $v_{cd} = v_{dd}$.

Without greed the action situation becomes a coordination challenge between two pure equilibria of mutual cooperation and mutual defection. Without greed and fear the only Nash equilibrium left is mutual cooperation.
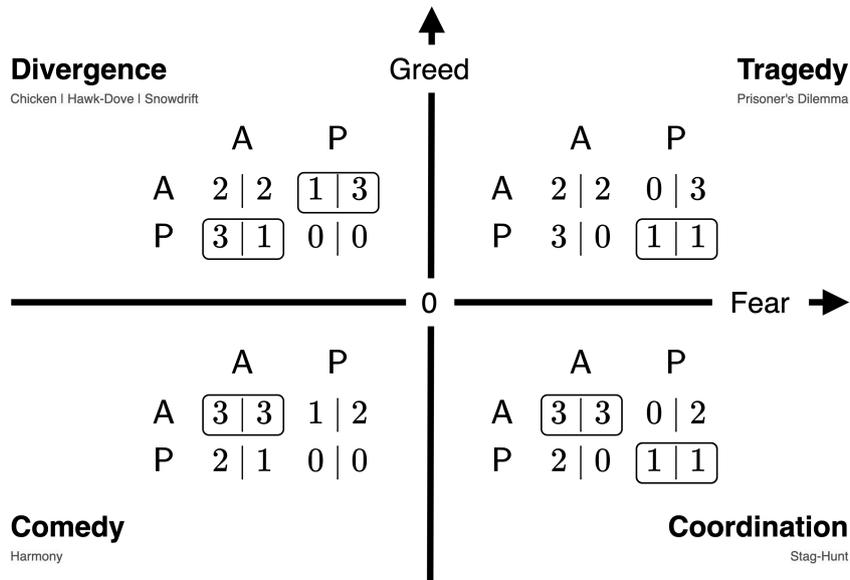
Figure 8.4: Dimensions of a social dilemma with ordinal payoffs and Nash equilibira shown in boxes from 03.02-StrategicInteractions.

Dilemma

The critical curve at which collapse avoidance becomes collectively optimal is obtained by setting $v_{cc} = v_{dd}$. Solving for the collapse impact m yields,

```
dilemma_m = sp.solve(Vdd_p - Vcc_p, m)[0]
dilemma_m
```

$$\frac{c\left(\gamma p_r r - \gamma p_r + \gamma q_c r - \gamma q_c - \gamma r + \gamma + r - 1.0\right)}{q_c\left(\gamma p_r + 1.0\right)}$$

We verify that the dilemma condition does not depend on environmental state by computing it also for the degraded state.

```
sp.solve(Vdd_g - Vcc_g, m)[0] - dilemma_m
```

0

Greed

The critical curve at which agents become indifferent to greed, i.e., exactly where cooperators are indifferent to cooperation and defection, given all other actors cooperate, is obtained by setting $v_{cc} = v_{dc}$. Solving for the collapse impact m yields,

```
greed_m = sp.solve(Vdc_p - Vcc_p, m)[0]
greed_m
```

$$\frac{0.5c\left(\gamma p_r q_c r + 2.0\gamma p_r r - 4.0\gamma p_r + \gamma q_c r - 2.0\gamma q_c - 2.0\gamma r + 4.0\gamma + q_c r + 2.0r - 4.0\right)}{q_c\left(\gamma p_r + 1.0\right)}$$

We verify that the greed condition does not depend on the environmental state by computing it also for the degraded state.

```
sp.solve(Vdc_g - Vcc_g, m)[0] - greed_m
```

0

Fear

The critical curve at which actors are indifferent to fear, i.e., exactly where defectors are indifferent to cooperation and defection, given all other actors defect, is obtained by setting $v_{cd} = v_{dd}$. Solving for the collapse impact m yields,

```
fear_m = sp.solve(Vcd_g - Vdd_g, m)[0]
fear_m
```

$$\frac{0.5c\left(-\gamma p_r q_c r + 2.0\gamma p_r q_c + 2.0\gamma p_r r - 4.0\gamma p_r - \gamma q_c^2 r + 2.0\gamma q_c^2 + 3.0\gamma q_c r - 6.0\gamma q_c - 2.0\gamma r + 4.0\gamma - q_c r + 2.0q_c + 2.}{q_c\left(\gamma p_r + 1.0\right)}$$

We verify that also the fear condition does not depend on the environmental state by computing it also for the degraded state.

```
sp.solve(Vcd_g - Vdd_g, m)[0] - fear_m
```

0

8.3.8 Visualization

Having obtained symbolic expressions for the critical curves, we can now visualize them as a function of the discount factor γ, indicating how much the agents value future rewards.

Parameter values

Let us apply the model to the case of global sustainability. We set public goods synergy factors $r = 1.2$ and the cost of cooperation to 5.

```
vr = 1.2
vc = 5
```

Regarding the transition probabilities, we apply the conversion rule developed in 02.04-StateTransitions to set the collapse leverage, q_c and the recovery probability p_r, in terms of typical timescales. Under current business-as-usual policies, there are about 50 years left to avert the collapse (Rockström et al., 2017). After a collapse, there is a potential lock-in into an unfavorable Earth system state for a timescale up to millennia (Steffen et al., 2018). Interpreting a time step as one year, we set the collapse leverage to $q_r = 1/50 = 0.02$ and the recovery probability to $p_r = 1/10000 = 0.0001$.

```
vqc = 0.02
vqr = 0.0001
```

Final graphic

We convert the symbolic expressions to numerical functions using `lambdify`.

```
Fdilemma_m = sp.lambdify((r, cost, qc, pr, dcf), dilemma_m, 'numpy')
Fgreed_m = sp.lambdify((r, cost, qc, pr, dcf), greed_m, 'numpy')
Ffear_m = sp.lambdify((r, cost, qc, pr, dcf), fear_m, 'numpy')
```

```
dcf_values = np.linspace(0.95, 1.0, 100)

plt.plot(dcf_values, Fdilemma_m(vr, vc, vqc, vqr, dcf_values),
         c='k', label='Dilemma')
plt.plot(dcf_values, Fgreed_m(vr, vc, vqc, vqr, dcf_values),
         c='b', label='Greed')
plt.plot(dcf_values, Ffear_m(vr, vc, vqc, vqr, dcf_values),
         c='g', label='Fear')

plt.fill_between(dcf_values, 0, 4, color='gray', alpha=0.4)

plt.annotate("Tragedy", (0.975, -1.6), ha='center', va='center')
plt.annotate("Coordination", (0.987, -4), ha='center', va='center')
plt.annotate("Comedy", (0.996, -6), ha='center', va='center')
plt.annotate("Collapse avoidance suboptimal",
             (0.9999, 2.5), ha='right', va='center')

plt.legend(loc='center left'); plt.ylim(-7, 3); plt.xlim(0.955, 1.0);
plt.xlabel('Discount factor $\gamma$'); plt.ylabel('Collapse impact $m$');
```
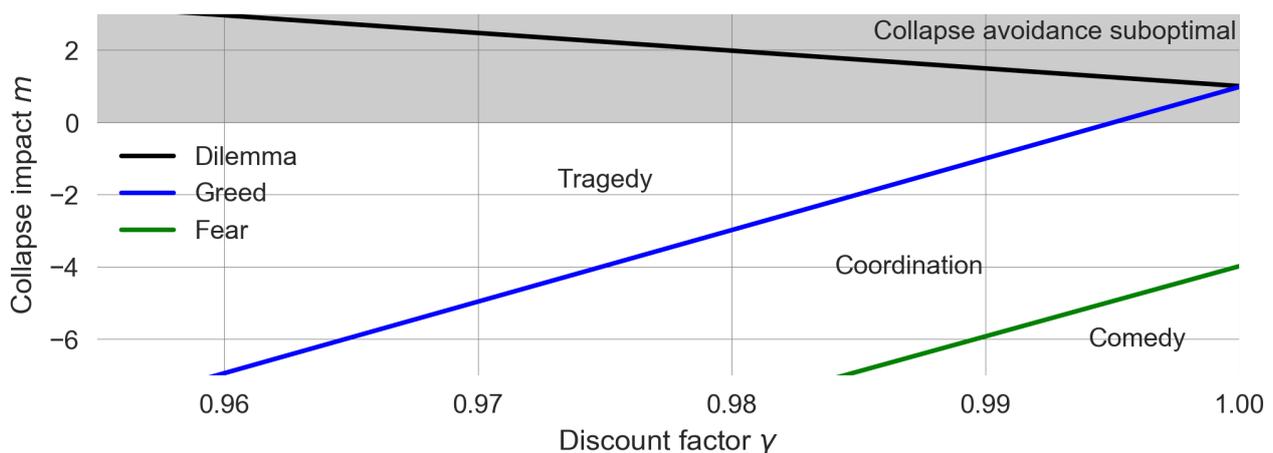

This plot is a precise reproduction of the result by (Barfuss et al., 2020). It highlights that

- the **same** *care for the future* that makes **individual decision-making** apt for the long-term also positively impacts collective decision-making.
- This *care for the future* **alone can turn a tragedy into a comedy**, where individual incentives are fully aligned with the collective interest - completely resolving the social dilemma.

- This is true, **given the anticipated collapse impact is sufficiently severe**. Thus, agents need to consider the catastrophic impact and, at the same time, be immensely future-oriented.
- **No other mechanism is required**: Agents are anonymous, cannot reciprocate, and cannot make any agreements.

8.4 Learning goals revisited

In this chapter,

- we introduced the concept of dynamic games and described the elements of a stochastic game.
- We applied the stochastic games framework to model human-environment interactions on the question, how the individual care for future consequences and the fact that we all are embedded into a shared environment impacts collective decision-making.
- We analyze the strategic interactions in this stochastic game model and found that caring for the future can turn a tragedy into a commedy of the commons.

Part III

Transformation Agency

In this last part, we cover transformation-agency or agent-based models. They operationalize **transformation knowledge**, which is knowledge about how to move from the existing system to the desired future. This knowledge includes concrete strategies and steps to take. In sustainability transitions, producing transformation knowledge could involve developing policy instruments, designing new institutions, or implementing new technologies. Transformation knowledge is strongly associated with **agency** and asks **how to?**.

Transformation-agency models (or *agent-based models applied to sustainability transitions*). Agent-based modeling (ABM) can be viewed as a merger between **dynamic systems** and **target equilibrium** models, in that typical agent-based models are about the *dynamics of agent behavior*. However, agent-based modeling in itself is diverse. Therefore, we can only provide a brief overview and specific aspects of ABM in this part.
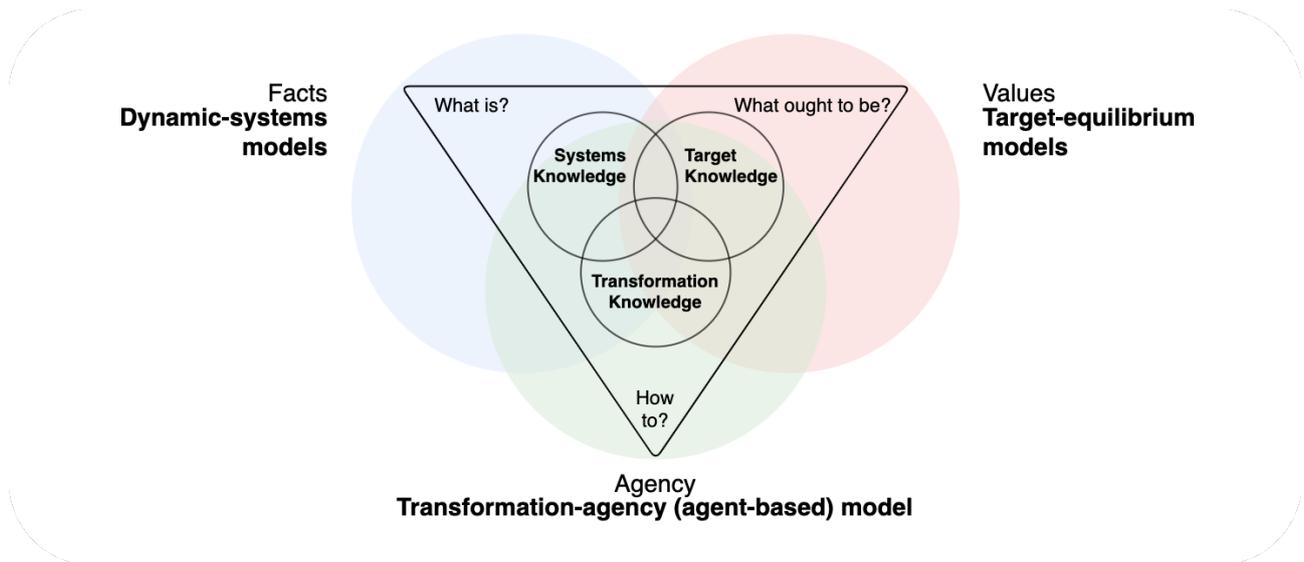
Figure 8.5: Three types of models based on three types of knowledge for transdisciplinary reserach

Specifically, we will cover

- Rule-based behavioral agency in agent-based models in Chapter 04.01
- Individual reinforcement learning in Chapter 04.02, and
- The non-linear dynamics of reinforcement learning in Chapter 04.03.

9 Behavioral agency

9.1 Motivation | Agent-based modeling of complex systems

The distinctive feature of the science of complex systems is the fascination that arises when **the whole becomes greater than the sum of its parts** (Figure 9.1) and properties on the macro-level emerge that do not exist on the micro-level.

Macro-level
Emergent collective behavior

Emergence

Feed-back

Time evolving
Non-equilibrium transient dynamics

Micro-level
Often **heterogeneous** adaptive **agents / entities** **relationally interacting** **in a shared environment** In non-linear ways
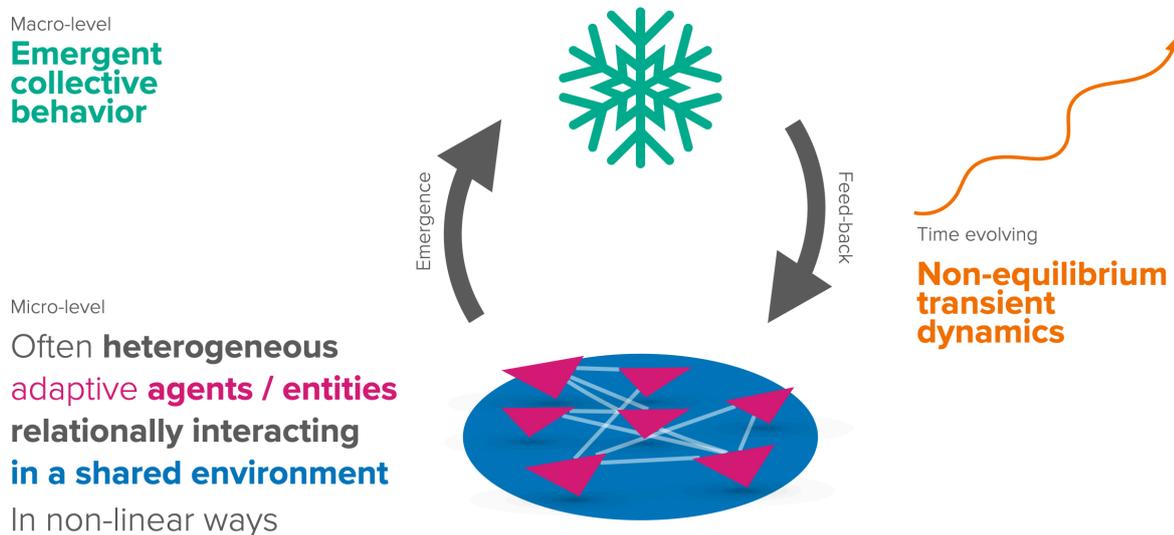
Figure 9.1: Properties of a complex system

Complex systems thinking is instrumental in understanding the **interactions** between **society** and **nature** (Figure 9.2).

Agent-based models capture the features of a complex system in the most direct way, in that they model the behavior of individual agents and their interactions.

9.1.1 Learning goals

After this lecture, students will be able to:

1) Explain the history and rationale of agent-based modeling and generative social science
2) Explain the advantages and challenges of agent-based modeling
3) Simulate two famous agent-based models in Python
4) Implement animations in Python
5) Write object-oriented Python programs

With a
Embedded economy
In an
Embedded society
In
Nature
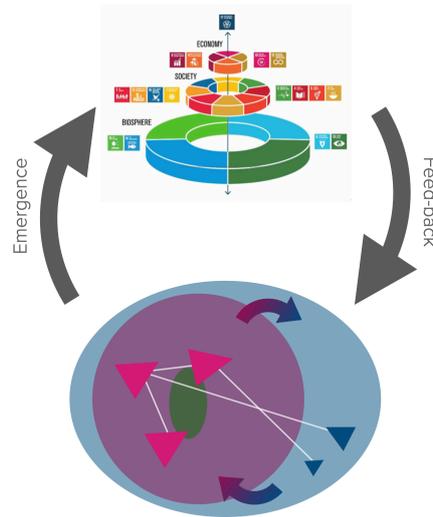
See e.g., Reyers et al. (2018) Social-Ecological Systems Insights for Navigating the Dynamics of the Anthropocene

Figure 9.2: Complex Society-Nature Systems

```python
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive
import matplotlib.animation as animation
from IPython.display import HTML
from functools import partial

import matplotlib.style as style; style.use('seaborn-v0_8')
plt.rcParams['figure.figsize'] = (7.8, 2.5); plt.rcParams['figure.dpi'] = 300
plt.rcParams['axes.facecolor'] = 'white'; plt.rcParams['grid.color'] = 'gray';
plt.rcParams['grid.linewidth'] = 0.25;
```

9.2 Overview | Generative social science

9.2.1 Essence of agent-based modeling

An agent-based model **explicitly** represents the **individual units** of the system and their repeated **interactions**. Beyond this, no further assumptions are made in agent-based modeling (Izquierdo et al., 2024).

Hence, agent-based modeling is **a very flexible modeling framework**.

> You can model anything with an agent-based model, but you are not guaranteed to understand it.

Adding complexity to your model allows you to study different phenomena. However, it can make the analysis and understanding of the model more difficult, even with advanced mathematics. As a result, agent-based models are usually created using a programming language and analyzed through computer simulation. This method is so common that agent-based modeling and simulation are often seen as synonymous (Izquierdo et al., 2024).

9.2.2 Generative Social Science

The flexibility of ABM offers a *generative* approach to social science (Epstein, 1999). **Explaining** the **emergence** of **macroscopic societal regularities** requires that one answers the following question:

> How could the **decentralized local interactions** of **heterogeneous autonomous agents** *generate* the **given regularity**?

Generative social science is a research approach that uses computational models to generate and explain complex social behaviors. It is characterized by a

- focus on explanation over prediction and
- its use of agent-based models to bridge the gap between micro and macro phenomena.

9.2.3 Features of agent-based modeling

Agent-based modeling allows us to cover some features that traditionally have been difficult to analyze mathematically (Epstein, 1999; Izquierdo et al., 2024):

- **Bounded Rationality**, which has two components: bounded information and bounded computing power. Typically, agents use more or less sophisticated rules or heuristics based on local information. They do not have global information or infinite computational power.

- **Autonomy and the micro-macro link**. ABM is particularly well suited to studying how global phenomena emerge from individual interactions and how these emergent global phenomena may constrain and shape individuals' actions. In agent-based models, there is no central, or "top-down," control over individual behavior. Micro and macro will typically co-evolve.

- **Out-of-equilibrium dynamics**. Dynamics are inherent to ABM. Running a simulation consists of repeatedly applying the rules that define the model. Equilibria are never imposed a priori; they may emerge as an outcome of the simulation, or they may not.

- **Agents' heterogeneity**. Since agents are explicitly modeled, their diversity can vary according to the modeler's preferences. Agents may differ in myriad ways — genetically, culturally, by social network, by preferences — all of which may change or adapt endogenously over time. Representative or aggregated agent methods - common in dynamic systems or target equilibria models - or not used.

- **Local interactions and explicit space**. The fact that agents and their environment are represented explicitly in ABM makes it particularly straightforward and natural to model local interactions.

- **Interdependencies between processes** (e.g., demographic, economic, biological, geographical, technological) that have traditionally been studied in different disciplines and are not often analyzed together. An agent-based model does not restrict the type of rules that can be implemented, so models can include rules linking disparate aspects of the world that are often studied in different disciplines. This feature makes ABM particularly well suited to studying complex Nature-Society systems.

9.2.4 The generativist's experiment

A given macroscopic regularity is to be explained by the canonical agent-based experiment as follows (Epstein, 1999):

> Situate an **initial population** of **autonomous, heterogeneous agents** in a relevant spatial environment; - allow them to **interact** according to simple **local rules**, - and thereby generate - or "**grow**" - the macroscopic regularity from the bottom up.

If you didn't grow it, you didn't explain its emergence. This generativist approach to social science has been successfully applied to explain the following phenomena:

- economic classes (Axtell etl al. 1999)
- cooperation in spatial games (Lindgren and Nordahl, 1994; Epstein, 1998; Huberman and Glance, 1993; Nowak and May, 1992; Miller, 1996)
- voting behaviors (Kollman, Miller, and Page, 1992),
- demographic histories (Dean et al. 1999)
- trade networks (Tesfatsion, 1995; Epstein and Axtell, 1996),
- right-skewed wealth distributions (Epstein and Axtell, 1996)

... and many more!

9.2.5 Variants of agent-based modeling

As agent-based modeling is so flexible, various subcategories of models can be distinguished:

- Cellular automata
- Network models
- Learning and evolution models in games
- ...

9.3 Example | Conway's Game of Life

The Game of Life is a **cellular automaton** devised by the British mathematician John Horton Conway in 1970.

A cellular automaton is a discrete model of computation consisting of a regular grid of cells, each in one of a finite number of states (e.g., on and off).

The game of life is a very influential model in the field of complex systems (although Conway wasn't particularly proud of it)

See Inventing Game of Life (John Conway) - Numberphile for a brief backstory on the game of life.

9.3.1 Questions

The game of life is a comparably simple model to answer two very fundamental questions:

- How can something reproduce itself?
- How can a complex structure (like the mind) emerge from a basic set of rules?

9.3.2 States

The cells of the cellular automaton can be in one of two states: dead or alive.

We can represent the state of a cell with a binary variable: 1 (black) for alive and 0 (white) for dead. The state of the whole system can the be represented as follows:

```
ROWS, COLS = 40, 100  # define the size of the grid
grid = np.random.choice([0, 1], size=(ROWS, COLS), p=[0.7, 0.3]) #generate random
↳  states
plt.imshow(grid, cmap='binary', interpolation='none') # plot the grid
plt.gca().set_xticks([]); plt.gca().set_yticks([]);  # remove x and y ticks
```


Figure 9.3: A Game-of-Life grid

9.3.3 Dynamics

The dynamics of the game of life are governed by the following rules:

- Living cells with fewer than two living neighbors die
- Living cells with more than three living neighbors die
- Dead cells with exactly three neighbors become alive

We will use the `matplotlib.animation.FuncAnimation` function to animate the game of life in Python. To do so, we need to implement the game's rules in a function that updates the grid.

It must receive the number of the current `frame` of the animation plus possible further function arguments. We supply it with an `image` argument variable representing the grid's image. It must return an iterable of artists, which the `matplotlib.animation.FuncAnimation` will use to update the plot.

```
# Function to update the grid based on the Game of Life rules
def update_grid(frame, image):
    global grid  # required to access the variable inside the function
    new_grid = grid.copy()

    for i in range(0, ROWS):
        for j in range(0, COLS):
            neighbors_sum = (  # the % sign is a modulo division, i.e., 13 % 13 = 0
                grid[(i - 1) % ROWS, (j - 1) % COLS] + grid[(i - 1) % ROWS, j] +
```

```
                grid[(i - 1) % ROWS, (j + 1) % COLS] + grid[i, (j - 1) % COLS] +
                grid[i, (j + 1) % COLS] + grid[(i + 1) % ROWS, (j - 1) % COLS] +
                grid[(i + 1) % ROWS, j] + grid[(i + 1) % ROWS, (j + 1) % COLS])

            # the rules of the game
            if grid[i, j] == 1 and (neighbors_sum < 2 or neighbors_sum > 3):
                new_grid[i, j] = 0
            elif grid[i, j] == 0 and neighbors_sum == 3:
                new_grid[i, j] = 1

    grid = new_grid
    image.set_array(grid)
    return image,  # must return an iterable of artists
```

With the `update_grid` function in place, we can now create the animation using the `FuncAnimation` function. The `%%caputure` magic command is used to suppress the output of the cell below, as we will call the animation separately.

```
%%capture
# Set up the Matplotlib figure and axis
fig, ax = plt.subplots(figsize=(16,9))
im = ax.imshow(grid, cmap='binary', interpolation='none')
ax.set_xticks([]); ax.set_yticks([])

# Create animation
ani = animation.FuncAnimation(fig, partial(update_grid, image=im),
                              frames=150, interval=150)
```

Finally, we can display the animation using the `HTML` function from the `IPython.display` module.

```
# Display the animation using HTML
# HTML(ani.to_jshtml())
```

9.3.4 Emerging structures

Despite the simplicity of the rules, complex structures of *species* **moving** and **reproducing** can emerge from these rules, despite them not having any concept of movement or reproduction.

See, for example, the Epic Conway's Game of Life or Life in life videos.

9.3.5 Impact

Although the rules are incredibly simple, it is impossible to say whether a given configuration persists or eventually dies out. **There are fundamental limits to prediction**.

It was shown that you can do any form of computation (that you can do on a regular computer) with the game of life.

Complex behavior does not require complicated rules. Complex behavior can emerge from simple rules. This realization has been a key insight of complexity sciences and has shaped the way complexity science is done today.

9.4 Example | Schelling's segregation model

The second example model studies the phenomenon of racially segregated neighborhoods. The content here is heavily inspired by QuantEcon's Quantitative Economics with Python.

9.4.1 Questions

We observe racially segregated neighborhoods.

Does that mean that all residents are racists?

9.4.2 Context

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation.

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

9.4.3 The Model

We will cover a variation of Schelling's model that is easy to program and captures the main idea.

Set-Up

Suppose we have two types of people: orange people and green people.

For the purpose of this lecture, we will assume there are 250 of each type.

These agents all live on a single-unit square.

The location of an agent is just a point (x, y), where $0 < x, y < 1$.

Preferences

We will say that an agent is *happy* if half or more of her 10 nearest neighbors are of the same type.

Here 'nearest' is in terms of Euclidean distance.

An agent who is not happy is called *unhappy*.

An important point here is that agents are not averse to living in mixed areas.

They are perfectly happy if half their neighbors are of the other color.

Behavior

Initially, agents are mixed together (integrated).

In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on $S = (0, 1)^2$.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

We assume that each agent will stay put if they are happy and move if unhappy.

The algorithm for moving is as follows

1. Draw a random location in S

2. If happy at the new location, move there

3. Else, go to step 1

In this way, we cycle continuously through the agents, moving as required.

We continue to cycle until no one wishes to move.

9.4.4 Implementation

We use object-oriented programming (OOP) to model agents as objects.

OOP is a programming paradigm based on the concept of objects, which can contain data and code: - data in the form of fields (often known as attributes or properties), and - code in the form of procedures (often known as methods).

Agent class

A class defines how an object will work. Typically, the class will define several methods that operate on instances of the class. A key method is the `__init__` method, which is called when an object is created.

```python
class Agent:

    # The init method is called when the object is created.
    def __init__(self, type, num_neighbors, require_same_type):
        self.type = type
        self.draw_location()
        self.num_neighbors = num_neighbors
        self.require_same_type = require_same_type

    def draw_location(self):
        self.location = np.random.uniform(0, 1), np.random.uniform(0, 1)

    def get_distance(self, other):
        "Computes the Euclidean distance between self and another agent."
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
        return np.sqrt(a + b)
```

```python
    def number_same_type(self, agents):
        "Number of neighbors of same type."
        distances = []
        # distances is a list of pairs (d, agent), where d is the distance from
        # agent to self
        for agent in agents:
            if self != agent:
                distance = self.get_distance(agent)
                distances.append((distance, agent))
        # == Sort from smallest to largest, according to distance == #
        distances.sort()
        # == Extract the neighboring agents == #
        neighbors = [agent for d, agent in distances[:self.num_neighbors]]
        # == Count how many neighbors have the same type as self == #
        return sum(self.type == agent.type for agent in neighbors)

    def happy(self, agents):
        "True if a sufficient number of nearest neighbors are of the same type."
        num_same_type = self.number_same_type(agents)
        return num_same_type >= self.require_same_type

    def update(self, agents):
        "If not happy, then randomly choose new locations until happy."
        while not self.happy(agents):
            self.draw_location()
```

Testing the agent class:

```python
A = Agent(0, num_neighbors=4, require_same_type=2)
type(A)
```

```
__main__.Agent
```

```python
A.location
```

```
(0.9530720480796, 0.566243724536969)
```

Creating a list of agents:

```python
np.random.seed(4)
agents = [Agent(0, 4, 2) for i in range(100)]
agents.extend(Agent(1, 4, 2) for i in range(100))
len(agents)
```

```
200
```

Is agent three happy?

```
a3 = agents[3]
a3.happy(agents), a3.location
```

```
(False, (0.9762744547762418, 0.006230255204589863))
```

Let's let agent three update its position:

```
a3.update(agents)
```

```
agents[3].happy(agents), agents[3].location
```

```
(True, (0.06780815958339637, 0.961674586087924))
```

Observation function

We implement a function to plot the distribution of agents.

```
def plot_distribution(agents, cycle_num, ax=None):
    "Plot the distribution of agents after cycle_num rounds of the loop."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    # == Obtain locations of each type == #
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    if ax is None: fig, ax = plt.subplots(figsize=(4, 4))
    plot_args = {'markersize': 4, 'alpha': 0.6}
    # ax.set_facecolor('azure')
    ax.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange', **plot_args)
    ax.plot(x_values_1, y_values_1, 'o', markerfacecolor='green', **plot_args)
    ax.set_xticks([]); ax.set_yticks([])
    ax.set_title(f'Cycle {cycle_num}')
```

Testing the observation function,

```
num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0, num_neighbors, require_same_type) for i in range(num_of_type_0)]
agents.extend(Agent(1, num_neighbors, require_same_type) for i in
↪  range(num_of_type_1))
```

```
plot_distribution(agents, 0)
```

Cycle 0
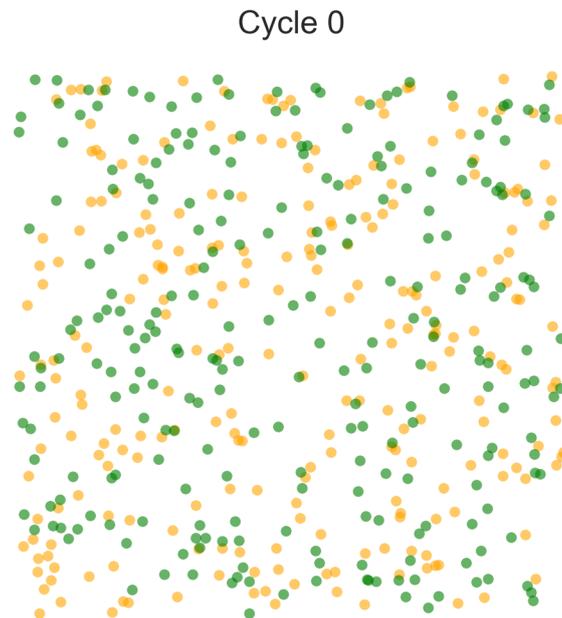
Figure 9.4: Initial distribution of agents.

Simulation run

```
np.random.seed(10)   # For reproducible random numbers

# == Main == #
num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0, num_neighbors, require_same_type) for i in range(num_of_type_0)]
agents.extend(Agent(1, num_neighbors, require_same_type) for i in
↪   range(num_of_type_1))

count = 1
# ==  Loop until none wishes to move == #
fig, axs = plt.subplots(2,3, figsize=(13, 6))
axs.flatten()

while True:
    print('Entering loop ', count)
    plot_distribution(agents, count, axs.flatten()[count-1])
    count += 1
```

```
    # Update and check whether everyone is happy
    no_one_moved = True
    for agent in agents:
        old_location = agent.location
        agent.update(agents)
        if agent.location != old_location:
            no_one_moved = False
    if no_one_moved:
        break

print('Converged, terminating.')
plt.tight_layout()
```

```
Entering loop  1
Entering loop  2
Entering loop  3
Entering loop  4
Entering loop  5
Entering loop  6
Converged, terminating.
```
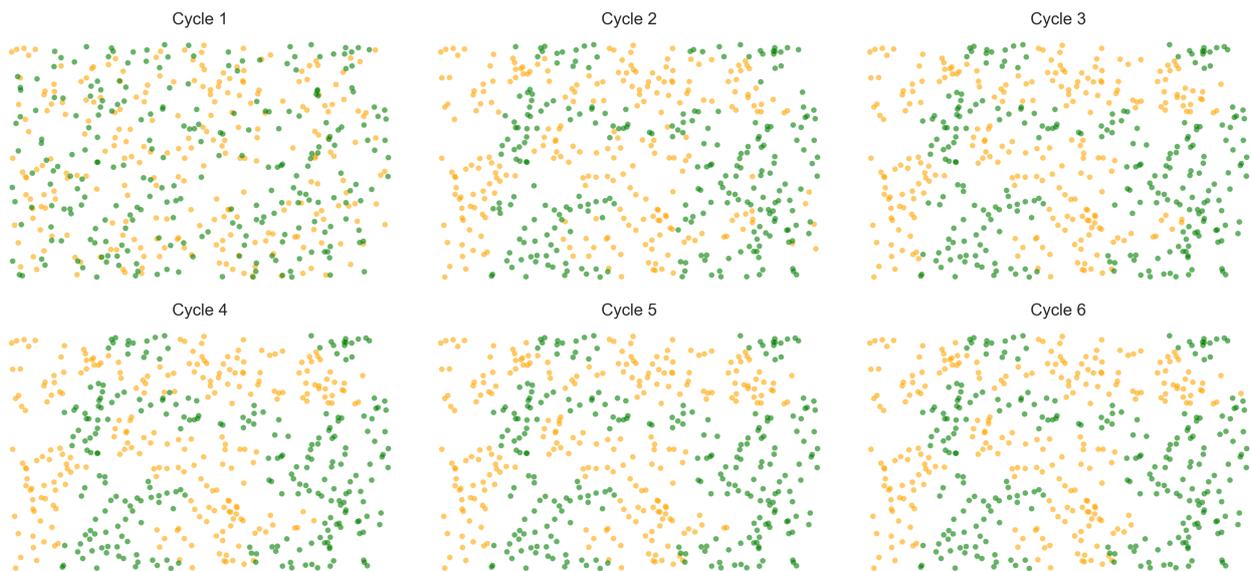

Figure 9.5: A simulation run of Schelling's model.

In this instance, the program terminated after 6 cycles through the set of agents, indicating that all agents had reached a state of happiness.

9.4.5 Interpretation

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

9.5 Challenges of agent-based modeling

- **Performance Limitations**. The execution speed of ABMs can be slow, which can be a limitation for extensive simulations.
- **Transparency and Reproducibility**. Providing a clear and accessible description is challenging due to model complexity.
- **Data Parameters and Validation**. Getting empirical data and validating models that may simulate unobservable associations is challenging.
- **Arbitrariness and Parameterization**. The many parameters that need to be set can lead to a high degree of arbitrariness.
- **Behavior modeling**. There are endless possibilities to design plausible behavioral rules. A sensitivity analysis is difficult.

Up next

- **Reinforcement learning** as a principled model for behavior to counter some arbitrariness in parameterization behavioral rules.
- Synthesis: Collective reinforcement learning **dynamics** to counter performance limitations and a lack of transparency and reproducibility.

9.6 Learning goals revisited

In this chapter,

we covered the history and rationale of agent-based modeling: **generative social science**.

We covered the advantages (**flexibility and expressiveness**) and challenges (**transparency**, **arbitrariness**, **performance**) of agent-based modeling

We implemented and simulated two famous agent-based models in Python: **Conway's Game of Life** and **Schelling's segregation model**.

We implement animations in Python using the `matplotlib.animations.FuncAnimation` function.

We wrote Schelling's segregation model as an **object-oriented program**.

10 Individual learning

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

10.1 Motivation

> Give a man a fish, and he'll eat for a day

> Teach a man to fish, and he'll eat for a lifetime

> Give a man a taste for fish, and he'll eat even if conditions change. [source]

In this chapter, we will introduce the basics of temporal-difference reward-prediction reinforcement learning.

10.1.1 Using behavioral theories in ABMs is challenging

General **agent-based modeling** is a flexible tool for studying different theories of human behavior. However, the social and behavioral sciences are not known for their tendency to integrate. **Knowledge about human behavior is fragmented into many different, context-specific, and often not formalized theories**. For example, in an attempt to order and use this knowledge for sustainability science, Constantino and colleagues presented a selection of 32 behavioral theories (Constantino et al., 2021).

Table 2 Theory map: linking theories to the framework

Conceptual origin of theories	Theories	Perception	Attention	Learning and updating	External information search	Memory search	Valuation	Choice	Behavior	Stable characteristics	Situational characteristics	Social and bio-physical environment
Constraints	Bounded rationality (satisficing)						x	x				
Constraints	Cognitive hierarchy theory (or K-level reasoning in behavioral game theory)						x	x				x
Constraints	Confirmation bias (theories related to selective and biased memory search)		x		x							
Constraints	Hyperbolic discounting (and other discounting theories, including psychological distance)					x						
Constraints	Prospect theory	x					x	x				x
Constraints	Psychological risk dimensions	x	x				x					x
Constraints	Selective attention	x	x		x	x				x		
Constraints	Query theory		x			x	x					
Context (physical)	Affordance theory	x							x	x	x	
Context (physical)	Attention restoration theory	x			x					x	x	
Context	Mental models	x		x		x				x		
Context (physical)	Sense of place						x	x		x	x	
Context (social)	Social identity theory and social categorization theory			x		x				x	x	
Context (social)	Social influence and persuasion						x	x		x		x
Context (social)	Social norms	x					x	x		x	x	x
Context (physical)	Stress reduction theory									x	x	
Dynamics	Active learning			x	x			x	x	x		
Dynamics	Reinforcement learning (incremental learning theories)	x		x						x	x	x
Dynamics	Social learning (in the context of NRM)			x						x	x	x
Multiplicity	Conditional cooperation (cooperation and reciprocity)							x			x	x
Multiplicity	Decision modes theory						x	x				x
Multiplicity	Inequity aversion (and fairness preferences)						x			x		x
Multiplicity	Theory of planned behavior						x	x		x		x
Multiplicity	Value-belief-norm theory							x	x	x	x	x
Multiplicity	Sunk cost accounting						x					
Multiplicity	Trust and reciprocity (trait-based)			x			x	x		x		x
Multiplicity	ABC theory (attitudes, behaviors, contexts)							x		x		x
Context	Choice architecture	x			x	x		x			x	x
Constraints, context	Embodied cognition	x	x		x	x		x		x	x	x
Dynamics	Habitual behavior	x		x				x		x	x	x
Context (social)	Social network structure (and social capital)			x							x	x

Constantino et al. (2021) Cognition and behavior in context: a framework and theories to explain natural resource use decisions in social-ecological systems

The many behavioral theories pose a significant **challenge for general agent-based modeling** when it comes to incorporating human decision-making into models of Nature-society systems (Schlüter et al., 2017):

1) **Fragmentation of theories**: A vast array of theories on human decision-making is scattered across different disciplines, making it difficult to navigate and select relevant theories. Each theory often focuses on specific aspects of decision-making, leading to **fragmented knowledge**.

2) **Incomplete theories**: The degree of formalization varies across theories. Many decision-making theories are incomplete or not fully formalized, requiring modelers to fill logical gaps with assumptions to make simulations work. This step introduces more degrees of freedom and possibly arbitrariness into the modeling process.

3) **Correlation-based theories**: Many theories focus on correlations rather than causal mechanisms, essential for dynamic modeling. This requires modelers to make explicit assumptions about causal relationships - introducing more degrees of freedom and possibly arbitrariness into the modeling process.

4) **Context-dependent theories**: The applicability of theories can vary greatly depending on the context, which adds complexity to their integration into models.

10.1.2 Reinforcement learning offers a principled take

Reinforcement learning (RL) offers a general prototype model for intelligent & adaptive decision-making in agent-based models.

Principle

"Do more of what makes you happy."

We do not need to specify the behavior of an agent directly.

Instead, we specify what an agent wants and how it learns. Crucially, how it learns does not depend on the details of the environment model. It is a more general process, applicable across different environments.

Then, it learns for itself what to do and we study the learning process and the learned behaviors.

This approach is **particularly valuable** for studying human-environment systems **when the decision environment changes** through a policy intervention or a global change process, like climate change or biodiversity loss. If we had specified the agent's behavior directly, the agent's behavior could not change when the environment changes. In contrast, if we specify the underlying agent's goal, we can study how the agent's behavior changes when the environment changes. Reinforcement learning agents can **learn while interacting** with the environment.

10.1.3 An integrating platform for cognitive mechanisms

RL, broadly understood, offers an interdisciplinary platform for integrating **cognitive mechanisms** into ABMs. It offers a comprehensive framework for studying the interplay among **learning** (adaptive behavior), **representation** (beliefs), and **decision-making** (actions) (Botvinick et al., 2020).
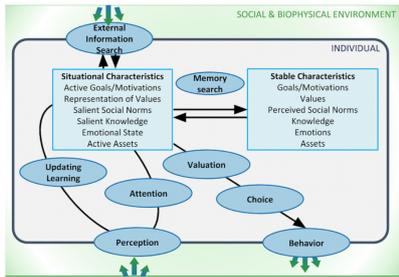
Constantino et al. (2021) Cognition and behavior in context: a framework and theories to explain natural resource use decisions in social-ecological systems
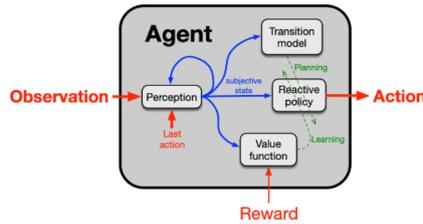
Sutton (2022) The Quest for a Common Model of the Intelligent Decision Maker
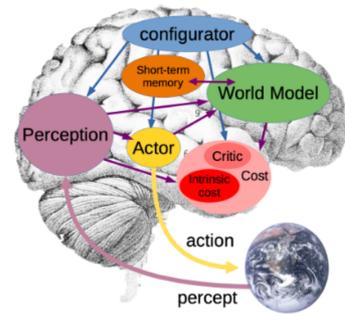
LeCun (2022) A Path Towards Autonomous Machine Intelligence

Figure 10.1: RL-based frameworks with cognitive mechanisms

Collective or multi-agent reinforcement learning is a natural extension of RL to study the emerging **collective behavior** of multiple agents in dynamic environments. It allows for formulating hypotheses on how different **cognitive mechanisms** affect **collective behavior** in **dynamic environments** (Barfuss, Flack, et al., 2024).

RL is also an **interdisciplinary endeavor**, studied in Psychology, Neuroscience, Behavioral economics, Complexity science, and Machine learning.
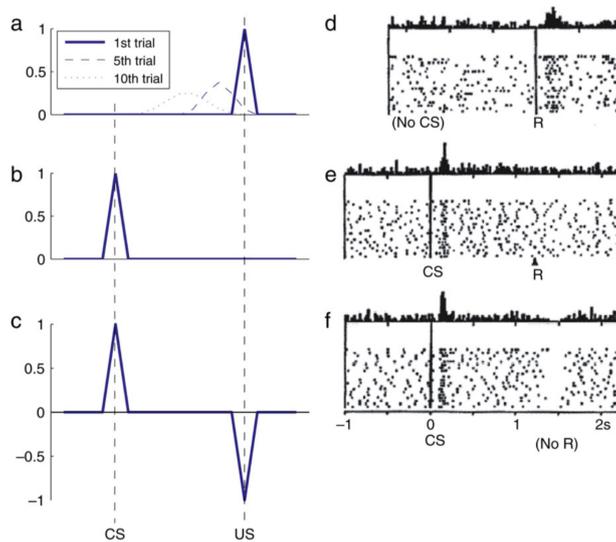
Niv (2009) Reinforcement learning in the brain

Figure 10.2: Reinforcement learning in the brain

Figure 10.2 shows the **remarkable analogy between the firing patterns of dopamine neurons in the brain and the prediction errors in a reinforcement learning simulation**.

Figure 10.2 (a-c) shows prediction errors in a Pavlovian RL conditioning task simulation. A conditional stimulus (CS) is presented randomly, followed 2 seconds later by a reward (Unconditional Stimulus - US). (a) In the early training phase, the reward is not anticipated, leading to prediction errors when the reward is presented. As learning occurs, these prediction errors begin to affect prior events in the trial (examples from trials 5 and 10) because predictive values are learned. (b) After learning, the

previously unexpected reward no longer creates a prediction error. Instead, the conditional stimulus now causes a prediction error when it occurs unexpectedly. (c) When the reward is omitted when expected, it results in a negative prediction error, signaling that what happened was worse than anticipated.

Figure 10.2 (d–f) Firing patterns of dopamine neurons in monkeys engaged in a similar instrumental conditioning task [SchultzEtAl1997]. Each raster plot shows action potentials (dots) with different rows for different trials aligned with the cue (or reward) timing. Histograms show combined activity across the trials below. (d) When a reward is unexpectedly received, dopamine neurons fire rapidly. (e) After conditioning with a visual cue (which predicted a food reward if the animal performed correctly), the reward no longer triggers a burst of activity; now, the burst happens at the cue's presentation. (f) If the food reward is omitted unexpectedly, dopamine neurons exhibit a distinct pause in firing, falling below their typical rate.

Source of confusion. Because of its broad scope and interdisciplinary nature, simply the phrase "reinforcement learning" can mean different things to different people. To mitigate this possible source of confusion, it is good to acknowledge that RL can refer to a **model of human learning**, an **optimization method**, a **problem description**, and a **field of research**.

10.1.4 Learning goals

After this chapter, students will be able to:

- Explain why reinforcement learning is valuable in models of human-environment interactions
- Implement and apply the different elements of the multi-agent environment framework, including a temporal-different learning agent.
- Explain and manage the trade-off between exploration and exploitation.
- Visualize the learning process
- Use the Python library `pandas` to manage data
- Refine their skills in object-oriented programming

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive
import matplotlib.animation as animation
from IPython.display import HTML
import sympy as sp
from copy import deepcopy

import matplotlib.style as style; style.use('seaborn-v0_8')
plt.rcParams['figure.figsize'] = (15, 4)
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;
plt.rcParams['figure.dpi'] = 140
```

10.2 Elements of the multi-agent environment interface

Generally, making sense of an agent without its environment is difficult, and vice versa.
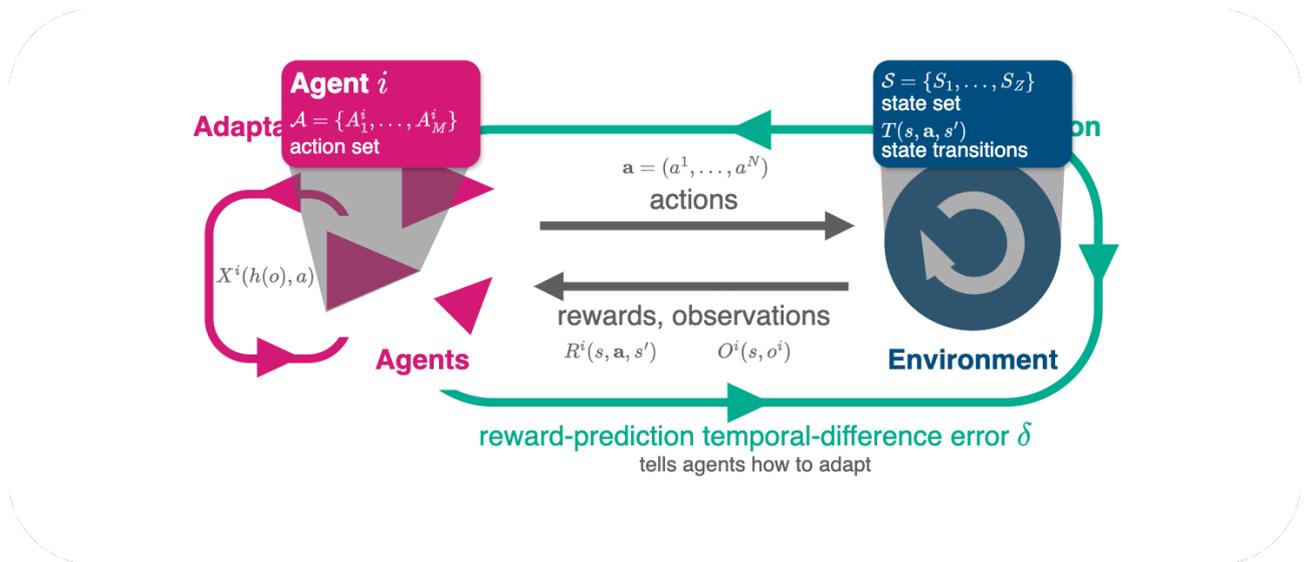
Figure 10.3: Reinforcement learning in the multi-agent environment interface

Interface

At the interface between agents and the environment are

- each agent's **set of** (conceivable) **actions** - from agents to environment,
- extrinsic **reward signals** - a single number from environment to each agent,
- possibly **observation signals** - from environment to agents.

Note: In general, the environment is composed of the *natural* and the *social* environment.

```python
def interface_run(agents, env, NrOfTimesteps):
    """Run the multi-agent environment for several time steps."""

    observations = env.observe()

    for t in range(NrOfTimesteps):

        actions = [agent.act(observations[i])
            for i, agent in enumerate(agents)]

        next_observations, rewards, info = env.step(actions)

        for i, agent in enumerate(agents):
            agent.update(observations[i], actions[i], rewards[i],
↪  next_observations[i])

        observations = next_observations
```

Environment

The environment delivers **extrinsic rewards** (motivations) to the agents based on the **agents' chosen actions** (choices). It may contain environmental **states**, which may not be fully **observable** to the agents

The most common **environment classes**:

	Agents	Environment	Observation
Multi-armed bandit	one	no states	-
Normal-form game	multiple	no states	-
Markov decision process	one	multiple states	full
Stochastic/Markov games	multiple	multiple states	full
Partially observable Markov decision process	one	multiple states	partial
Partially observable stochastic games	multiple	multiple states	partial

In all cases, **reward signals** may be **stochastic** and or **multi-dimensional**.

```python
class Environment:
    """Abstract environment class."""

    def obtain_StateSet(self):
        """Default state set representation `state_s`."""
        return [str(s) for s in range(self.Z)]

    def obtain_ActionSets(self):
        """Default action set representation `action_a`."""
        return [str(a) for a in range(self.M)]

    def step(self,
             jA # joint actions
            ) -> tuple:  # (observations_Oi, rewards_Ri, info)
        """
        Iterate the environment one step forward.
        """
        # choose a next state according to transition tensor T
        tps = self.TransitionTensor[tuple([self.state]+list(jA))].astype(float)
        next_state = np.random.choice(range(len(tps)), p=tps)

        # obtain the current rewards
        rewards = self.RewardTensor[tuple([slice(self.N),self.state]
                                          +list(jA)+[next_state])]

        # advance the state and collect info
        self.state = next_state
        obs = self.observe()

        # report the true state in the info dict
        info = {'state': self.state}

        return obs, rewards.astype(float), info

    def observe(self):
```

```
        """Observe the environment."""
        return [self.state for _ in range(self.N)]
```

Agents

Agents act (oftentimes to reach a goal). We need to specify their

- **Actions**, describing which choices are available to the agent.
- **Goal**, describing what an agent wants (in the long run). They may contain *intrinsic motivations*.
- **Representation**, e.g., defining upon which conditions agents select actions (e.g., *history* of past actions in multi-agent situations).
- **Value beliefs** (value functions), capturing what is *good* for the agent regarding its *goal* in the long run.
- **Behavioral rule** (policy, strategy), defining how to select actions.
- **Learning rule**, describing how value beliefs are updated in light of new information.
- (optionally), a **model** of the environment and rewards. Models are used for *planning*, i.e., deciding on a *behavioral rule* by considering possible future situations before they are actually experienced.

```
class BehaviorAgent:

    def __init__(self, policy):
        self.policy_Xoa = policy / policy.sum(-1, keepdims=True)
        self.ActionIxs = range(self.policy_Xoa.shape[1])

    def act(self, obs):
        return np.random.choice(self.ActionIxs, p=self.policy_Xoa[obs])

    def update(self, *args, **kwargs):
        pass
```

10.3 Example | Risk Reward Dilemma
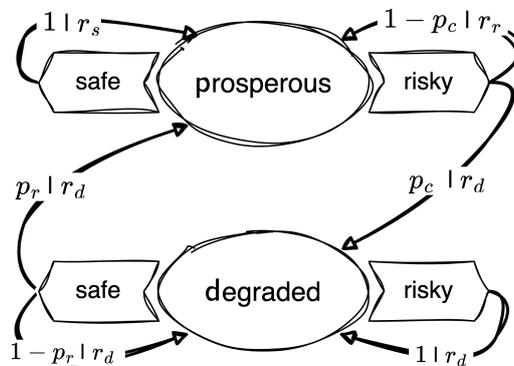
Figure 10.4: Risk Reward Dilemma

```python
class RiskRewardDilemma(Environment):
    """A simple risk-reward dilemma environment."""

    def obtain_StateSet(self):
        return ['p', 'd']   # prosperous, degraded

    def obtain_ActionSets(self):
        return [['c', 'r']]   # cautious, risky
```

10.3.1 Transitions | Environmental dynamics

The **environmental dynamics**, i.e., the transitions between environmental state contexts are modeled by two parameters: a collapse probability, p_c, and a recovery probability, p_r.

```python
pc, pr = sp.symbols('p_c p_r')
pc
```

p_c

```python
p = RiskRewardDilemma().obtain_StateSet().index('p')
d = RiskRewardDilemma().obtain_StateSet().index('d')
c = RiskRewardDilemma().obtain_ActionSets()[0].index('c')
r = RiskRewardDilemma().obtain_ActionSets()[0].index('r')
p,d,c,r
```

```
(0, 1, 0, 1)
```

We implement the transitions as a three-dimensional array or **tensors**, with dimensions $Z \times M \times Z$, where Z is the number of states and M is the number of actions.

```python
T = np.zeros((2,2,2), dtype=object)
```

The cautious action guarantees to remain in the prosperous state, $T(\mathsf{p}, \mathsf{c}, \mathsf{p}) = 1$. Thus, the agent can avoid the risk of environmental collapse by choosing the cautious action, $T(\mathsf{p}, \mathsf{c}, \mathsf{d}) = 0$.

```python
T[p,c,d] = 0
T[p,c,p] = 1
```

The risky action risks the collapse to the degraded state, $T(\mathsf{p}, \mathsf{r}, \mathsf{d}) = p_c$, with a collapse probability p_c. Thus, with probability $1 - p_c$, the environment remains prosperous under the risky action, $T(\mathsf{p}, \mathsf{r}, \mathsf{p}) = 1 - p_c$.

```python
T[p,r,d] = pc
T[p,r,p] = 1-pc
```

At the degraded state, recovery is only possible through the cautious action, $T(\mathsf{d}, \mathsf{c}, \mathsf{p}) = p_r$, with recovery probability p_r. Thus, with probability $1 - p_r$, the environment remains degraded under the cautious action, $T(\mathsf{d}, \mathsf{c}, \mathsf{d}) = 1 - p_r$.

```
T[d,c,p] = pr
T[d,c,d] = 1-pr
```

Finally, the risky action at the degraded state guarantees a lock-in in the degraded state, $T(\mathsf{d},\mathsf{r},\mathsf{d}) = 1$. Thus, the environment cannot recover from the degraded state under the risky action, $T(\mathsf{d},\mathsf{r},\mathsf{p}) = 0$.

```
T[d,r,p] = 0
T[d,r,d] = 1
```

Last, we make sure that our transition tensor is normalized, i.e., the sum of all transition probabilities from a state-action pair to all possible next states equals one, $\sum_{s'} T(s, a, s') = 1$.

```
T.sum(-1)
```

```
array([[1, 1],
       [1, 1]], dtype=object)
```

All together, the transition tensor looks as follows:

```
sp.Array(T)
```

$$\left[\begin{bmatrix} 1 & 0 \\ 1 - p_c & p_c \end{bmatrix} \begin{bmatrix} p_r & 1 - p_r \\ 0 & 1 \end{bmatrix}\right]$$

Recap | **Substituting parameter values**. In this chapter, we defined the transition and reward tensors as general `numpy` arrays with data types `object`, which we filled with symbolic expressions from `sympy`. To manipulate and substitute these expressions, we can use the `sympy.subs` method, however, not directly on the `numpy` array. Instead, we define a helper function `substitute_in_array` that takes a `numpy` array and a dictionary of substitutions and returns a new array with the substitutions applied.

```
def substitute_in_array(array, subs_dict):
    result = array.copy()
    for index,_ in np.ndenumerate(array):
        if isinstance(array[index], sp.Basic):
            result[index] = array[index].subs(subs_dict)
    return result
```

To make this work, it seems to be of critical importance that the subsitution dictionary is given as a dictionary in the form of `{<symbol_variable>: <subsitution>, ...}` and *not* as `dict(<symbol_variable>=<subsitution>, ...)`. For example,

```
substitute_in_array(T, {pc: 0.1, pr: 0.05}).astype(float)
```

```
array([[[1.  , 0.  ],
        [0.9 , 0.1 ]],

       [[0.05, 0.95],
        [0.  , 1.  ]]])
```

With the help of the `substitue_in_array` function we give the risk-reward dilemma class its environmental dynamics:

```python
def create_TransitionTensor(self):
    """Create the transition tensor."""
    return substitute_in_array(T, {pc: self.pc, pr: self.pr}).astype(float)


RiskRewardDilemma.create_TransitionTensor = create_TransitionTensor
```

10.3.2 Rewards | Short-term welfare

The rewards or welfare the agent receives represent the ecosystem services the environment provides. It is modeled by three parameters: a safe reward r_s, a risky reward $r_r > r_s$, and a degraded reward $r_d < r_s$. We assume the following default values,

```python
rs, rr, rd = sp.symbols('r_s r_r r_d')
```

We implement the rewards as a four-dimensional array or **tensor**, with dimensions $N \times Z \times M \times Z$, where $N = 1$ is the number of agents, Z is the number of states and M is the number of actions. The additional agent dimension is necessary to accommodate multi-agent environments.

```python
R = np.zeros((1,2,2,2), dtype=object)
```

The cautious action at the prosperous state guarantees the safe reward, $R(\mathsf{p}, \mathsf{c}, \mathsf{p}) = r_s$,

```python
R[0,p,c,p] = rs
```

The risky action at the prosperous leads to the risky reward if the environment does not collapse, $R(\mathsf{p}, \mathsf{r}, \mathsf{p}) = r_r$,

```python
R[0,p,r,p] = rr
```

Yet, whenever the environment enters, remains, or leaves the degraded state, it provides only the degraded reward $R(\mathsf{d}, :, :) = R(:, :, \mathsf{d}) = r_d$, where : denotes all possible states and actions.

```python
R[0,d,:,:] = R[0,:,:,d] = rd
```

Together, the reward tensor looks as follows:

```python
sp.Array(R)
```

$$\left[\left[\begin{bmatrix} r_s & r_d \\ r_r & r_d \end{bmatrix} \begin{bmatrix} r_d & r_d \\ r_d & r_d \end{bmatrix}\right]\right]$$

Again, we use the `substitute_in_array` function to give the risk-reward dilemma class its reward function:

```python
def create_RewardTensor(self):
    """Create the reward tensor."""
    return substitute_in_array(
        R, {rr: self.rr, rs: self.rs, rd: self.rd}).astype(float)


RiskRewardDilemma.create_RewardTensor = create_RewardTensor
```

10.3.3 Init method

```python
def __init__(self, CollapseProbability, RecoveryProbability,
             RiskyReward, SafeReward, DegradedReward, state=0):
    self.N = 1; self.M = 2; self.Z = 2

    self.pc = CollapseProbability
    self.pr = RecoveryProbability
    self.rr = RiskyReward
    self.rs = SafeReward
    self.rd = DegradedReward

    self.StateSet = self.obtain_StateSet()
    self.ActionSets = self.obtain_ActionSets()
    self.TransitionTensor = self.create_TransitionTensor()
    self.RewardTensor = self.create_RewardTensor()

    self.state = state
RiskRewardDilemma.__init__ = __init__
```

Basic testing

```python
env = RiskRewardDilemma(0.11, 0.4, 1.0, 0.8, 0.0)
env.TransitionTensor
```

```
array([[[1.  , 0.  ],
        [0.89, 0.11]],

       [[0.4 , 0.6 ],
        [0.  , 1.  ]]])
```

```python
env.RewardTensor
```

```
array([[[[0.8, 0. ],
         [1. , 0. ]],

        [[0. , 0. ],
         [0. , 0. ]]]])
```

```
env.step([1])
```

```
([0], array([1.]), {'state': 0})
```

10.3.4 Testing the interface

```
agent = BehaviorAgent(policy=np.ones((2,2)))
env = RiskRewardDilemma(0.2, 0.1, 1.0, 0.8, 0.0)
interface_run([agent], env, 10)
```

Obviously, this is not very insightful. We need to track the learning process.

We need to track the learning process. We can do this by storing the actions, observations, and rewards in a `pandas` DataFrame. Pandas is a powerful data manipulation library in Python that provides data structures and functions to work with structured data. We will store the data of each time step into a row and its attributes into a set of respective columns of the DataFrame.

```
def interface_run(agent, env, NrOfTimesteps):
    """Run the multi-agent environment for several time steps."""

    columns = ["action", "observation", "reward"]
    df = pd.DataFrame(index=range(NrOfTimesteps), columns=columns)

    observations = env.observe()

    for t in range(NrOfTimesteps):

        action = agent.act(observations[0])

        next_observations, rewards, info = env.step([action])

        agent.update(observations[0], action,
                     rewards[0], next_observations[0])

        df.loc[t] = (action, observations[0], rewards[0])

        observations = next_observations

    return df
```

```
df = interface_run(agent, env, 25)
df.tail()
```

	action	observation	reward
20	0	0	0.8
21	0	0	0.8
22	0	0	0.8

	action	observation	reward
23	1	0	1.0
24	1	0	0.0

```python
def plot_ActionsRewardsObservations(df):
    fig, axes = plt.subplots(3,1, figsize=(10,5))

    axes[0].plot(df.action, 'o', label='Agent 0')
    axes[0].set_ylabel('Action')
    axes[0].set_yticks([0, 1])
    axes[0].set_yticklabels([env.ActionSets[0][0], env.ActionSets[0][1]])

    axes[1].plot(df.reward, 'o', label='Agent 0');
    axes[1].set_ylabel('Reward')

    axes[2].plot(df.observation, 'o', label='Agent 0');
    axes[2].set_ylabel('Observation')
    axes[2].set_yticks([0, 1])
    axes[2].set_yticklabels([env.StateSet[0], env.StateSet[1]]);
```

```python
df = interface_run(agent, env, 25); plot_ActionsRewardsObservations(df)
```
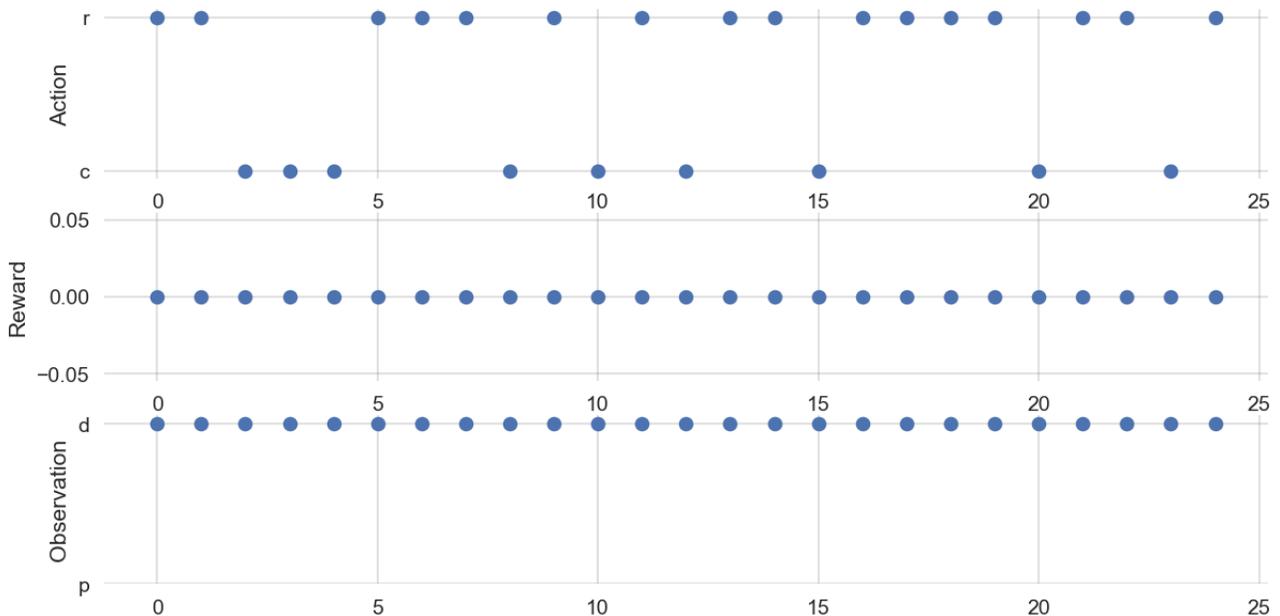

Figure 10.5: Action-Reward-Observation Dynamics

10.4 Reinforcement learning agent

Agents act (oftentimes to reach a goal). We need to specify their

- **Actions**, describing which choices are available to the agent.
- **Goal**, describing what an agent wants (in the long run). They may contain *intrinsic motivations*.

211

- **Representation**, e.g., defining upon which conditions agents select actions (e.g., *history* of past actions in multi-agent situations).
- **Value beliefs** (value functions), capturing what is *good* for the agent regarding its *goal* in the long run.
- **Behavioral rule** (policy, strategy), defining how to select actions.
- **Learning rule**, describing how value beliefs are updated in light of new information.
- (optionally), a **model** of the environment and rewards. Models are used for *planning*, i.e., deciding on a *behavioral rule* by considering possible future situations before they are actually experienced.

Agents act (oftentimes to reach a goal). As in Lecture 03.01-SequentialDecision, the agent aims to maximize the discounted sum of future rewards,

$$G_t = (1 - \gamma) \sum_{\tau=0}^{\infty} \gamma^\tau R_{t+\tau},$$

where $1 - \gamma$ is a normalizing factor and $R_{t+\tau+1}$ is the reward received at time step $t + \tau + 1$.

However, in contrast to Lecture 03.01-SequentialDecision, we assume that the agent does **not know the environment's dynamics and rewards**. Instead, the agent **learns** about the environment **while interacting** with it.

The challenge is that actions may have **delayed** and **uncertain** consequences.

Delayed consequences mean that an action may influence the environmental state, which, in turn, influences the reward the agent receives at a later time step. For example, in our risk-reward dilemma, opting for a sustainable policy may initially reduce the agent's immediate reward but ensures a comparably higher long-term welfare. **Uncertain** consequences refer to the **stochasticity** in the environmental transitions (and possibly the reward signals themselves). For example, in our risk-reward dilemma, the risky action in the prosperous state may lead to a high reward but may also cause a transition to the degraded state. Moreover, uncertainty may also refer to the fact that the environmental transition dynamics may change over time.

Thus, the agent can't just try each action in each state once and then immediately know which course of action is best. It **must learn** the best course of action **over successive trials**, each of which gives possible noisy data.

We start implement the learning class by defining `__init__` method.

```python
class Learner():
    """A simple reinforcement learning agent."""

    def __init__(self, ValueBeliefs_Qoa,
                 DiscountFactor, LearningRate, ChoiceIntensity):

        self.DiscountFactor = self.df = DiscountFactor
        self.LearningRate = self.lr = LearningRate
        self.ChoiceIntensity = self.ci = ChoiceIntensity

        self.ValueBeliefs_Qoa = ValueBeliefs_Qoa

        self.ActionIxs = range(ValueBeliefs_Qoa.shape[1])
```

The agent receives the following parameters: the initial value beliefs `ValueBeliefs_Qoa`, the discount factor `DiscountFactor`, the learning rate `LearningRate`, and the choice intensity `ChoiceIntensity`. Furthermore, we give the agent an attribute `ActionIxs` that stores the indices of the possible actions. This will be helpful when selecting actions.

10.4.1 Value beliefs

The general strategy we focus on to solve the challenges of delayed and uncertain consequences is to let the agent **learn value beliefs**. Value beliefs are the agent's estimates of the long-term value of each action a in each state s. The agent then uses these value beliefs to select actions. These estimates are also often called Q values. You may think of the *quality* of an action a in state s which tells the agents Which action to select in which state.

For example, in our risk-reward dilemma, we can represent the agent's value beliefs by

```
ValueBeliefs_Qoa = 10 * np.random.rand(2,2)
ValueBeliefs_Qoa
```

```
array([[8.83140215, 5.58254363],
       [8.59330674, 1.20765508]])
```

The challenge of uncertain consequences in then solved by an appropriate **behavioral rule** which handles the so-called **exploration-exploitation trade-off**

The challenge of delayed consequences is solved by the **learning rule**, which updates the value beliefs in light of new information using the Bellman equation, as in Lecture 03.01-SequentialDecisions.

10.4.2 Behavioral rule | Exploration-exploitation trade-off

The **exploration-exploitation trade-off** poses **a fundamental problem for decision-making under uncertainty**.

Under too much exploitation, the agent may pick an action that is not optimal, as it has not yet sufficiently explored all possible actions. It acts under the false belief that its current value beliefs are already correct or optimal. Thus, it *loses out* on possible rewards it would have gotten if it had explored more and discovered that a different course of action is better.

Under too much exploration, the agent may continue to try all actions to gain as much information about the transitions and reward distributions as possible. It is *losing out* because it never settles on the best course of action, continuing to pick all actions until the end.

What is needed is a **behavioral rule** that **balances exploitation and exploration** to explore enough to find the best option but not too much so that the best option is exploited as much as possible.

We use the so-called *softmax* function,

$$x(s,a) = \frac{\exp \beta Q(s,a)}{\sum_{b \in \mathcal{A}} \exp \beta Q(s,b)},$$

which converts any set of value beliefs into probabilities that sum to one.

The higher the relative value belief, the higher the relative probability.

```python
def obtain_softmax_probabilities(ChoiceIntensity, ValueBeliefs):
    expValueBeliefs = np.exp(ChoiceIntensity*np.array(ValueBeliefs))
    return expValueBeliefs / expValueBeliefs.sum(-1, keepdims=True)
```

The softmax function contains a parameter β, denoting the **choice intensity** (sometimes called *inverse temperature*, that determines how **exploitative** (or *greedy*) the agent is.

When $\beta = 0$, arms are chosen entirely at random with no influence of the Q values. This is super exploratory, as the agent continues to choose all arms irrespective of observed rewards.

```python
obtain_softmax_probabilities(0, ValueBeliefs_Qoa)
```

```
array([[0.5, 0.5],
       [0.5, 0.5]])
```

As β increases, there is a higher probability of picking the arm with the highest Q value. This is increasingly exploitative (or 'greedy').

```python
obtain_softmax_probabilities(1, ValueBeliefs_Qoa)
```

```
array([[9.62632074e-01, 3.73679265e-02],
       [9.99380298e-01, 6.19702179e-04]])
```

When β is very large, then only the arm that currently has the highest Q value will be chosen, even if other arms might actually be better.

```python
obtain_softmax_probabilities(50, ValueBeliefs_Qoa).round(9)
```

```
array([[1., 0.],
       [1., 0.]])
```

For example, assuming we are in state zero and using $\beta = 1$, we can select an action by

```python
obs = 0
Xoa = obtain_softmax_probabilities(1, ValueBeliefs_Qoa)
np.random.choice([0,1], p=Xoa[obs])
```

```
0
```

We summarize this logic in the agent's `act` method:

```python
def act(self, obs):
    Xoa = self.obtain_policy_Xoa()
    return np.random.choice(self.ActionIxs, p=Xoa[obs])
Learner.act = act
```

where we define the `ActionIxs` as `range(self.NrActions)` in the `__init__` method of the agent. We also define the `obtain_policy_Xoa` method in the agent class:

```
def obtain_policy_Xoa(self):
    return obtain_softmax_probabilities(self.ChoiceIntensity, self.ValueBeliefs_Qoa)
Learner.obtain_policy_Xoa = obtain_policy_Xoa
```

Testing the `act` and `obtain_policy_Xoa` methods:

```
learner = Learner(np.ones((2,2)), 0.9, 0.1, 1.0)
```

```
learner.obtain_policy_Xoa()[0]
```

```
array([0.5, 0.5])
```

Selecting action uniformly at random for 1000 times should give a mean index of approx. 0.5:

```
np.mean([learner.act(0) for _ in range(1000)])
```

```
0.479
```

10.4.3 Learning rule | Temporal-difference learning

The learning rule solves the challenge of delayed consequences. The value beliefs are updated using the Bellman equation in light of new information. As the Bellman equation describes how state(-action) values relate at different timesteps, this reinforcement learning update class is called **temporal-difference learning**.

Given an observed state s, the agent selects an action a and receives a reward r. Then, the agent updates its value beliefs (for state-action pair s-a) according to

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \left((1-\gamma)r + \gamma \sum_b x_t(s',b)Q_t(s',b) - Q_t(s,a) \right). \tag{10.1}$$

DeepDive | There is some freedom into designing the specifics of the temporal-difference update, especially regarding estimating the value of the next state or observation. The specific update used above is called **Expected SARSA**. It is beyond the scope of this course to discuss the different temporal-difference learning algorithms. The interested reader is referred to excellent material on (multi-agent) reinforcement learning, e.g., (Albrecht et al., 2024; Sutton & Barto, 2018).

The extent to which the value beliefs are updated is controlled by a second parameter, $\alpha \in (0,1)$, called the **learning rate**.

When $\alpha = 0$, there is no updating, and the reward does not affect value beliefs,

$$Q_{t+1}(s,a) = Q_t(s,a) =: \text{old estimate}.$$

The value belief update always remains the *old estimate* of the value beliefs.

When $\alpha = 1$, the value belief for the state-action pair (s,a) becomes a discount-factor weighted average between the current reward r and the expected value of the next state $\sum_b x_t(s',b)Q_t(s',b)$,

$$Q_{t+1}(s, a) = (1 - \gamma)r + \gamma \sum_b x_t(s', b)Q_t(s', b) =: \text{new estimate.}$$

The value belief update is entirely determined by the *new estimate* of the value beliefs, which is the current reward received r plus the discount factor γ multiplied by the expected value of the following state $\sum_b x_t(s', b)Q_t(s', b)$, and adequately normalized with the prefactor $(1 - \gamma)$.

When $0 < \alpha < 1$, the new value belief for the rewarded arm is a *weighted average* between old value belief and new reward information,

$$Q_{t+1}(s, a) = (1 - \alpha) \text{ old estimate} \qquad + \alpha \text{ new estimate} \tag{10.2}$$

$$= (1 - \alpha) \, Q_t(s, a) \qquad + \alpha \left((1 - \gamma)r + \gamma \sum_b x_t(s', b)Q_t(s', b) \right). \tag{10.3}$$

Once more, we face a trade-off. Clearly, setting $\alpha = 0$ is ineffective since the agent does not acquire knowledge. Yet, if α is excessively high; the agent tends to *forget* previously learned state-action information.

Temporal-difference reward-prediction error. Another way to think about the update equation (Equation 10.1) is as follows: The value beliefs are updated by the *temporal-difference reward-prediction error* (TDRP error) times the learning rate. The TDRP error equals the difference between the *new estimate* and the *old estimate* of the value beliefs. If the TDRP error is zero, the agent correctly predicted the next reward, and thus, no further adjustments in the value beliefs are necessary.

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \qquad\qquad \text{TDRP-error,} \tag{10.4}$$

$$= Q_t(s, a) + \alpha \Big(\qquad\quad \text{new estimate} \qquad\quad - \text{old estimate} \Big), \tag{10.5}$$

$$= Q_t(s, a) + \alpha \Big((1 - \gamma)r + \gamma \sum_b x_t(s', b)Q_t(s', b) - \quad Q_t(s, a) \quad \Big) \tag{10.6}$$

DeepDive | The exact terminology *temporal-difference reward-prediction error* is used rather rarely. We use it here to express the interdisciplinary nature of temporal-difference reward-prediction learning. In machine learning, the term *temporal-difference error* is more common. It describes the difference between the predicted and the observed reward. In psychology and neuroscience, the term *reward-prediction error* is used in the context of the brain's dopamine system, where it is thought to signal the difference between the expected and the observed reward. The term *temporal-difference reward-prediction error* combines both terms, expressing the idea that the agent learns by predicting future rewards.

Executing the value belief update in Python may look like

```python
def update(self,
           obs: int,
           action: int,
           reward: float,
           next_obs: int,
           ):
    """Updates the value beliefs / Q-value of an action."""

    temporal_difference =self.obtain_temporal_difference(
```

```
        obs, action, reward, next_obs)

    self.ValueBeliefs_Qoa[obs, action] = (
        self.ValueBeliefs_Qoa[obs, action] + self.LearningRate * temporal_difference
    )

Learner.update = update
```

We program the `update` method highly modular. It calls the `obtain_temporal_difference` method to compute the temporal-difference error and then updates the value beliefs accordingly. The `obtain_temporal_difference` method is defined as follows:

```
def obtain_temporal_difference(self,
                                obs: int,
                                action: int,
                                reward: float,
                                next_obs: int,
                                ):
    """Compute temporal-difference eorror"""
    next_Qoa = self.obtain_nextQoa(next_obs)
    new_estimate = (1-self.DiscountFactor) * reward + self.DiscountFactor * next_Qoa
    old_estimate = self.ValueBeliefs_Qoa[obs][action]
    return new_estimate - old_estimate
Learner.obtain_temporal_difference = obtain_temporal_difference
```

In here, we call the `obtain_nextQoa` method to compute the expected value of the next state. The `obtain_nextQoa` method is defined as follows:

```
def obtain_nextQoa(self, next_obs: int):
    policy_Xoa = self.obtain_policy_Xoa()
    return np.sum(policy_Xoa[next_obs] * self.ValueBeliefs_Qoa[next_obs])
Learner.obtain_nextQoa = obtain_nextQoa
```

Testing the update method: First, let's assume the agent does not care about future rewards at all and has a discount factor of zero

```
learner = Learner(ValueBeliefs_Qoa = np.ones((2,2)),
                  DiscountFactor = 0.0,
                  LearningRate = 0.1,
                  ChoiceIntensity = 1.0)
```

```
learner.ValueBeliefs_Qoa
```

```
array([[1., 1.],
       [1., 1.]])
```

Let's assume the agent selected the action with index `0` after observing the state with index `0`, received a reward of zero, and observed the next state with index `1`.

```
learner.update(obs=0, action=0, reward=0, next_obs=1)
learner.ValueBeliefs_Qoa.round(4)
```

```
array([[0.9, 1. ],
       [1. , 1. ]])
```

The value belief for the action 0 in state 0 is updated exactly as a learning rate weighted average: $\alpha \cdot$ new estimate $+ (1-\alpha) \cdot$ old estimate $= \alpha 0 + (1-\alpha)1 = 0.1 \cdot 0 + 0.9 \cdot 1$.

Repeating this update a hundred more time steps updates the value beliefs for the action 0 in state 0 to the expected value of zero.

```
for _ in range(100): learner.update(obs=0, action=0, reward=0, next_obs=1)
learner.ValueBeliefs_Qoa.round(4)
```

```
array([[0., 1.],
       [1., 1.]])
```

Now, we repeat that test, but with an agent with a discount factor of $\gamma = 0.8$.

```
learner = Learner(ValueBeliefs_Qoa = 1*np.ones((2,2)),
                  DiscountFactor = 0.8,
                  LearningRate = 0.1,
                  ChoiceIntensity = 1.0)
```

```
learner.ValueBeliefs_Qoa
```

```
array([[1., 1.],
       [1., 1.]])
```

```
for _ in range(100): learner.update(obs=0, action=0, reward=0, next_obs=1)
learner.ValueBeliefs_Qoa.round(4)
```

```
array([[0.8, 1. ],
       [1. , 1. ]])
```

Now, the value belief for the action 0 in state 0 is updated to 0.8. Can you explain why?

We have convince ourselves that the learner's **update** methods works as we expect.

Now, we are ready to let it learn in the risk-reward dilemma environment.

10.4.4 Testing the interface

```
learner = Learner(ValueBeliefs_Qoa = np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.1,
                  ChoiceIntensity = 1.0)
print(learner.obtain_policy_Xoa())

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

print(" - - - - ")
df = interface_run(learner, env, 10000)
print(" - - - - ")

print(learner.obtain_policy_Xoa())
```

```
[[0.5 0.5]
 [0.5 0.5]]
 - - - -
 - - - -
[[0.5091271  0.4908729 ]
 [0.50860671 0.49139329]]
```

The learning agent's policy changed. However, not too much. We know from Lecture 03.01-SequentialDecisions that the agent should learn to prefer the cautious action in both states under these parameter settings.

Try re-executing the above cell while make some changes to the parameters. Can you get an intuition what is important for a successful learning process?

The process of finding the right parameters for the agent is called **hyperparameter tuning**. It is a crucial step in machine learning and often requires a lot of trial and error.

From a modeling point of view, we aim to go beyond finding the *right* parameter. We aim to **understand** how the parameters influence the learning process.

Comparing the initial with the final policy is not the best way to facilitate both aims. We need a more refined way to keep track of the learning process.

10.5 Investigating the learning process

To keep track of the learning process, we store the value beliefs and the policy in a `pandas` DataFrame. Aditionally, we also record the learning rate and the choice intensity.

```
def interface_run(agent, env, NrOfTimesteps):
    """Run the multi-agent environment for several time steps."""

    columns = ["action", "observation", "reward", "beliefs", "policy",
               "ChoiceIntensity", "LearningRate"]
    df = pd.DataFrame(index=range(NrOfTimesteps), columns=columns)

    observations = env.observe()
```

```
    for t in range(NrOfTimesteps):

        action = agent.act(observations[0])

        next_observations, rewards, info = env.step([action])

        agent.update(observations[0], action,
                     rewards[0], next_observations[0])

        df.loc[t] = (action, next_observations[0], rewards[0],
                     deepcopy(agent.ValueBeliefs_Qoa),
                     deepcopy(agent.obtain_policy_Xoa()),
                     deepcopy(agent.ChoiceIntensity),
                     deepcopy(agent.LearningRate))

        observations = next_observations

    return df
```

```
learner = Learner(ValueBeliefs_Qoa = 0*np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.05,
                  ChoiceIntensity = 8.0)

print(learner.obtain_policy_Xoa())

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

df = interface_run(learner, env, 10000)
```

```
[[0.5 0.5]
 [0.5 0.5]]
```

As we stored the value beliefs and policies as two-dimensional `numpy` arrays, we convert them into three-dimensional `numpy` with time running on the first dimension:

```
beliefs_Qtoa = np.array(df.beliefs.values.tolist())
policy_Xtoa = np.array(df.policy.values.tolist())
beliefs_Qtoa.shape, policy_Xtoa.shape
```

```
((10000, 2, 2), (10000, 2, 2))
```

We include these conversions into a plotting function that visualizes the learning process

```
def plot_learning_process(df, plot_varying_parameters=False):
    beliefs_Qtoa = np.array(df.beliefs.values.tolist())
    policy_Xtoa = np.array(df.policy.values.tolist())
    beliefs_Qtoa.shape, policy_Xtoa.shape
```

```
    fig = plt.figure(figsize=(14,6))

    ax0 = fig.add_subplot(311)
    ax0.plot(beliefs_Qtoa[:,p,c], label='Q(p,c)', color='blue', lw=2)
    ax0.plot(beliefs_Qtoa[:,p,r], label='Q(p,r)', color='red', lw=2)
    ax0.plot(beliefs_Qtoa[:,d,c], label='Q(d,c)', color='darkblue', ls='--')
    ax0.plot(beliefs_Qtoa[:,d,r], label='Q(d,r)', color='darkred', ls='--')
    ax0.set_ylabel('Value beliefs');
    ax0.legend(loc='center right'); ax0.set_xlim(-10, len(df)*1.1)

    ax1 = fig.add_subplot(312, sharex=ax0)
    ax1.plot(policy_Xtoa[:,p,c], label='X(p,c)', color='blue', lw=2)
    ax1.plot(policy_Xtoa[:,p,r], label='X(p,r)', color='red', lw=2)
    ax1.plot(policy_Xtoa[:,d,c], label='X(d,c)', color='darkblue', ls='--')
    ax1.plot(policy_Xtoa[:,d,r], label='X(d,r)', color='darkred', ls='--')
    ax1.set_ylabel('Policy'); ax1.set_xlabel('Time steps')
    ax1.legend(loc='center right')

    if plot_varying_parameters:
        ax2 = fig.add_subplot(615, sharex=ax0)
        ax2.plot(df.LearningRate, label='LearningRate', color='k')
        ax2.set_ylabel("Learning\nRate")

        ax3 = fig.add_subplot(616, sharex=ax0)
        ax3.plot(df.ChoiceIntensity, label='ChoiceIntensity', color='k')
        ax3.set_ylabel("Choice\nIntensity"), ax3.set_xlabel('Time steps')

    # plt.tight_layout()
    plt.subplots_adjust(hspace=0.35)

    # plt.legend()
    ax0.set_ylim(0, 1);
```

10.5.1 To little exploitation | To much exploration

```
learner = Learner(ValueBeliefs_Qoa = 0*np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.1,
                  ChoiceIntensity = 1.0)

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

df = interface_run(learner, env, 10000)
```

```
plot_learning_process(df)
```
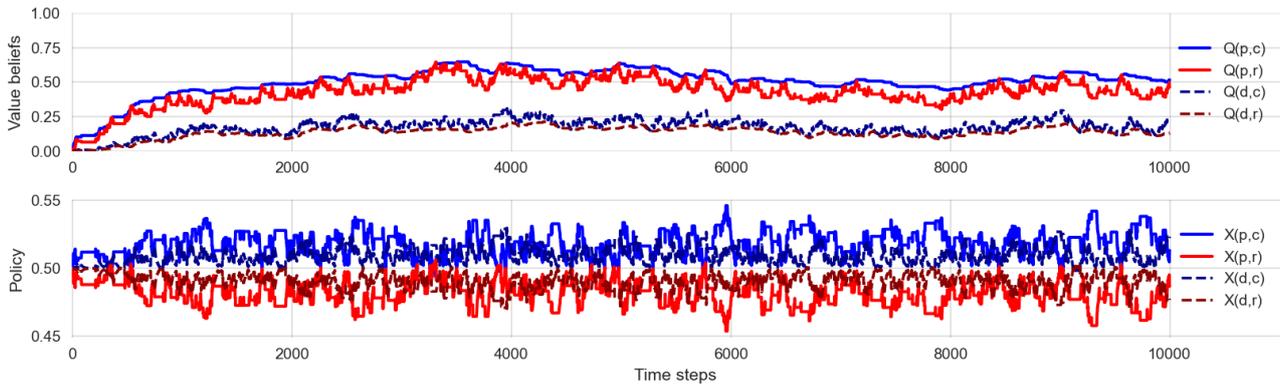
Figure 10.6: Learning the risk-reward dilemma with to much exploration.

The order of the **value beliefs** seems roughly **consistent with** the **optimal policy**, that prefers the cautious action over the risky on in both states. However, the **agents policy is fluctuating around**. The action choice probabilities are fluctuating around their uniformly random value of 0.5. This is a sign that the agent explores too much and exploits too little.

10.5.2 To much exploitation | To little exploration

```
learner = Learner(ValueBeliefs_Qoa = 0*np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.1,
                  ChoiceIntensity = 100.0)

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

df = interface_run(learner, env, 10000)
```

```
plot_learning_process(df)
```
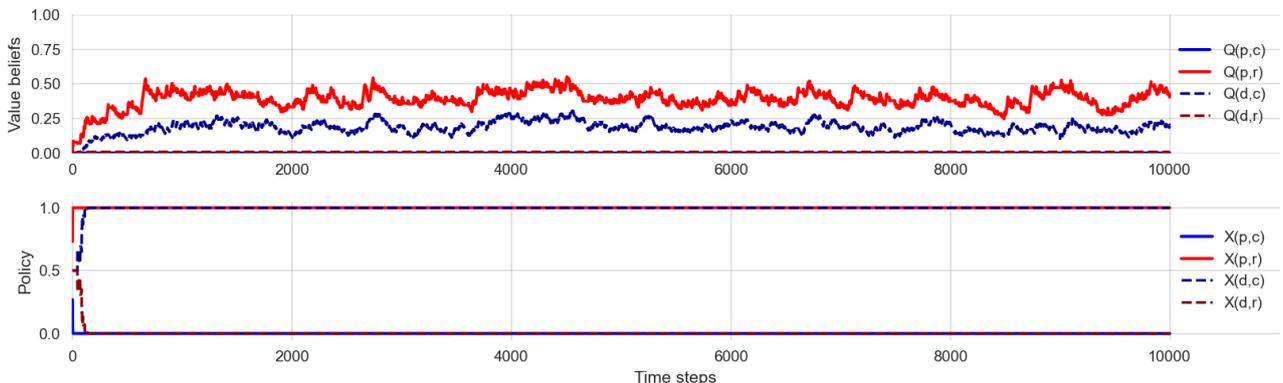

Figure 10.7: Learning the risk-reward dilemma with to much exploitation.

Increasing the choice intensity leads to a more exploitative policy. However, what the agent learns depends on the stochasticity of the learning process. **Try to convince yourself of that fact by re-executing the above cell multiple times.**

In other words, the agent may not learn the optimal policy. To little exploration harms the learning.

10.5.3 Decaying exploration | Increasing exploitation

To mitigate the negative effects of too much exploration towards the end of the learning process and the negative effects of too much exploitation towards the beginning of the learning process, we could let the choice intensity increase over time. This is called **decaying exploration**.

We implement this idea by creating a new agent class `AdjustingLearner` that inherits from the `Learner` class. This shows the power of object-oriented programming. We overwrite the `update` method to include an increasing choice intensity. We also make the learning rate decay over time.

```python
class AdjustingLearner(Learner):

    def __init__(self,
                 ValueBeliefs_Qoa,
                 DiscountFactor,
                 LearningRate, MinLearningRate, LearningRateDecayFactor,
                 ChoiceIntensity, MaxChoiceIntensity, ChoiceIntensityGrowthFactor,
                 ):

        self.DiscountFactor = DiscountFactor

        self.LearningRate = LearningRate
        self.MinLearningRate = MinLearningRate
        self.LearningRateDecayFactor = LearningRateDecayFactor

        self.ChoiceIntensity = ChoiceIntensity
        self.MaxChoiceIntensity = MaxChoiceIntensity
        self.ChoiceIntensityGrowthFactor = ChoiceIntensityGrowthFactor

        self.ValueBeliefs_Qoa = ValueBeliefs_Qoa
        self.ActionIxs = range(ValueBeliefs_Qoa.shape[1])
```

```python
def update(self,
           obs: int,
           action: int,
           reward: float,
           next_obs: int,
           ):
    """Updates the value beliefs / Q-value of an action."""

    temporal_difference =self.obtain_temporal_difference(
        obs, action, reward, next_obs)

    self.ValueBeliefs_Qoa[obs, action] = (
        self.ValueBeliefs_Qoa[obs, action] + self.LearningRate * temporal_difference
    )

    self.LearningRate = max(self.MinLearningRate,
                            self.LearningRate * self.LearningRateDecayFactor)
    self.ChoiceIntensity = min(self.MaxChoiceIntensity,
```

```
                                self.ChoiceIntensity *
                            ↪  self.ChoiceIntensityGrowthFactor)


AdjustingLearner.update = update
```

Now, we are ready to perform a new learning simulation.

```
learner = AdjustingLearner(ValueBeliefs_Qoa = 0*np.ones((2,2)),
                           DiscountFactor = 0.9,
                           LearningRate = 0.1, MinLearningRate = 0.01,
 ↪  LearningRateDecayFactor = 0.999,
                           ChoiceIntensity = 1.0, MaxChoiceIntensity = 50.0,
 ↪  ChoiceIntensityGrowthFactor = 1.001)

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

df = interface_run(learner, env, 10000)
```

```
plot_learning_process(df, plot_varying_parameters=True)
```
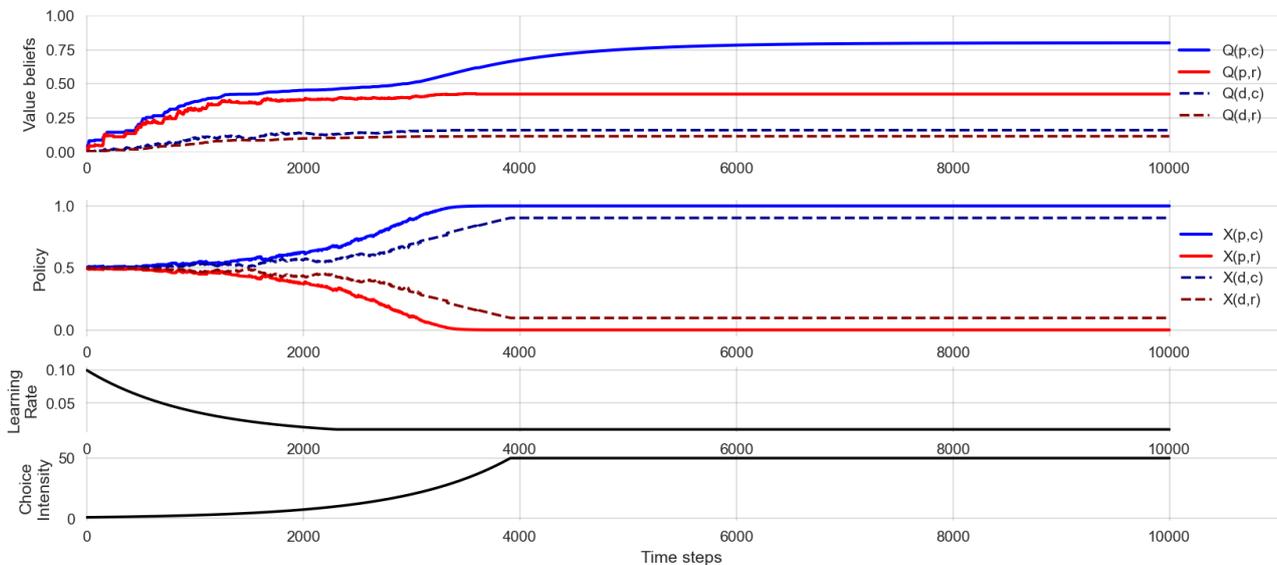

Figure 10.8: Learning the risk-reward dilemma with an increasing choice intensity.

We find the `AdjustingLearner` is able to consitently learn the optimal policy. **Try to convince yourself that this is true by re-executing the simulation above multiple times.** We also find that it learns the optimal policy in approx. less than 4000 time steps. **How does the learning process depend on the additional parameters?**

Obviously, the `AdjustingLearner` is a more complex agent than the `Learner`. There is also a prominent *trick* to give our simpler `Learner` agent an initial exploration bonus.

10.5.4 Initial exploration bonus

We can give the agent an initial exploration bonus by setting the initial value beliefs to a high value. This is called **optimistic initialization**.

```
learner = Learner(ValueBeliefs_Qoa = 8*np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.1,
                  ChoiceIntensity = 60.0)

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

df = interface_run(learner, env, 10000)
```

```
plot_learning_process(df)
```
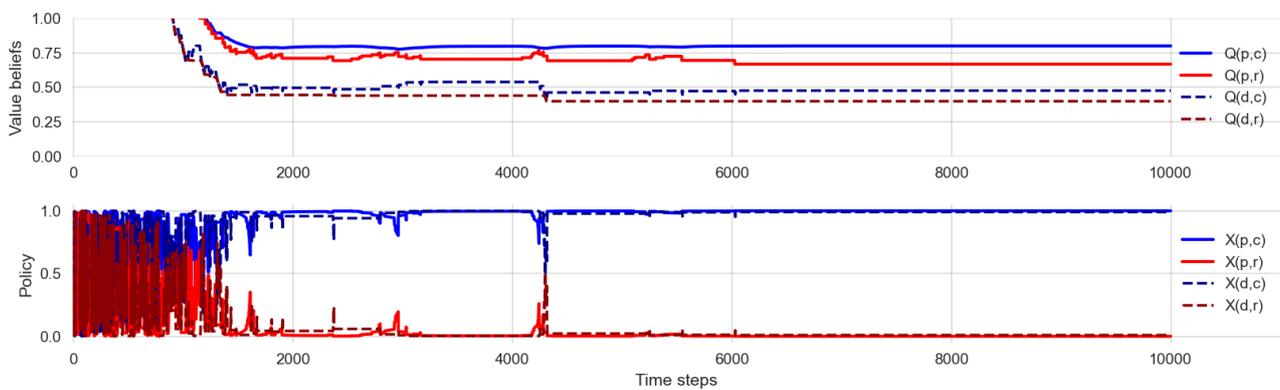

Figure 10.9: Learning the risk-reward dilemma with an initial exploration bonus.

So far, we have investigated the learning process in a single environment.

Next, we will explore learning in normal-form games to illustrate the modularity of the agent-environment interface, utilizing the same learning agents as previously discussed.

10.6 Multi-agent environments | Games

10.6.1 Interface

First, we adjust the `interface_run` function to work with the multi-agent environment.

We can make the *columns* of a dataframe adaptive to the number of agents.

```
NrAgents = 2; NrOfTimesteps = 4

def create_dataframe(NrAgents, NrOfTimesteps):
    columns = list(np.array([(f"action{i}", f"observation{i}", f"reward{i}",
                             f"beliefs{i}", f"policy{i}",
                             f"ChoiceIntensity{i}", f"LearningRate{i}")
```

```
                            for i in range(NrAgents)]).flatten())

    return pd.DataFrame(index=range(NrOfTimesteps), columns=columns)

create_dataframe(NrAgents, NrOfTimesteps)
```

	action0	observation0	reward0	beliefs0	policy0	ChoiceIntensity0	LearningRate0	action1	observat
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

We also write a function to populate the DataFrame with the values of the learning process.

```
def fill_dataframe(actions, next_observations, rewards, agents):
    data = []
    for i in range(len(agents)):
        data += [actions[i], next_observations[i], rewards[i],
                 deepcopy(agents[i].ValueBeliefs_Qoa),
                 deepcopy(agents[i].obtain_policy_Xoa()),
                 deepcopy(agents[i].ChoiceIntensity),
                 deepcopy(agents[i].LearningRate)]
    return data
```

Adjusting the `interface_run` function to work with the multi-agent environment yields a clean and readable implementation.

```
def interface_run(agents, env, NrOfTimesteps):
    """Run the multi-agent environment for several time steps."""
    df = create_dataframe(len(agents), NrOfTimesteps)

    observations = env.observe()

    for t in range(NrOfTimesteps):

        actions = [agent.act(observations[i])
            for i, agent in enumerate(agents)]

        next_observations, rewards, info = env.step(actions)

        for i, agent in enumerate(agents):
            agent.update(observations[i], actions[i], rewards[i],
↪   next_observations[i])

        df.loc[t] = fill_dataframe(actions, next_observations, rewards, agents)

        observations = next_observations

    return df
```

Testing whether it works in our previous environment, the risk-reward dilemma looks promising.

```
learner = Learner(ValueBeliefs_Qoa = 8*np.ones((2,2)),
                  DiscountFactor = 0.9,
                  LearningRate = 0.1,
                  ChoiceIntensity = 60.0)

env = RiskRewardDilemma(CollapseProbability=0.2, RecoveryProbability=0.1,
                        SafeReward=0.8, RiskyReward=1.0, DegradedReward=0.0)

# Note: we need to pass the learner as a list of agents
df = interface_run([learner], env, 4)
```

```
df
```

	action0	observation0	reward0	beliefs0	policy0
0	1	0	1.0	[[8.0, 7.93], [8.0, 8.0]]	[[0.9852259683067277,
1	0	0	0.8	[[7.927906923600332, 7.93], [8.0, 8.0]]	[[0.46864505269071266,
2	0	0	0.8	[[7.8567279493493345, 7.93], [8.0, 8.0]]	[[0.012172569010807209
3	1	0	1.0	[[7.8567279493493345, 7.86061972818162], [8.0,...	[[0.4418871301229423,

The real test, however, comes with a true multi-agent environment.

10.6.2 Social dilemmas environment

Let's extend our treatment of reinforcement learning to **multiple agents**. From the perspective of each individual agent, other agents make the environment **non-stationary**. This can complicate reinforcement learning significantly.

We here focus on normal-form games and use the generic model of a social dilemma, introduced in Lecture 03.02-StrategicInteractions.

	Abate	Pollute
Abate	$1 \mid 1$	$-1 - F \mid +1 + G$
Pollute	$+1 + G \mid -1 - F$	$-1 \mid -1$

Depending on whether the greed G and fear F are positive or negative, we can distinguish four types of games Figure 10.10.

In Figure 10.10, the payoff values are ordinal, meaning that only their order, $3 > 2 > 1 > 0$, is considered of relevance.

We also implement it as a class using the same interface as before and letting it inherit from our base environment.

```
class SocialDilemma(Environment):
    """A simple social dilemma environment."""

    def obtain_StateSet(self):
        return ['.']  # a dummy state
```

227

Divergence
Chicken | Hawk-Dove | Snowdrift

	A	P
A	2 | 2	[1 | 3]
P	[3 | 1]	0 | 0

Greed

Tragedy
Prisoner's Dilemma

	A	P
A	2 | 2	0 | 3
P	3 | 0	[1 | 1]

0 — Fear →

Comedy
Harmony

	A	P
A	[3 | 3]	1 | 2
P	2 | 1	0 | 0

Coordination
Stag-Hunt

	A	P
A	[3 | 3]	0 | 2
P	2 | 0	[1 | 1]
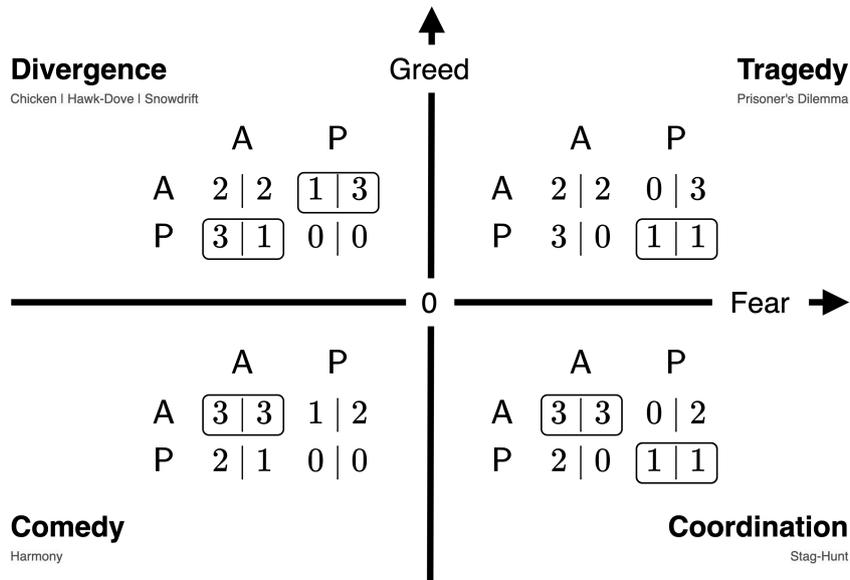
Figure 10.10: Dimensions of a social dilemma with ordinal payoffs and Nash equilibira shown in boxes.

```python
def obtain_ActionSets(self):
    # abate, pollute for two agents
    return [['a', 'p'], ['a', 'p']]
```

Due to the absence of environmental state transitions, the environment consistently exists in a single, effective dummy state. Consequently, the transition tensor is simplified significantly.

```python
def create_TransitionTensor(self):
    """Create the transition tensor."""
    return np.ones((self.Z, self.M, self.M, self.Z))

SocialDilemma.create_TransitionTensor = create_TransitionTensor
```

The reward tensor is slighlty more complicated. The two defining parameters of the social dilemma environemtn are the greed G and the fear F.

```python
F, G = sp.symbols('F G')
```

We represent rewards using a five-dimensional tensor with dimensions $N \times Z \times M \times M \times Z$. Here, N denotes the number of agents, $Z = 1$ signifies the state count, and M indicates the number of actions. A uni-dimensional state dimension is essential for accommodating multi-state environments.

```python
R = np.zeros((2,1,2,2,1), dtype=object)
```

Helper variable for the indices facilitate the construction of the reward tensor.

```python
a = SocialDilemma().obtain_ActionSets()[0].index('a')
p = SocialDilemma().obtain_ActionSets()[0].index('p')
a,p
```

228

```
(0, 1)
```

Mutual abatement yields a reward of one for both agents.

```
R[0, 0, a, a, 0] = R[1, 0, a, a, 0] = 1
```

Mutual pollution yields a reward of minus one for both agents.

```
R[0, 0, p, p, 0] = R[1, 0, p, p, 0] = -1
```

Pollution by one agent and abatement by the other agent yields a reward of one plus the greed for the polluting agent and minus the fear for the abating agent.

```
R[0, 0, p, a, 0] = R[1, 0, a, p, 0] = 1 + G
R[0, 0, a, p, 0] = R[1, 0, p, a, 0] = -1 - F
```

In sum, the reward tensor for agent zero reads,

```
sp.Array(R[0,0,:,:,0])
```

$$\begin{bmatrix} 1 & -F-1 \\ G+1 & -1 \end{bmatrix}$$

and for agent one,

```
sp.Array(R[1,0,:,:,0])
```

$$\begin{bmatrix} 1 & G+1 \\ -F-1 & -1 \end{bmatrix}$$

```
def create_RewardTensor(self):
    """Create the reward tensor."""
    return substitute_in_array(
        R, {F: self.Fear, G: self.Greed}).astype(float)

SocialDilemma.create_RewardTensor = create_RewardTensor
```

The two defining parameters of the social dilemma are the greed G and the fear F.

```
def __init__(self, Greed, Fear):
    self.N = 2; self.M = 2; self.Z = 1

    self.Greed = Greed
    self.Fear = Fear

    self.StateSet = self.obtain_StateSet()
    self.ActionSets = self.obtain_ActionSets()
    self.TransitionTensor = self.create_TransitionTensor()
    self.RewardTensor = self.create_RewardTensor()

    self.state = 0
SocialDilemma.__init__ = __init__
```

10.6.3 Testing the implementation:

```
env = SocialDilemma(Fear=0.65, Greed=0.75)
```

Mutual cooperation by two abating agents:

```
env.step([a,a])
```

```
([0, 0], array([1., 1.]), {'state': 0})
```

Mutual defection by two polluting agents:

```
env.step([p,p])
```

```
([0, 0], array([-1., -1.]), {'state': 0})
```

Different actions:

```
env.step([a,p])
```

```
([0, 0], array([-1.65,  1.75]), {'state': 0})
```

```
env.step([p,a])
```

```
([0, 0], array([ 1.75, -1.65]), {'state': 0})
```

Testing whether the implementation works,

```
learner1 = Learner(ValueBeliefs_Qoa = 8*np.ones((1,2)),
                   DiscountFactor = 0.9,
                   LearningRate = 0.1,
                   ChoiceIntensity = 60.0)
learner2 = deepcopy(learner1)

env = SocialDilemma(Fear=1, Greed=2)

df = interface_run([learner1, learner2], env, 4)
df
```

	action0	observation0	reward0	beliefs0	policy0
0	1	0	3.0	[[8.0, 7.95]]	[[0.9525741268224331, 0.04742
1	0	0	-2.0	[[7.899786583570701, 7.95]]	[[0.0468507276383191, 0.95314
2	1	0	-1.0	[[7.899786583570701, 7.860288271841277]]	[[0.9145029408174301, 0.08549
3	0	0	1.0	[[7.830484788680395, 7.860288271841277]]	[[0.14329244697889626, 0.8567

throws no erros.

10.6.4 Transient cooperation

In this section, we show that reinforcement learning agents can learn to cooperate in a tragedy social dilemma environment. However, this cooperation is not stable. It is only a transient phenomenon (Goll et al., 2024).

We use a social dilemma with fear $F = 1$ and greed $G = 2$.

```
env = SocialDilemma(Fear=1, Greed=2)
```

We want to give the agents an inital boost to cooperate or abate. Thus, we give them an inital higher value belief for abate than pollute.

```
learner1 = Learner(ValueBeliefs_Qoa = np.array([[0.5, -0.5]]),
                   DiscountFactor = 0.9,
                   LearningRate = 0.01,
                   ChoiceIntensity = 5.0)
```

We assume the second agent to be identical to the first one.

```
learner2 = deepcopy(learner1)
```

```
np.random.seed(42)
df = interface_run([learner1, learner2], env, 20000)
```

```
def plot_TwoAgentBeliefsPolicies(df):
    beliefs0_Qtoa = np.array(df.beliefs0.values.tolist())
    policy0_Xtoa = np.array(df.policy0.values.tolist())
    beliefs1_Qtoa = np.array(df.beliefs1.values.tolist())
    policy1_Xtoa = np.array(df.policy1.values.tolist())

    fig = plt.figure(figsize=(14,6))

    ax0 = fig.add_subplot(311); ax0.set_ylabel('Value beliefs');
    ax0.plot(beliefs0_Qtoa[:,0,a], label='Q1(a)', color='blue',lw=2)
    ax0.plot(beliefs0_Qtoa[:,0,p], label='Q1(p)', color='red', lw=2)
    ax0.plot(beliefs1_Qtoa[:,0,a], label='Q2(a)', color='darkblue', ls='--')
    ax0.plot(beliefs1_Qtoa[:,0,p], label='Q2(p)', color='darkred', ls='--')
    ax0.legend(loc='center right'); ax0.set_xlim(-10, len(df)*1.1)

    ax1 = fig.add_subplot(312, sharex=ax0); ax1.set_ylabel('Policy')
    ax1.plot(policy0_Xtoa[:,0,a], label='X1(a)', color='blue', lw=2)
    ax1.plot(policy0_Xtoa[:,0,p], label='X1(p)', color='red', lw=2)
    ax1.plot(policy1_Xtoa[:,0,a], label='X2(a)', color='darkblue', ls='--')
    ax1.plot(policy1_Xtoa[:,0,p], label='X2(p)', color='darkred', ls='--')
    ax1.set_xlabel('Time steps'); ax1.legend(loc='center right')
```

```
plot_TwoAgentBeliefsPolicies(df)
```
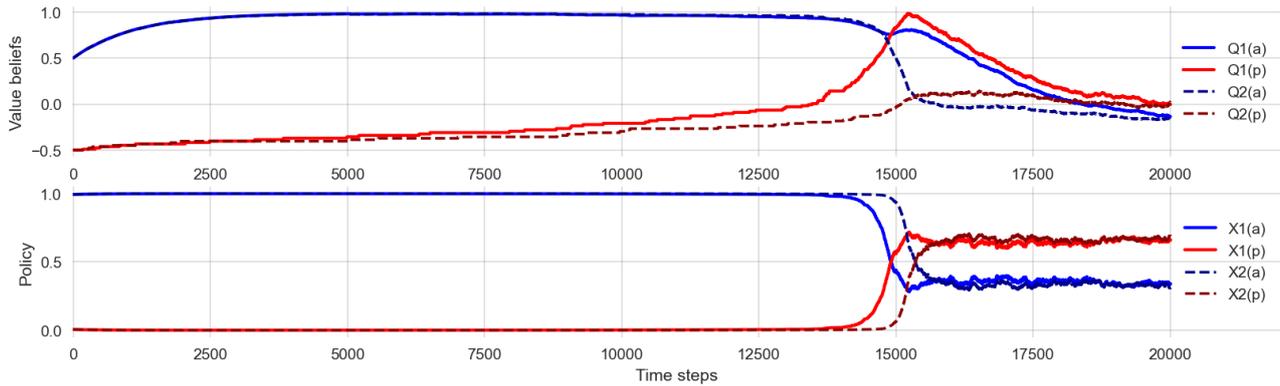

Figure 10.11: Transient cooperation in a tragedy social dilemma.

A propensity for cooperation coupled with excessive exploitation initially leads to transient cooperation within the stochastic learning dynamics. If one is unaware of this phenomenon, it may seem that the issue of cooperation in social dilemmas is resolved. However, as learning continues, this cooperation phase diminishes, resulting in increased defection (pollution) over cooperation (abate). During this phase, agents explore excessively, necessitating a higher choice intensity to establish more deterministic policies that align with the Nash equilibrium of full defection.

The timeing of when the breakdown of cooperation happens is stochastic. **Re-run the simulation above with differnt random seeds to see that this is true**. Beaware, that you must re-initalize the learners to begin from scratch.

10.7 Learning goals revisited

- Reinforcement learning is **valuable in models of human-environment interactions** as a principled take to integrate individual cognition in dynamic environments and emerging collective behavior
- We implemented the different elements of the multi-agent environment framework (interface, environment, agents).

  - We implemented and applied a **basic temporal-different learning** agent.
  - We implemented and applied the **risk-reward dilemma** (Lecture 03.01) and **social dilemma** (Lecture 03.02)
  - We visualized the learning process

- We introduced and studied the **exploration-exploitation trade-off**, a general challenge for decision-making under uncertainty.
- We made all of this possible by using the Python library `pandas` to manage data and refining our skills in object-oriented programming.

10.7.1 Key advantages of an RL framework

- Cognitive mechanisms are more integrated / less fragmented than behavioral theories
- Cognitive mechanisms (as in RL) are more formalized than behavioral theories

232

- The RL frame provides a natural dynamic extension to some economic equilibrium decision models.
- The Rl frame allows for the study of behavior changes (e.g., after experimental policy interventions or environmental catastrophes)

10.7.2 Challenges

- The **learning is inefficient**. The agents require many interactions with the environment to learn what to do as they do not learn any model of the environment. This is a cognitive wasteful process.
- Dealing with **rare states/events** is **challenging** when learning from only experience. Even more sample interactions are required to have enough experience of the raw events.
- The **stochasticity and hyperparameter tuning** make it a **cumbersome modeling tool**. Both elements are invaluable for RL as an optimization method. For RL as a model of the cognitive processes underpinning human behavior, stochasticity and hyperparameter tuning complicate the modeling process considerably. They make studying the learning dynamics more difficult than necessary.

Up next: Deterministic approximations

11 Learning dynamics

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

11.1 Motivation

Modeling model-based reinforcement learning agents

This chapter introduces **collective reinforcement learning dynamics** - treating the multi-agent reinforcement learning process as a non-linear dynamic system.

11.1.1 Recap | Reinforcement learning

In chapter 04.02-IndividualLearning, we introduced the basics of the temporal-difference reward-prediction reinforcement learning process. In essence, learning means **updating the quality estimates**, $Q_t^i(s, a)$, with the current reward-prediction error, $\delta_t^i(s, a)$, after selection action a_t in state s_t according to

$$Q_{t+1}^i(s_t, a_t) = Q_t^i(s_t, a_t) + \alpha^i \delta_t^i(s_t, a_t), \tag{11.1}$$

where $\alpha^i \in (0, 1)$ is the learning rate of agent i, which regulates how much new information the agent uses for the update.

The reward-prediction error, $\delta_t^i(s_t, a_t)$, equals the difference of the **new quality estimate**, $(1-\gamma^i)r_t^i + \gamma^i \mathcal{Q}_n^i(s_{t+1})$, and the **current quality estimate**, $\mathcal{Q}_c^i(s_t)$,

$$\delta_t^i(s_t, a_t) = (1 - \gamma^i)r_t^i + \gamma^i \mathcal{Q}_n^i(s_{t+1}, a_{t+1}) - \mathcal{Q}_c^i(s_t, a_t), \tag{11.2}$$

where the $\mathcal{Q}_n^i$ represents the quality estimate of the *next* state and $\mathcal{Q}_c^i$ represents the quality estimate of the *current* state. Depending on how we choose, $\mathcal{Q}_n^i$, and $\mathcal{Q}_c^i$, we recover various well-known temporal-difference reinforcement learning update schemes (Barfuss et al., 2019).

For example, we covered the *Expected SARSA* update with $\mathcal{Q}_n^i(s_{t+1}, a_{t+1}) = \mathcal{Q}_n^i(s_{t+1}) = \sum_b x_t^i(s_{t+1}, b)Q_t^i(s_{t+1}, b)$, and $\mathcal{Q}_c^i = Q_t^i$. The temporal-difference reward-prediction error then reads,

$$\delta_t^i(s_t, a_t) = (1 - \gamma^i)r_t^i + \gamma^i \sum_b x_t^i(s_{t+1}, b)Q_t^i(s_{t+1}, b) - Q_t^i(s_t, a_t).$$

11.1.2 Modeling challenges of reinforcement learning

Classic reinforcement learning processes are highly *stochastic* since, generally, all agent strategies $x^i(s,a)$, and the environments transition function $T(s,a,s')$ are probability distributions. This stochasticity induces some challenges for using reinforcement learning as a modeling tool in complex human-environment systems:

1) **Sample inefficiency.** The agents need many samples to learn something, as they immediately forget a sample experience after a value-belief update.
2) **Computationally intense.** Learning simulations are computationally intense since one requires many simulations to make sense of the noise, and each takes a long time to address the sample inefficiency.
3) **Rare events.** Due to the stochasticity, dealing with rare events is particularly difficult to learn from experience alone.
4) **Hard to explain.** The stochasticity can sometimes make it hard to explain why a phenomenon occurred in a simulation.

In contrast, **human learning is highly efficient.** Thus, as a model of human behavior, this basic reinforcement learning update scheme is implausible:

- Human cognition is not that simplistic, and their actions are not that stochastic.
- Humans typically build and use a model of the world around them.
- Sometimes, it is possible to invest into multiple options at the same time

How can we address these challenges?

11.1.3 Dynamics of collective reinforcement learning

The essential idea of the collective reinforcement learning dynamics approach is to **replace** the **individual sample realizations** of the temporal-difference reward-prediction error **with its strategy average** plus a small error term,

$$\delta \leftarrow \delta_{\mathbf{x}} + \epsilon.$$

Thus, collective reinforcement learning dynamics **describe how agents with access to** (a good approximation of) **the strategy-average reward-prediction error would learn**.

There are multiple interpretations to motivate how the agents can obtain the strategy averages:

- **Model-based learners**. Agents have a model of how the environment works, including how the other agents behave currently, but not how the other agents learn. The agents use their world model to stabilize learning. In the limit of a perfect model (and sufficient cognitive resources), the error term vanishes, $\epsilon \to 0$.
- **Batch learners**. The agents store experiences (state observations, rewards, actions, next state observations) inside a memory batch and replay these experiences to make the learning more stable. Batch learning is a common algorithmic technique in machine learning. In the limit of an infinite memory batch, the error term vanishes, $\epsilon \to 0$ (Barfuss, 2020).
- **Different timescales**. The agents learn on two different time scales. On one time scale, the agents interact with the environment, collecting experiences and integrating them to improve their quality estimates while keeping their strategies fixed. On the other time scale, they use the accumulated experiences to adapt their strategy. Timescale separation is a common technique used in theoretical physics. In the limit of a complete time scale separation, having infinite experiences between two strategy updates, the error term vanishes, $\epsilon \to 0$ (Barfuss, 2022).

- **Proportional investors**. Instead of choosing actions individually, agents can invest an endowment into actions proportional to their policy. Assuming by analogy, that the environment is not in one of its states but described by its state distribution, agents receive feedback proportionally to their investment. When there is no noise in the rewards itself, the error term vanishes, $\epsilon \to 0$

In the following, we focus on the idealized case of a vanishing error term, $\epsilon \to 0$.

11.1.4 Learning goals

After this chapter, students will be able to:

- Explain the rationale of a dynamic systems treatment of reinforcement learning for complex human-environment interactions.
- Study dynamic system properties of multi-agent reinforcement learning in human-environment models
- Use open-source Python packages.

In the next section, we will **derive** the strategy-average deterministic approximation model of the multi-agent reinforcement learning process. It goes beyond this lecture to implement the learning dynamics ourselves (although we could if we invested enough time). Luckily, we can utilize an open-source Python package to **apply** and study the learning dynamics, which we will do in the section afterward.

11.2 Derivation

We import our usual libraries.

```
import numpy as np
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;
```

Then, we install the `pyCRLD` package from Github to compare the mathematical derivation with the respective code method.

```
!pip install git+https://github.com/barfusslab/pyCRLD.git
```

```
from pyCRLD.Agents.Base import abase as AgentBaseClass
```

From Equation 11.2,

$$\delta_t^i(s_t, a_t) = (1 - \gamma^i)r_t^i + \gamma^i Q_n^i(s_{t+1}, a_{t+1}) - Q_c^i(s_t, a_t),$$

we see that we need to construct the strategy-average reward, the strategy-average value of the next state, and the strategy-average value of the current state.

11.2.1 1) Rewards

The strategy-average version of the current reward is obtained by considering each agent i taking action a in state s when all other agents j act according to their strategy $x^j(s, a^j)$, causing the environment to transition to the next state s' with probability $T(s, a, s')$, during which agent i receives reward $R^i(s, a, s')$. Mathematically, we write,

$$R^i_{\mathbf{x}}(s, a) = \sum_{s'} \sum_{a^j} \prod_{j \neq i} x^j(s, a^j) T(s, a, s') R^i(s, \mathbf{a}, s').$$

Notation-wise, the formulation $\sum_{a^j} \prod_{j \neq i} X^j(s, a^j)$ is short for

$$\sum_{a^j} \prod_{j \neq i} X^j(s, a^j) = \sum_{a^1 \in \mathcal{A}^1} \cdots \sum_{a^{i-1} \in \mathcal{A}^{i-1}} \sum_{a^{i+1} \in \mathcal{A}^{i+1}} \cdots \sum_{a^N \in \mathcal{A}^N} x^1(s, a^1) \cdots x^{i-1}(s, a^{i-1}) x^{i+1}(s, a^{i+1}) \cdots x^N(s, a^N)$$

In the `pyCRLD` package, it is implemented as follows.

```
AgentBaseClass.Risa??
```

```
Signature:       AgentBaseClass.Risa(self, Xisa: jax.Array) -> jax.Array
Call signature:  AgentBaseClass.Risa(*args, **kwargs)
Type:            PjitFunction
String form:     <PjitFunction of <function abase.Risa at 0x140f4c4a0>>
File:            ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/Base.py
Source:
    @partial(jit, static_argnums=0)
    def Risa(self,
            Xisa:jnp.ndarray # Joint strategy
          ) -> jnp.ndarray:  # Average reward
        """Compute average reward `Risa`, given joint strategy `Xisa`"""
        i = 0; a = 1; s = 2; s_ = 3 # Variables
        j2k = list(range(4, 4+self.N-1))  # other agents
        b2d = list(range(4+self.N-1, 4+self.N-1 + self.N))  # all actions
        e2f = list(range(3+2*self.N, 3+2*self.N + self.N-1))  # all other acts

        sumsis = [[j2k[l], s, e2f[l]] for l in range(self.N-1)]  # sum inds
        otherX = list(it.chain(*zip((self.N-1)*[Xisa], sumsis)))

        args = [self.Omega, [i]+j2k+[a]+b2d+e2f] + otherX\
            + [self.T, [s]+b2d+[s_], self.R, [i, s]+b2d+[s_],
                [i, s, a]]
        return jnp.einsum(*args, optimize=self.opti)
```

The `@partial(jit, static_argnums=0)` decorator above the method makes the code execution fast. `jit` stands for just-in-time compilation and comes from the Python package JAX. Using JAX is very similar to using `numpy`. Hence, there is the JAX numpy module, `jnp`. See, for example, `jnp.einsum` in the code above.

Another *trick* is the use of the `self.Omega` object, which is a tensor of zeros and ones constructed to make the summation $\sum_{a^j} \prod_{j \neq i} X^j(s, a^j)$ work with the fast `einsum` method.

11.2.2 2) Next quality estimates

The strategy average of the following state value is likewise computed by averaging the over all actions of the other agents and the following states.

For each agent i, state s, and action a, all other agents $j \neq i$ choose their action a^j with probability $x^j(s, a^j)$. Consequently, the environment transitions to the next state s' with probability $T(s, a, s')$. At s', the agent estimates the quality of the next state to be of $v^i_{\mathbf{x}}(s') = \sum_{a^i \in \mathcal{A}^i} x^i(s', a^i) q^i_{\mathbf{x}}(s', a^i)$. Mathematically, we write,

$$^n Q^i_{\mathbf{x}}(s, a) = \sum_{s'} \sum_{a^j} \prod_{j \neq i} x^j(s, a^j) T(s, a, s') v^i_{\mathbf{x}}(s').$$

11.2.3 State values

We compute the state values $v^i_{\mathbf{x}}(s)$ exactly like in Chapters 03.01 and 03.03. We write the **Bellman equation in matrix form** and bring the values $\mathbf{v}^i_{\mathbf{x}}$ on one side,

$$\mathbf{v}^i_{\mathbf{x}} = (1 - \gamma^i)(\mathbf{1}_Z - \gamma^i \underline{\mathbf{T}}_{\mathbf{x}})^{-1} \mathbf{R}^i_{\mathbf{x}}.$$

In the `pyCRLD` package, it is implemented as follows.

```
AgentBaseClass.Vis??
```

```
Signature:
AgentBaseClass.Vis(
    self,
    Xisa: jax.Array,
    Ris: jax.Array = None,
    Tss: jax.Array = None,
    Risa: jax.Array = None,
) -> jax.Array
Call signature: AgentBaseClass.Vis(*args, **kwargs)
Type:           PjitFunction
String form:    <PjitFunction of <function abase.Vis at 0x140f4cb80>>
File:           ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/Base.py
Source:
    @partial(jit, static_argnums=0)
    def Vis(self,
            Xisa:jnp.ndarray, # Joint strategy
            Ris:jnp.ndarray=None, # Optional reward for speed-up
            Tss:jnp.ndarray=None, # Optional transition for speed-up
            Risa:jnp.ndarray=None  # Optional reward for speed-up
          ) -> jnp.ndarray:  # Average state values
        """Compute average state values `Vis`, given joint strategy `Xisa`"""
        # For speed up
        Ris = self.Ris(Xisa, Risa=Risa) if Ris is None else Ris
        Tss = self.Tss(Xisa) if Tss is None else Tss

        i = 0  # agent i
        s = 1  # state s
        sp = 2  # next state s'

        n = np.newaxis
        Miss = np.eye(self.Z)[n,:,:] - self.gamma[:, n, n] * Tss[n,:,:]

        invMiss = jnp.linalg.inv(Miss)
```

```
        return self.pre[:,n] * jnp.einsum(invMiss, [i, s, sp], Ris, [i, sp],
                                    [i, s], optimize=self.opti)
```

11.2.4 Transition matrix

The **transition matrix** $\underline{\mathbf{T}}_{\mathbf{x}}$ is a $Z \times Z$ matrix, where the element $T_{\mathbf{x}}(s, s')$ is the probability of transitioning from state s to s' under the joint policy $\mathbf{x}$. It is computed as

$$T_{\mathbf{x}}(s, s') = \sum_{a^i} \prod_i x^i(s, a^i) T(s, \mathbf{a}, s').$$

In the `pyCRLD` package, it is implemented as follows.

`AgentBaseClass.Tss??`

```
Signature:       AgentBaseClass.Tss(self, Xisa: jax.Array) -> jax.Array
Call signature:  AgentBaseClass.Tss(*args, **kwargs)
Type:            PjitFunction
String form:     <PjitFunction of <function abase.Tss at 0x140f4afc0>>
File:            ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/Base.py
Source:
    @partial(jit, static_argnums=0)
    def Tss(self,
            Xisa:jnp.ndarray  # Joint strategy
          ) -> jnp.ndarray: # Average transition matrix
        """Compute average transition model `Tss`, given joint strategy `Xisa`"""
        # i = 0  # agent i (not needed)
        s = 1  # state s
        sprim = 2  # next state s'
        b2d = list(range(3, 3+self.N))  # all actions

        X4einsum = list(it.chain(*zip(Xisa, [[s, b2d[a]] for a in range(self.N)])))
        args = X4einsum + [self.T, [s]+b2d+[sprim], [s, sprim]]
        return jnp.einsum(*args, optimize=self.opti)
```

11.2.5 State rewards

The **average reward** $\mathbf{R}_{\mathbf{x}}^i$ is a $N \times Z$-matrix, where the element $R_{\mathbf{x}}^i(s)$ is the expected reward agent i receives in state s under the joint policy $\mathbf{x}$. It is computed as

$$R_{\mathbf{x}}^i(s) = \sum_{s'} \sum_{a^i} \prod_i x^i(s, a^i) T(s, \mathbf{a}, s') R^i(s, \mathbf{a}, s').$$

In the `pyCRLD` package, it is implemented as follows.

`AgentBaseClass.Ris??`

```
Signature:       AgentBaseClass.Ris(self, Xisa: jax.Array, Risa: jax.Array = None) -> jax.Array
Call signature:  AgentBaseClass.Ris(*args, **kwargs)
Type:            PjitFunction
String form:     <PjitFunction of <function abase.Ris at 0x140f4bd80>>
File:            ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/Base.py
Source:
    @partial(jit, static_argnums=0)
```

```
def Ris(self,
        Xisa:jnp.ndarray, # Joint strategy
        Risa:jnp.ndarray=None # Optional reward for speed-up
       ) -> jnp.ndarray: # Average reward
    """Compute average reward `Ris`, given joint strategy `Xisa`"""
    if Risa is None:  # for speed up
        # Variables
        i = 0; s = 1; sprim = 2; b2d = list(range(3, 3+self.N))

        X4einsum = list(it.chain(*zip(Xisa,
                                 [[s, b2d[a]] for a in range(self.N)])))

        args = X4einsum + [self.T, [s]+b2d+[sprim],
                           self.R, [i, s]+b2d+[sprim], [i, s]]
        return jnp.einsum(*args, optimize=self.opti)

    else:  # Compute Ris from Risa
        i=0; s=1; a=2
        args = [Xisa, [i, s, a], Risa, [i, s, a], [i, s]]
        return jnp.einsum(*args, optimize=self.opti)
```

11.2.6 3) Current quality estimates

Assuming that agents select their actions according to a softmax policy function,

$$x_t^i(s,a) = \frac{\exp \beta^i Q_t^i(s,a)}{\sum_b \exp \beta^i Q_t^i(s,b)}, \tag{11.3}$$

where β^i is the intensity of choice of agent i, we can **reformulate the update of the state-action quality estimates (Equation 11.1) into an update of the policy**, i.e., state-action probabilities. Doing so reduces the dynamic system's state space size, as we do not need to track the quality estimates of each agent in each state-action pair. Instead, we only need to track the state-action probabilities of each agent in each state-action pair. This is advantageous as the **lower dimensional dynamic state space is more straightforward to analyze and visualize**.

For the derivation of the joint policy update, we need to solve the policy function for $Q_t^i(s,a)$,

$$Q_t^i(s,a) = \frac{1}{\beta^i} \ln x_t^i(s,a) + \frac{1}{\beta^i} \ln \left[\sum_b \exp \beta^i Q_t^i(s,b) \right] \tag{11.4}$$

$$= \frac{1}{\beta^i} \ln x_t^i(s,a) + C^i(s) \tag{11.5}$$

where $C^i(s)$ denots a constant in actions. It may vary for each agent and state but not for actions.

The step-by-step derivation of the joint policy update is as follows:

$$x_{t+1}^i(s,a) = \frac{\exp \beta^i Q_{t+1}^i(s,a)}{\sum_b \exp \beta^i Q_{t+1}^i(s,b)} \tag{11.6}$$

$$= \frac{\exp\left[\beta^i\left(Q_t^i(s,a) + \alpha^i \delta_t^i(s,a)\right)\right]}{\sum_b \exp\left[\beta^i\left(Q_t(s,b) + \alpha^i \delta_t^i(s,b)\right)\right]} \qquad \text{Inserting the belief update} \tag{11.7}$$

$$= \frac{\exp\left[\beta^i Q_t^i(s,a)\right] \exp\left[\alpha^i \beta^i \delta_t^i(s,a)\right]}{\sum_b \exp\left[\beta^i Q_t(s,b)\right] \exp\left[\alpha^i \beta^i \delta_t^i(s,b)\right]} \qquad \text{Factoring the exponentials} \tag{11.8}$$

$$= \frac{x_t^i(s,a) \exp\left[\alpha^i \beta^i \delta_t^i(s,a)\right]}{\sum_b x_t^i(s,b) \exp\left[\alpha^i \beta^i \delta_t^i(s,b)\right]} \qquad \text{Multiplying by } \tfrac{\frac{1}{z}}{\frac{1}{z}} \text{ with } z = \sum_c \exp \beta^i Q_t^i(s,c) \tag{11.9}$$

$$= \frac{x_t^i(s,a) \exp\left[\alpha^i \beta^i \delta_{\mathbf{x}}^i(s,a)\right]}{\sum_b x_t^i(s,b) \exp\left[\alpha^i \beta^i \delta_{\mathbf{x}}^i(s,b)\right]} \qquad \text{Replacing sample } \delta_t^i \text{ with strategy-average } \delta_{\mathbf{x}}^i \tag{11.10}$$

$$= \frac{x_t^i(s,a) \exp\left[\alpha^i \beta^i \left((1-\gamma^i)R_{\mathbf{x}}^i(s,a) + \gamma^i \cdot {}^nQ_{\mathbf{x}}^i(s,a) - Q_t^i(s,a)\right)\right]}{\sum_b x_t^i(s,b) \exp\left[\alpha^i \beta^i \left((1-\gamma^i)R_{\mathbf{x}}^i(s,b) + \gamma^i \cdot {}^nQ_{\mathbf{x}}^i(s,b) - Q_t^i(s,b)\right)\right]} \qquad \text{Filling } \delta_{\mathbf{x}}^i \tag{11.11}$$

$$= \frac{x_t^i(s,a) \exp\left[\alpha^i \beta^i \left((1-\gamma^i)R_{\mathbf{x}}^i(s,a) + \gamma^i \cdot {}^nQ_{\mathbf{x}}^i(s,a) - \frac{1}{\beta^i}\ln x_t^i(s,a)\right)\right]}{\sum_b x_t^i(s,b) \exp\left[\alpha^i \beta^i \left((1-\gamma^i)R_{\mathbf{x}}^i(s,b) + \gamma^i \cdot {}^nQ_{\mathbf{x}}^i(s,b) - \frac{1}{\beta^i}\ln x_t^i(s,b)\right)\right]} \tag{11.12}$$

$$\text{Using } Q_t^i(s,a) = \frac{1}{\beta^i}\ln x_t(s,a) + C^i(s) \tag{11.13}$$

In summary, the strategy-average of the current state-action value, $Q_t^i(s,a)$ is

$$\frac{1}{\beta^i}\ln x^i(s,a).$$

11.2.7 Strategy-average reward-prediction temporal-difference error

```
from pyCRLD.Agents.StrategySARSA import stratSARSA
```

Taken together, the strategy-average reward-prediction error is

$$\delta_{\mathbf{x}}^i(s,a) = (1-\gamma^i)R_{\mathbf{x}}^i(s,a) + \gamma^i \cdot {}^nQ_{\mathbf{x}}^i(s,a) - \frac{1}{\beta^i}\ln x^i(s,a),$$

to be inserted in the joint policy update,

$$x_{t+1}^i(s,a) = \frac{x_t^i(s,a) \exp\left[\alpha^i \beta^i \delta_{\mathbf{x}}^i(s,a)\right]}{\sum_b x_t^i(s,b) \exp\left[\alpha^i \beta^i \delta_{\mathbf{x}}^i(s,b)\right]}.$$

We made the strategy update independent of the quality beliefs.

In the pyCRLD package, update step is implement as follows,

```
stratSARSA.step??
```

```
Signature:       stratSARSA.step(self, Xisa) -> tuple
Call signature: stratSARSA.step(*args, **kwargs)
Type:            PjitFunction
String form:     <PjitFunction of <function strategybase.step at 0x140f907c0>>
File:            ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/StrategyBase.py
Source:
    @partial(jit, static_argnums=0)
    def step(self,
             Xisa  # Joint strategy
            ) -> tuple:  # (Updated joint strategy, Prediction error)
        """
        Performs a learning step along the reward-prediction/temporal-difference error
        in strategy space, given joint strategy `Xisa`.
        """
        TDe = self.TDerror(Xisa)
        n = jnp.newaxis
        XexpaTDe = Xisa * jnp.exp(self.alpha[:,n,n] * TDe)
        return XexpaTDe / XexpaTDe.sum(-1, keepdims=True), TDe
```

The `step` method comes from a parent class, called `strategybase`, and calls the `TDerror` method,
which is initialized upon creating a specific agent collective with the concrete reward-prediction error
method from the SARSA agent.

The reward-prediction error of the SARSA agent is implemented as follows.

```
stratSARSA.RPEisa??
```

```
Signature:       stratSARSA.RPEisa(self, Xisa, norm=False) -> numpy.ndarray
Call signature: stratSARSA.RPEisa(*args, **kwargs)
Type:            PjitFunction
String form:     <PjitFunction of <function stratSARSA.RPEisa at 0x140f905e0>>
File:            ~/Other/miniconda3/envs/iw-dev/lib/python3.11/site-packages/pyCRLD/Agents/StrategySARSA.py
Source:
    @partial(jit, static_argnums=(0,2))
    def RPEisa(self,
               Xisa,  # Joint strategy
               norm=False # normalize error around actions?
              ) -> np.ndarray:  # RP/TD error
        """
        Compute reward-prediction/temporal-difference error for
        strategy SARSA dynamics, given joint strategy `Xisa`.
        """
        R = self.Risa(Xisa)
        NextQ = self.NextQisa(Xisa, Risa=R)

        n = jnp.newaxis
        E = self.pre[:,n,n]*R + self.gamma[:,n,n]*NextQ - 1/self.beta[:, n, n] * jnp.log(Xisa)
        E *= self.beta[:,n,n]

        E = E - E.mean(axis=2, keepdims=True) if norm else E
        return E
```

11.3 Application

Let us apply the collective reinforcement learning dynamics to the ecological public good environment
from Chapter 03.03. We will highlight the complex dynamics phenomena that arise from the collective
reinforcement learning dynamics (Barfuss, Flack, et al., 2024).
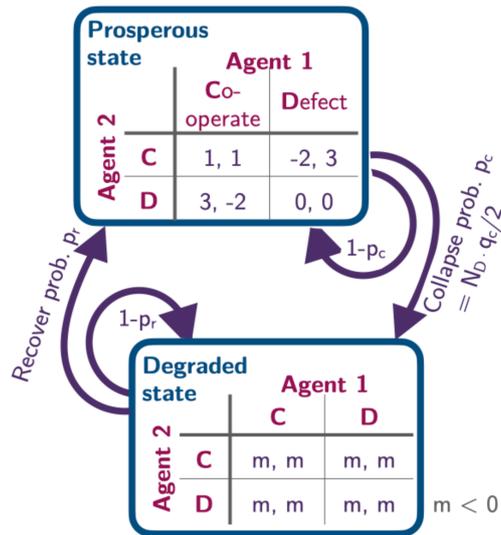
Barfuss et al. (2020) Caring for the future can turn tragedy into comedy for long-term collective action under risk of collapse

Figure 11.1: Ecological public good collective decision-making environment

For convenience, we import the environment class from the `pyCRLD` package. However, you now possess all the skills needed to implement it on your own.

```
from pyCRLD.Environments.EcologicalPublicGood import EcologicalPublicGood as EcoPG
```

We initialize the environment with two agents, a benefit-to-cost ratio of $f = 1.2$, a cost of $c = 5$, a collapse impact of $m = -5$, a collapse leverage of 0.2, and a recovery probability of 0.01. We set the `degraded_choice` parameter to `False` to remove all agency from the agents in the degraded state. In other word, regardless what they do in the degraded state, they have to wait for the recovery on average $1/q_r$ timesteps.

```
# Inititalize the ecological public good environment
env = EcoPG(N=2, f=1.2, c=5, m=-5, qc=0.2, qr=0.01, degraded_choice=False)
```

These parameters ensure to have the same short-term welfare values in the prosperous state as shown in the Figure above.

```
p = env.Sset.index('p'); g = env.Sset.index('g') # indices of the prosperous and
↪  degraded state
print("Agent zero's welfare\n", env.R[0, p, :, :, p])
print("\nAgent one's welfare\n", env.R[1, p, :, :, p])
```

```
Agent zero's welfare
 [[ 1. -2.]
 [ 3.  0.]]

Agent one's welfare
 [[ 1.  3.]
 [-2.  0.]]
```

11.3.1 Learning trajectories

We create a multi-agent-environment interface `MAEi` composed of SARSA agents with a learning rate of 0.05, a choice intensity of 50.0, and a discount factor of 0.75. We set the `use_prefactor` parameter to `True` to use the pre-factor $(1 - \gamma)$ in the policy update.

```
MAEi = stratSARSA(env, learning_rates=0.05, choice_intensities=50.0,
 ↪  discount_factors=0.75, use_prefactor=True)
```

Let us evolve the learning from a random initial joint policy,

```
x = MAEi.random_softmax_strategy()
x
```

```
Array([[[0.65391284, 0.34608716],
        [0.540063  , 0.45993698]],

       [[0.23748323, 0.7625168 ],
        [0.6527203 , 0.34727973]]], dtype=float32)
```

for a maximum of 5000 time steps with a convergence tolerance of 10^{-5}. Thus, if two consecutive joint policies are closer than 10^{-5}, the learning process stops.

```
policy_trajectory_Xtisa, fixedpointreached = MAEi.trajectory(x, Tmax=5000,
 ↪  tolerance=10**-5)
fixedpointreached
```

```
True
```

We have reached a fixed point and the learning trajecotry has a length of

```
len(policy_trajectory_Xtisa)
```

```
153
```

Let us visualize the time evolution of learning trajectory.

```
c = env.Aset[0].index('c'); d = env.Aset[0].index('d')  # action indices
plt.plot(policy_trajectory_Xtisa[:, 0, p, c], label='Agent zero in prosperous
 ↪  state', c='blue', lw=3, alpha=0.5)
plt.plot(policy_trajectory_Xtisa[:, 1, p, c], label='Agent one in prosperous state',
 ↪  c='blue', ls='--')
plt.plot(policy_trajectory_Xtisa[:, 0, g, c], label='Agent zero in degraded state',
 ↪  c='red', lw=3, alpha=0.5)
plt.plot(policy_trajectory_Xtisa[:, 1, g, c], label='Agent one in degraded state',
 ↪  c='red', ls='--');
plt.xlabel('Time steps'); plt.ylabel('Cooperation probability'); plt.legend();
 ↪  plt.ylim(0, 1);
```
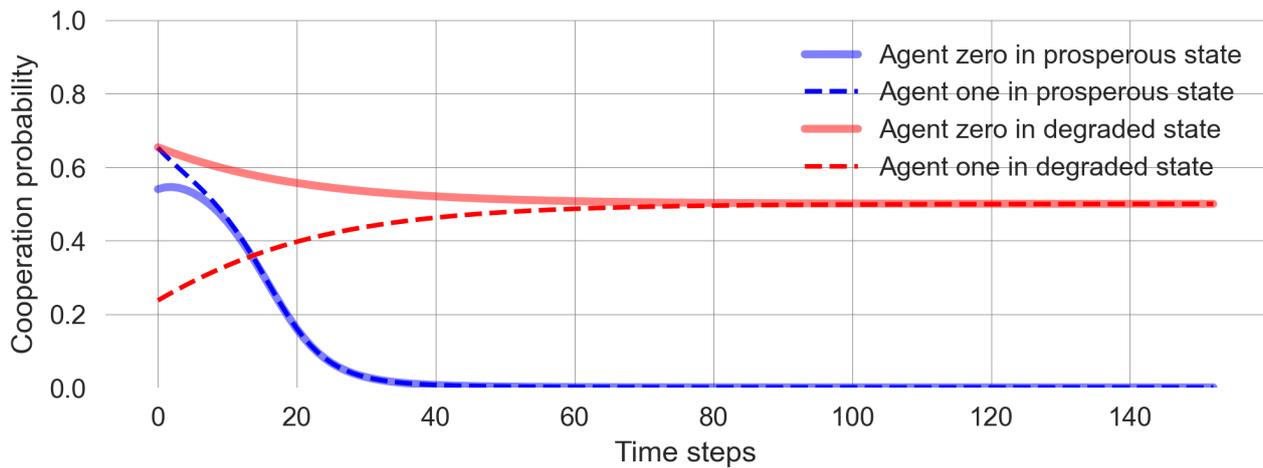
Let's repeat this serveral times from different random joint policies. **Exectue the cell below multiple times and observe what happens**.

```
x = MAEi.random_softmax_strategy()
policy_trajectory_Xtisa, fixedpointreached = MAEi.trajectory(x, Tmax=5000,
↪  tolerance=10**-5)

plt.plot(policy_trajectory_Xtisa[:, 0, p, c], label='Agent zero in prosperous
↪  state', c='blue', lw=3, alpha=0.5)
plt.plot(policy_trajectory_Xtisa[:, 1, p, c], label='Agent one in prosperous state',
↪  c='blue', ls='--')
plt.plot(policy_trajectory_Xtisa[:, 0, g, c], label='Agent zero in degraded state',
↪  c='red', lw=3, alpha=0.5)
plt.plot(policy_trajectory_Xtisa[:, 1, g, c], label='Agent one in degraded state',
↪  c='red', ls='--');
plt.xlabel('Time steps'); plt.ylabel('Cooperation probability'); plt.legend();
↪  plt.ylim(0, 1);
```
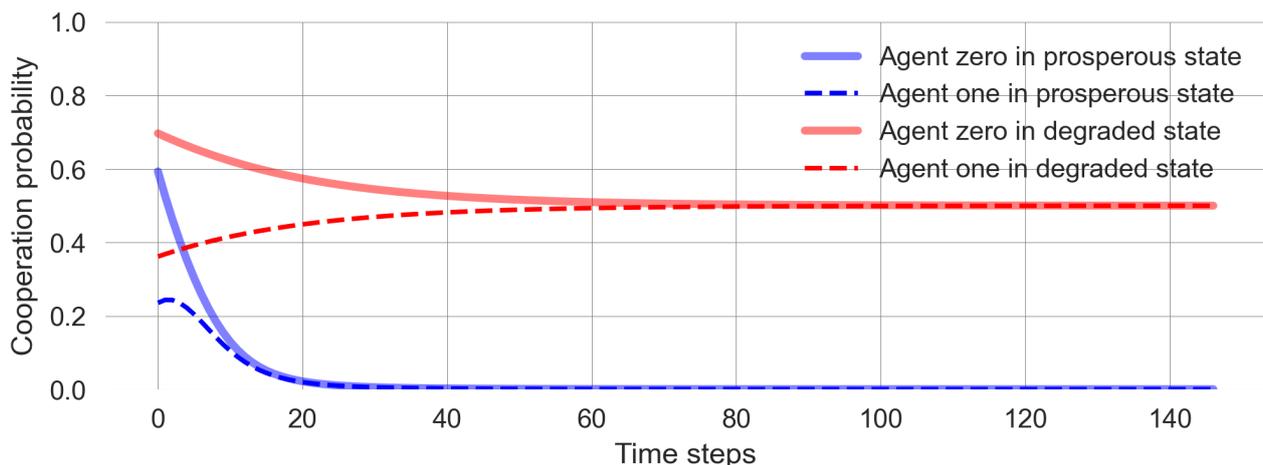

Some observations you should make:

- **Learning occurs fast**. The agents quickly attain a stable state within just a few hundred steps, and their execution is remarkably rapid.

- **Learning is deterministic**. Given an initial joint policy, the learning process has no stochastic fluctuations. The agents learn deterministically. However, *what* they learn is a probability distribution.
- **Outcome is bistable**. The agents learn to either cooperate or defect completely in the prosperous state, depending on where they start. If they start closer to cooperation, they learn to cooperate. If they start closer to defection, they learn to defect.
- **Agents randomize**. In the degraded state, agents learn to randomize over actions fully, i.e., choose each of their two options with a probability of 0.5. This is because the agents cannot influence the outcome of their actions and, thus, are driven only by exploration. You can imagine the desire to explore as a form of intrinsic motivation that dominates here without controllable extrinsic rewards.

11.3.2 Flow plot

The determinism and the fast computation allow for an improved visualization of the learning process. As with any deterministic dynamic system, we can visualize the flow plot of the dynamics (See Chapter 02.01).

In the `pyCRLD` package, we have a special module for that purpose.

```
from pyCRLD.Utils import FlowPlot as fp
```

Applying this function yields a flow plot of the learning dynamics which highlights the bistability of the learning process in the prosperous state and the randomization in the degraded state.

```
x = ([0], [g,p], [c])  # which (agent, observation, action) to plot on x axis
y = ([1], [g,p], [c])  # which (agent, observation, action) to plot on y axis
eps=10e-3; action_probability_points = np.linspace(0+eps, 1.0-eps, 9)
ax = fp.plot_strategy_flow(MAEi, x, y, action_probability_points, conds=env.Sset)
```
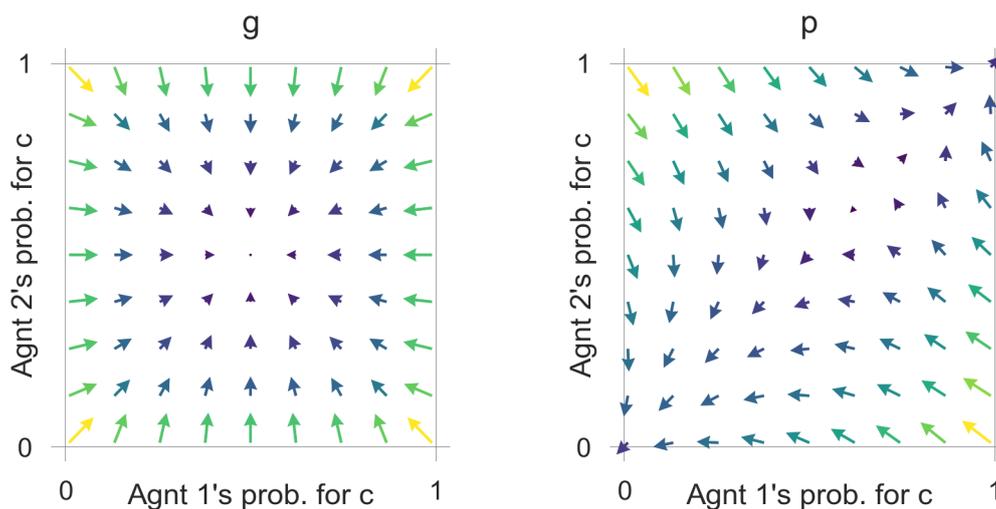

Figure 11.2: Basic flow of collective reinforcement learning dynamics.

These flow plots allow for a geometric understanding of the collective learning dynamics over the whole joint policy space. In contrast to a standard flow plot, per default, the **arrows show the temporal-difference reward prediction error**. Thus, they have a cognitive interpretation.

We may use them to study how the parameters of the learning agents and the environment influence the outcome.

```
def plot_flow(DiscountFactor=0.75, ChoiceIntensity=50, CollapseImpact=-5,
↪ CollapseLeverage=0.2):
    env = EcoPG(N=2, f=1.2, c=5, m=CollapseImpact, qc=CollapseLeverage,
                qr=0.01, degraded_choice=False)
    MAEi = stratSARSA(env, learning_rates=0.05, choice_intensities=ChoiceIntensity,
                      discount_factors=DiscountFactor, use_prefactor=True)

    x = ([0], [g,p], [c])  # which (agent, observation, action) to plot on x axis
    y = ([1], [g,p], [c])  # which (agent, observation, action) to plot on y axis
    eps=10e-3; action_probability_points = np.linspace(0+eps, 1.0-eps, 9)
    ax = fp.plot_strategy_flow(MAEi, x, y, action_probability_points,
↪ conds=env.Sset)
```

When working with this material in a Jupyter notebook, we can interactively study the parameter dependence of the flow plot.

For example, caring more for the future makes the cooperative basin of attraction larger.

```
plot_flow(DiscountFactor=0.8)
```
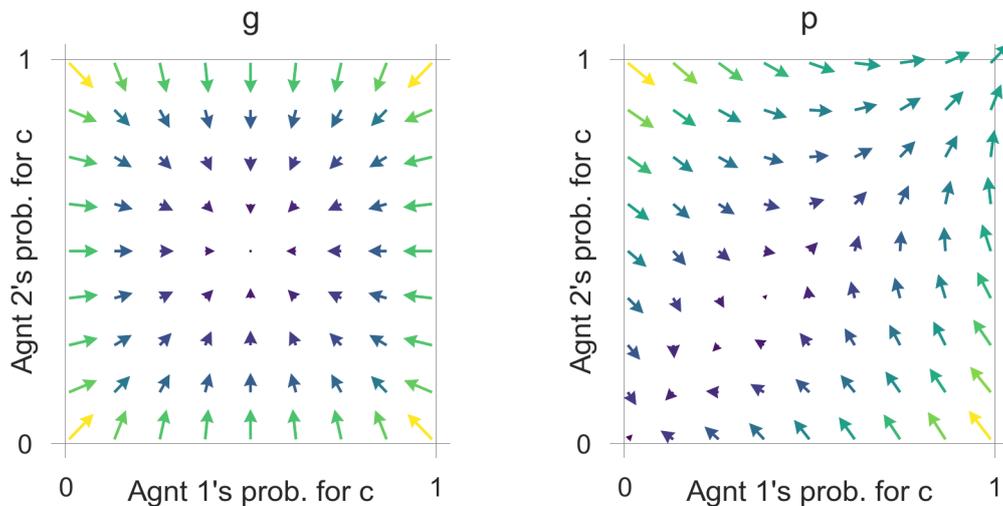

Figure 11.3: Learning flow with more future caring.

So does a more severe collapse impact,

```
plot_flow(CollapseImpact=-6)
```
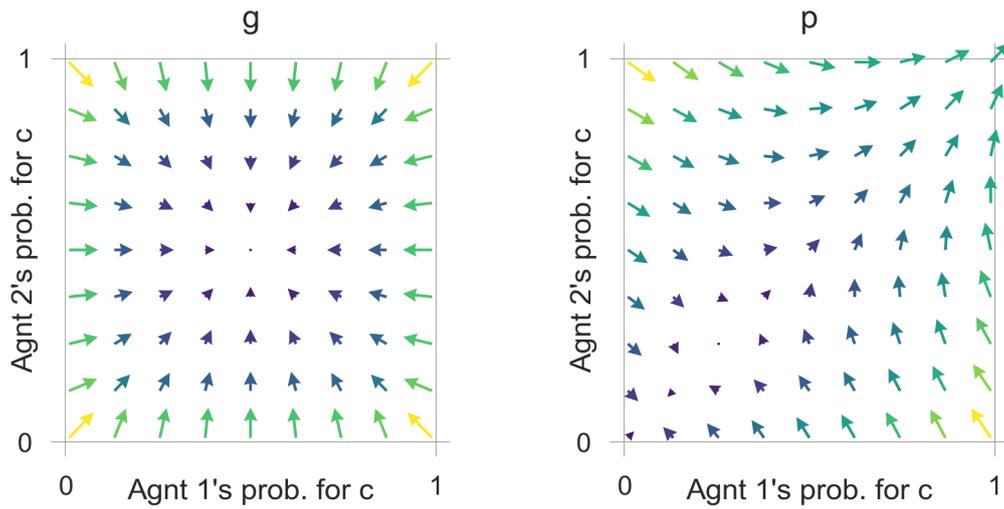
Figure 11.4: Learning flow with a more severe collapse impact.

and a collapse that occurse more likely or faster.

```
plot_flow(CollapseLeverage=0.3)
```
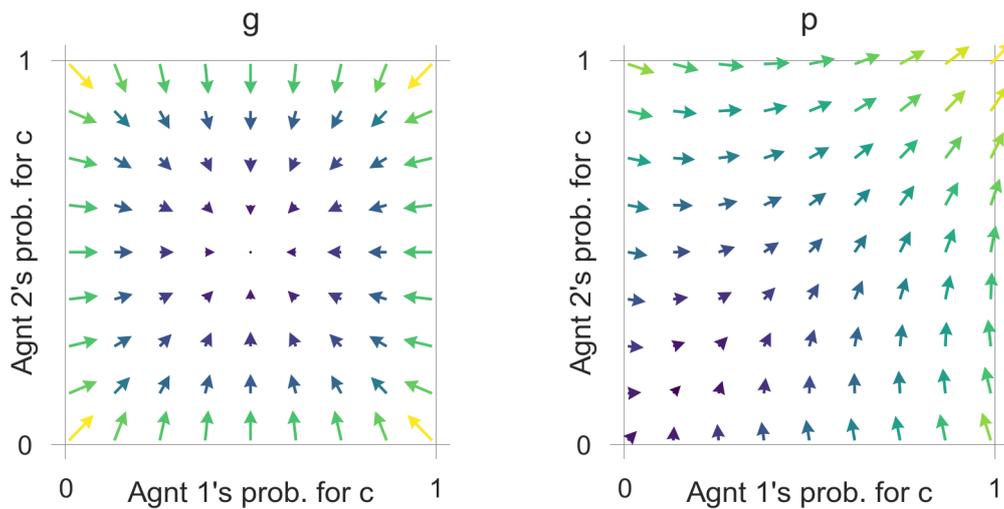

Figure 11.5: Learning flow with a higher collapse leverage.

The flow in the degraded state is unaffected by these parameter modulations.

A very low choice intensity makes the desire to explore (i.e., randomize) dominate also in the prosperous state.

```
plot_flow(ChoiceIntensity=1)
```
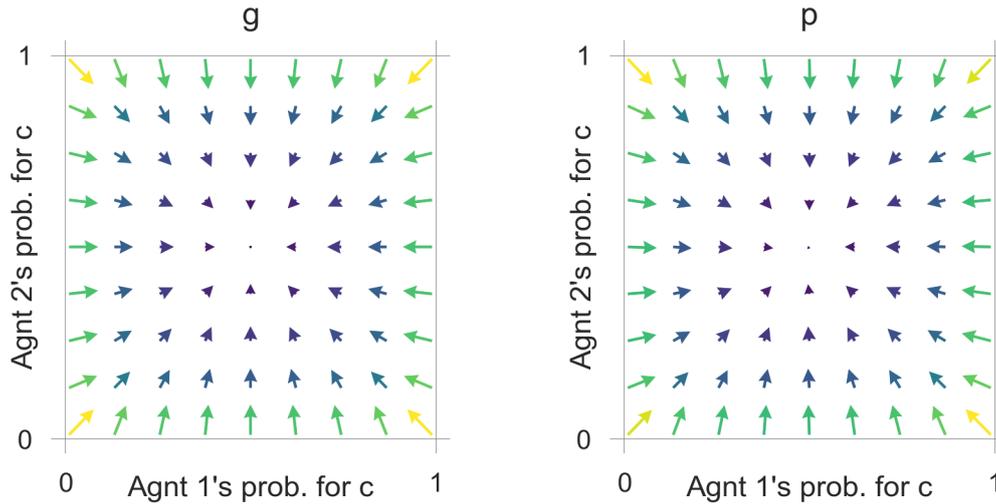
Figure 11.6: Learning flow with a small intensity of choice makes explorative behavior dominant.

11.3.3 Critical transition

Let us study the learning behavior around the separatrix of the bistable region.

First, we define a function that allows us to enter initial cooperation probabilities for both agents and return a proper joint policy. This function sets the cooperation probability in the degraded state to 0.5 for both agents, as we have seen that the agents will eventually learn to randomize in the degraded state and we are not interested in that part of the learning behavior.

```
def compile_strategy(p0c:float,  # cooperation probability of agent zero
                     p1c:float):  # cooperation probability of agent one
    Pi = np.array([0.5, p0c])  # coop. prob. in the degraded state set to 0.5
    Pj = np.array([0.5, p1c])
    xi = np.array([Pi, 1-Pi]).T
    xj = np.array([Pj, 1-Pj]).T
    return np.array([xi, xj])
```

We setup the multiagent-environment interaface.

```
env = EcoPG(N=2, f=1.2, c=5, m=-5, qc=0.2, qr=0.01, degraded_choice=False)
MAEi = stratSARSA(env=env, learning_rates=0.01, choice_intensities=100,
↪  discount_factors=0.75,
                  use_prefactor=True)
```

To get a feeling for the critical transition, we create three well chosen learning trajectories.

```
xtrajs = []  # storing strategy trajectories
fprs = []    # and whether a fixed point is reached
for pc in [0.18, 0.19, 0.20]:  # cooperation probability of agent 1
    X = compile_strategy(pc, 0.95)
    xtraj, fixedpointreached = MAEi.trajectory(X, Tmax=5000, tolerance=10**-5)
    xtrajs.append(xtraj); fprs.append(fixedpointreached)
    print("Trajectory length:",len(xtraj))
```

```
Trajectory length: 178
Trajectory length: 234
Trajectory length: 174
```

We plot them ontop of the learning flow.

```
fig = plt.figure(figsize=(12, 3.5)); ax = fig.add_subplot(132) # to center the plot
fig.add_subplot(131, xticks=[], yticks=[]); fig.add_subplot(133, xticks=[],
↪  yticks=[]);

x = ([0], [p], [c])  # which (agent, observation, action) to plot on x axis
y = ([1], [p], [c])  # which (agent, observation, action) to plot on y axis
eps=10e-3; action_probability_points = np.linspace(0+eps, 1.0-eps, 9)
fp.plot_strategy_flow(MAEi, x, y, action_probability_points, axes=[ax])

# Add trajectories to flow plot
fp.plot_trajectories(xtrajs, x=x, y=y, fprs=fprs, cols=['red','blue','blue'],
                     lws=[2], msss=[2], lss=['-'], alphas=[0.75], axes=[ax]);

ax.set_ylabel("Agent 2's cooperation probability");
ax.set_xlabel("Agent 1's cooperation probability");
```
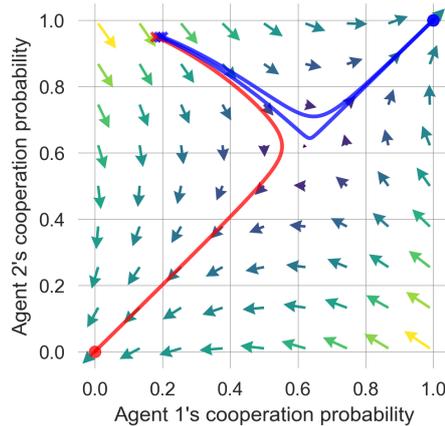

Next, we create a more fine-grained bundle of learning trajectories.

```
# Cooperation probability of agent 1
pcs = np.concatenate([np.linspace(0.01, 0.99, 51), np.linspace(0.185, 0.195, 151)])
pcs = np.sort(np.unique(pcs))

Xktisa = []  # storing strategy trajectories
fprs = []    # and whether a fixed point is reached
for i, pc in enumerate(pcs):
    print(f"Progress: {((i+1)/len(pcs)):.2%}", end="\r")
    X = compile_strategy(pc, 0.95)
    PolicyTrajectories_Xtisa, fixedpointreached = MAEi.trajectory(X, Tmax=5000,
↪  tolerance=10**-5)
    Xktisa.append(PolicyTrajectories_Xtisa)
    fprs.append(fixedpointreached)
```

```
Progress: 100.00%
```

We obtain the critical point in this bundle of learning trajectories where the two agents switch or tip from complete defection to complete cooperation.

First, we check that all trajectories converged.

```
np.all(fprs)
```

```
True
```

Then, we obtain the cooperation probabilities at convergence.

```
converged_pcs = np.array([Xtisa[-1][:, p, c] for Xtisa in Xktisa])
converged_pcs.shape
```

```
(201, 2)
```

Last, we show the biomodal distribution of full defection and full cooperation.

```
np.histogram(np.array(converged_pcs).mean(-1), range=(0,1))[0]
```

```
array([ 80,   0,   0,   0,   0,   0,   0,   0,   0, 121])
```

Thus, the critical point lies at the index

```
cp = np.histogram(np.array(converged_pcs).mean(-1), range=(0,1))[0][0]
cp
```

```
80
```

and has an approximate value between

```
print(pcs[cp-1], 'and', pcs[cp], '.')
```

```
0.18966666666666668 and 0.18973333333333334 .
```

We use this more fine-grained bundle of learning trajectories to visualize the phenomenon of a **critical slowing down** by plotting the time steps required to reach convergence.

```
plt.plot(pcs[:cp], [len(Xtisa) for Xtisa in Xktisa[:cp]],
         '-', color='red', lw=2, alpha=0.8)  # defectors in red
plt.plot(pcs[cp:], [len(Xtisa) for Xtisa in Xktisa[cp:]],
         '-', color='blue', lw=2, alpha=0.6) # cooperators in blue
plt.ylim(0); plt.ylabel('Timesteps to convergence')
plt.xlabel(f"Agent 1's cooperation probability in the prosperous state");
```
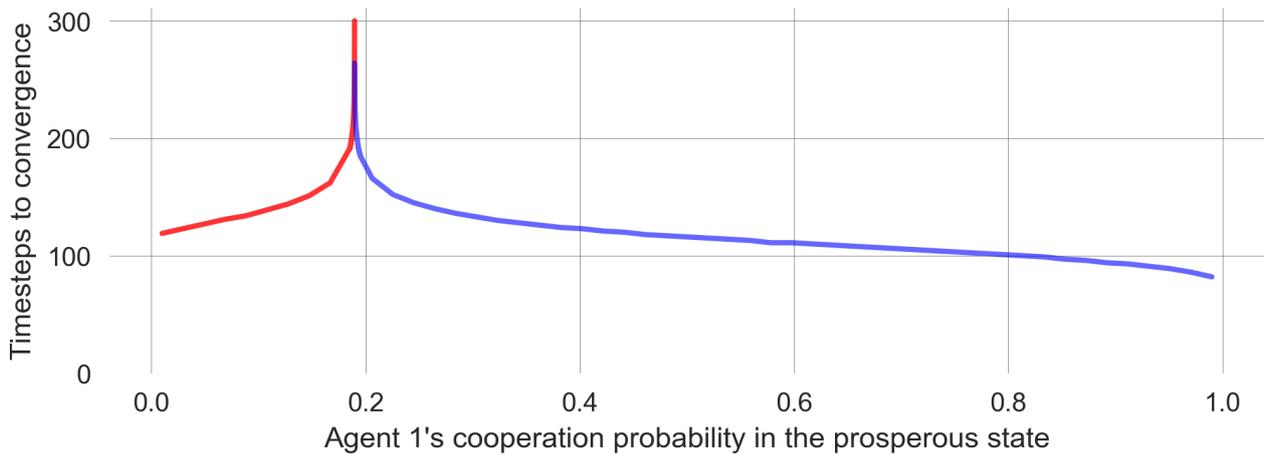
Figure 11.7: Time steps required to convergence show a critical slowing down around the tipping point.

We also observe a kind of **transient tipping point** in the learning dynamics, when plotting the two closest trajectories around the critical point.

```python
def plot_TransientTipping(xlim=None):
    # Plot the defecting learners in red
    plt.plot(Xktisa[cp-1][:, 0, p, c], color='red', lw=5, ls=':', label='Agent
↪ zero')
    plt.plot(Xktisa[cp-1][:, 1, p, c], color='red', lw=4, ls="--", alpha=0.4,
↪ label='Agent one')

    # Plot the cooperating learners in blue
    plt.plot(Xktisa[cp][:, 0, p, c], color='blue', lw=3, ls=':', label='Agent zero')
    plt.plot(Xktisa[cp][:, 1, p, c], color='blue', lw=2, ls="--", alpha=0.4,
↪ label='Agent one')

    plt.xlim(xlim); plt.legend(); plt.xlabel("Timesteps"); plt.ylabel("Cooperation")
```

```python
plot_TransientTipping()
```
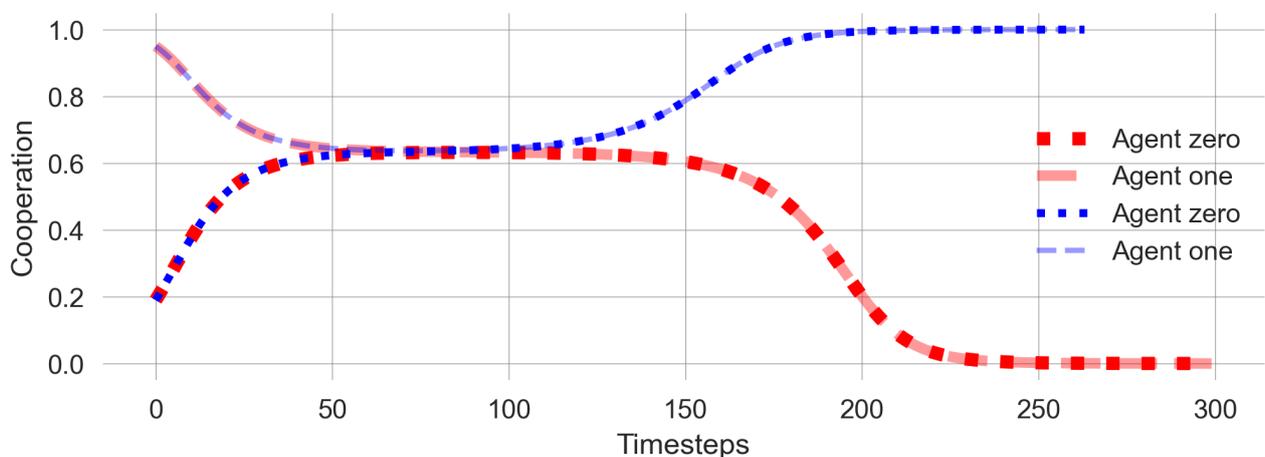

Figure 11.8: Emergent time scale seperation at the critical point.

252

During this **emergent timescale separation**, the learning process seems to settle on a mixed policy after approximately 50 timesteps. It remains at this point for another 50 steps, which is the same duration it took to reach this mixed policy (Figure 11.9). The learning adjusts the policies more rapidly after this period until they converge to two deterministic policies.

```
plot_TransientTipping((0, 95))
```
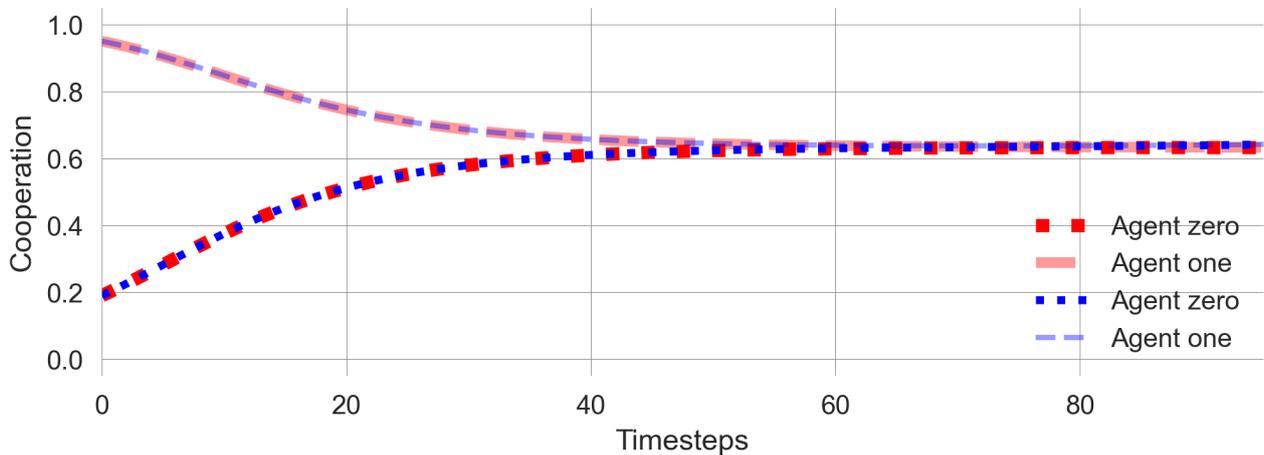

Figure 11.9: Apparent convergence to a mixed policy.

11.3.4 Hysteresis

The last phenomenon we want to highlight is hysteresis (See Chapter 02.02). We study the cooperation probabilities of the agents in the prosperous state as a function of the discount factor γ. We know from Chapter 03.03 that caring for the future can turn a tragedy of the commons into a comedy while passing through the coordination regime.

In the following, we start at a relatively low level of caring for the future, increase it, and then decrease it again, all while letting the agent learn along

First, let us create the discount factor values.

```
dcfs = list(np.arange(0.6, 0.9, 0.005))
hystcurve = dcfs + dcfs[::-1]
```

Then, we set up the environment and start the simulation from a random policy. We let the agents learn for 2500 time steps or until the learning process converges with a tiny tolerance. Then, we record the final policy, advance the discount factor, and restart from the previous final policy.

```
# Set up the ecological public goods environment
env = EcoPG(N=2, f=1.2, c=5, m=-5, qc=0.2, qr=0.01, degraded_choice=False)

coops = []  # for storing the cooperation probabilities
X = MAEi.random_softmax_strategy()
for i, dcf in enumerate(hystcurve):
    # Adjust multi-agent environment interface with discount factor
    MAEi = stratSARSA(env=env, discount_factors=dcf, use_prefactor=True,
                      learning_rates=0.05, choice_intensities=50)
```

```
    trj, fpr = MAEi.trajectory(X, Tmax=2500, tolerance=10e-12)
    print(f"Progress: {((i+1)/len(hystcurve)):6.2%} |",
            f"Discount Factor {dcf:5.4} | Conv?: {fpr}" , end="\r")
    X = trj[-1] # select last strategy
    coops.append(X[:, 1, 0]) # append to storage container
```

```
Progress: 100.00% | Discount Factor   0.6 | Conv?: True
```

Now, we plot the computed data. We use the points' size and color to indicate the time dimensions of the discount factor changes. The time flows from big to small data points and from dark to light ones.

```
# Plot background line
plt.plot(hystcurve, np.array(coops).mean(-1),'-',alpha=0.5,color='k',zorder=-1)
# Plot data points with size and color indicating the time dimension
plt.scatter(hystcurve, np.array(coops).mean(-1), alpha=0.9, cmap='viridis',
            s=np.arange(len(hystcurve))[::-1]+1, c=np.arange(len(hystcurve)))

plt.ylabel('Cooperation'); plt.xlabel('Discount Factor'); #plt.ylim(0,1)
```
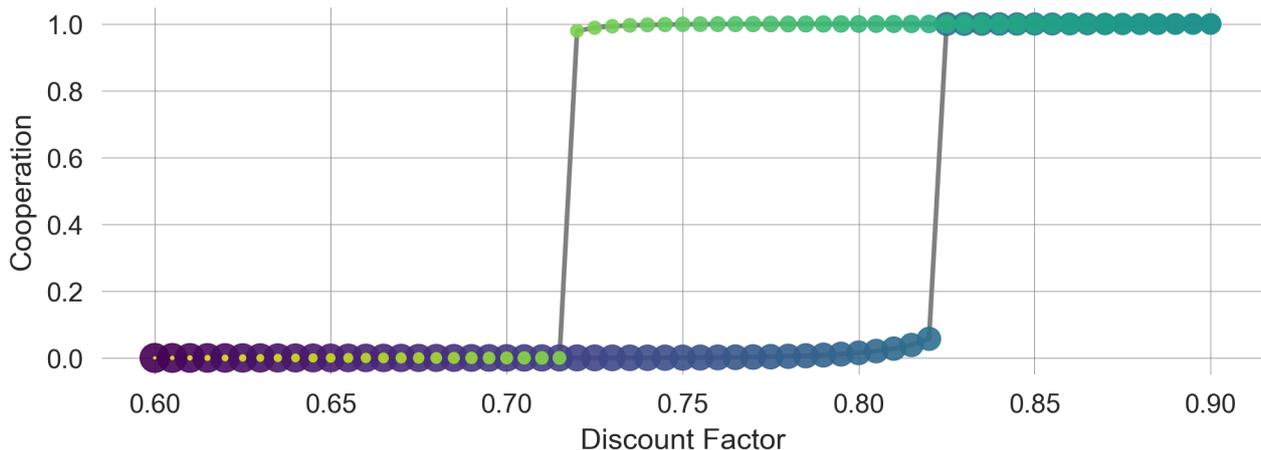

Figure 11.10: Hysteresis curve

The hysteresis curve shows that the probability of cooperation among agents in the prosperous state depends on the history of the discount factor. The agents' learning dynamics exhibit a memory of the past, a typical feature of complex systems.

11.4 Learning goals revisited

In this chapter,

- we introduced deterministic approximation models of the stochastic reinforcement learning process as a valuable tool for modeling complex human-environment interactions. Collective reinforcement learning dynamics model adaptive agents (in stylized model environments)

    - that use a perfect model of the world

254

- in a computationally fast
  - transparent
  - and deterministically evolving way.

- We studied complex dynamic phenomena of multi-agent reinforcement learning in the ecological public good environment.
- To do so, we used the open-source `pyCRLD` Python package.

11.5 Synthesis

Collective reinforcement learning dynamics bridge agent-based, equation-based (dynamic systems), and equilibrium-based modeling:

- agent-based: derived from individual agent characteristics
- equation-based: treated as a dynamical systems
- equilibrium-based: fixed points are (close to) the classic equilibrium solutions
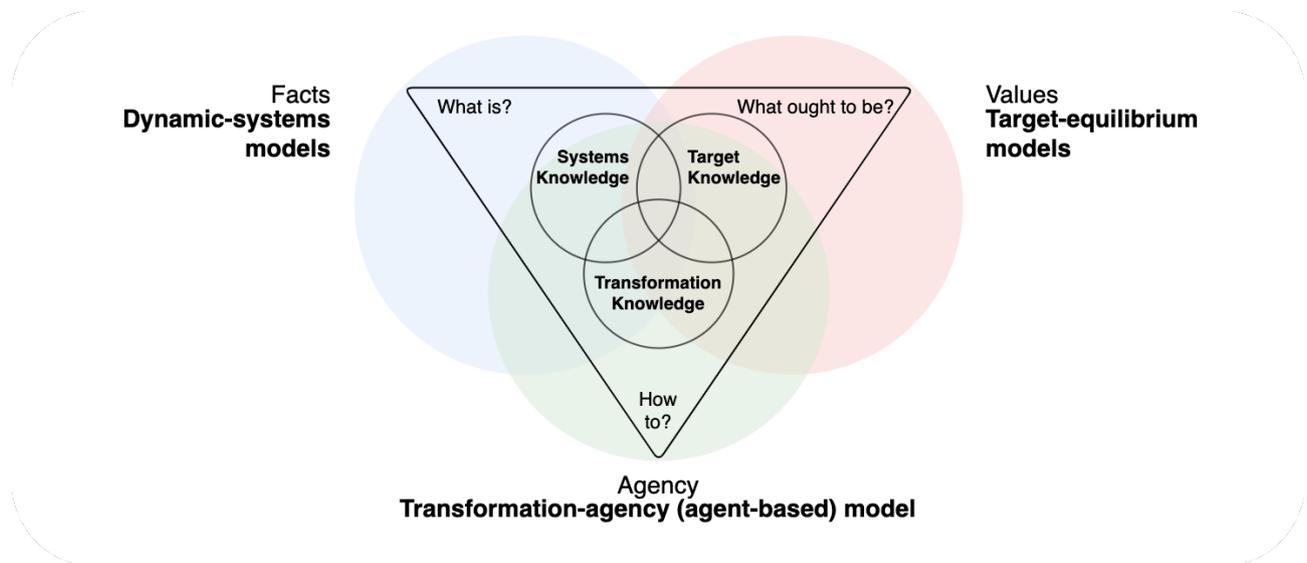
Figure 11.11: Three types of models

References

Albrecht, S. V., Christianos, F., & Schäfer, L. (2024). *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.*

Anderies, J. M., Folke, C., Walker, B., & Ostrom, E. (2013). Aligning Key Concepts for Global Change Policy: Robustness, Resilience, and Sustainability. *Ecology and Society*, *18*(2). https://www.jstor.org/stable/26269292

Anderies, J. M., & Janssen, M. A. (2016). *Sustaining the commons.* Independent. https://dlc.dlib.indiana.edu/dlc/handle/10535/8839

Armstrong McKay, D. I., Staal, A., Abrams, J. F., Winkelmann, R., Sakschewski, B., Loriani, S., Fetzer, I., Cornell, S. E., Rockström, J., & Lenton, T. M. (2022). Exceeding 1.5°C global warming could trigger multiple climate tipping points. *Science*, *377*(6611), eabn7950. https://doi.org/10.1126/science.abn7950

Barfuss, W. (2020). Reinforcement Learning Dynamics in the Infinite Memory Limit. *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, 1768–1770.

Barfuss, W. (2022). Dynamical systems as a level of cognitive analysis of multi-agent learning. *Neural Computing and Applications*, *34*(3), 1653–1671. https://doi.org/10.1007/s00521-021-06117-0

Barfuss, W., Donges, J. F., & Kurths, J. (2019). Deterministic limit of temporal difference reinforcement learning for stochastic games. *Physical Review E*, *99*(4), 043305. https://doi.org/10.1103/PhysRevE.99.043305

Barfuss, W., Donges, J. F., Lade, S. J., & Kurths, J. (2018). When optimization for governing human-environment tipping elements is neither sustainable nor safe. *Nature Communications*, *9*(1), 2354. https://doi.org/10.1038/s41467-018-04738-z

Barfuss, W., Donges, J. F., Vasconcelos, V. V., Kurths, J., & Levin, S. A. (2020). Caring for the future can turn tragedy into comedy for long-term collective action under risk of collapse. *Proceedings of the National Academy of Sciences*, *117*(23), 12915–12922. https://doi.org/10.1073/pnas.1916545117

Barfuss, W., Donges, J., & Bethge, M. (2024). *Ecologically-mediated collective action in commons with tipping elements.* OSF. https://doi.org/10.31219/osf.io/7pcnm

Barfuss, W., Flack, J. C., Gokhale, C. S., Hammond, L., Hilbe, C., Hughes, E., Leibo, J. Z., Lenaerts, T., Levin, S. A., Madhushani Sehwag, U., McAvoy, A., Meylahn, J. M., & Santos, F. P. (2024). Collective Cooperative Intelligence. *Forthcomming in the Proceedings of the National Academy of Sciences.*

Barrett, S. (1994). Self-Enforcing International Environmental Agreements. *Oxford Economic Papers*, *46*(Supplement_1), 878–894. https://doi.org/10.1093/oep/46.Supplement_1.878

Barrett, S. (2005). *Environment and Statecraft: The Strategy of Environmental Treaty-Making.* Oxford University Press. https://doi.org/10.1093/0199286094.001.0001

Barrett, S., & Dannenberg, A. (2012). Climate negotiations under scientific uncertainty. *Proceedings of the National Academy of Sciences*, *109*(43), 17372–17376. https://doi.org/10.1073/pnas.1208417109

Biggs, R., Preiser, R., de Vos, A., Schlüter, M., Maciejewski, K., & Clements, H. (2021). *The Routledge Handbook of Research Methods for Social-Ecological Systems* (1st ed.). Routledge. https://doi.org/10.4324/9781003021339

Boers, N., & Rypdal, M. (2021). Critical slowing down suggests that the western Greenland Ice Sheet is close to a tipping point. *Proceedings of the National Academy of Sciences*, *118*(21), e2024192118. https://doi.org/10.1073/pnas.2024192118

Botvinick, M., Wang, J. X., Dabney, W., Miller, K. J., & Kurth-Nelson, Z. (2020). Deep Reinforcement Learning and Its Neuroscientific Implications. *Neuron*, *107*(4), 603–616. https://doi.org/10.1016/j.neuron.2020.06.014

Brander, J. A., & Taylor, M. S. (1998). The Simple Economics of Easter Island: A Ricardo-Malthus Model of Renewable Resource Use. *The American Economic Review*, *88*(1), 119–138. https://www.jstor.org/stable/116821

Brockmann, D. (2021). *Im Wald vor lauter Bäumen: Unsere komplexe Welt besser verstehen.* Deutscher Taschenbuch Verlag.

Carpenter, S., Walker, B., Anderies, J. M., & Abel, N. (2001). From Metaphor to Measurement: Resilience of What to What? *Ecosystems*, *4*(8), 765–781. https://doi.org/10.1007/s10021-001-0045-9

Constantino, S. M., Schlüter, M., Weber, E. U., & Wijermans, N. (2021). Cognition and behavior in context: A framework and theories to explain natural resource use decisions in social-ecological systems. *Sustainability Science*, *16*(5), 1651–1671. https://doi.org/10.1007/s11625-021-00989-w

Daniel, C. J., Frid, L., Sleeter, B. M., & Fortin, M.-J. (2016). State-and-transition simulation models: A framework for forecasting landscape change. *Methods in Ecology and Evolution*, *7*(11), 1413–1423. https://doi.org/10.1111/2041-210X.12597

Elsawah, S., Filatova, T., Jakeman, A. J., Kettner, A. J., Zellner, M. L., Athanasiadis, I. N., Hamilton, S. H., Axtell, R. L., Brown, D. G., Gilligan, J. M., Janssen, M. A., Robinson, D. T., Rozenberg, J., Ullah, I. I. T., & Lade, S. J. (2020). Eight grand challenges in socio-environmental systems modeling. *Socio-Environmental Systems Modelling*, *2*, 16226–16226. https://doi.org/10.18174/sesmo.2020a16226

Epstein, J. M. (1999). Agent-based computational models and generative social science. *Complexity*, *4*(5), 41–60. https://doi.org/10.1002/(SICI)1099-0526(199905/06)4:5%3C41::AID-CPLX9%3E3.0.CO;2-F

Farahbakhsh, I., Bauch, C. T., & Anand, M. (2022). Modelling coupled human–environment complexity for the future of the biosphere: Strengths, gaps and promising directions. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *377*(1857), 20210382. https://doi.org/10.1098/rstb.2021.0382

Folke, C., Carpenter, S., Walker, B., Scheffer, M., Chapin, T., & Rockström, J. (2010). Resilience Thinking: Integrating Resilience, Adaptability and Transformability. *Ecology and Society*, *15*(4). https://doi.org/10.5751/ES-03610-150420

Garbe, J., Albrecht, T., Levermann, A., Donges, J. F., & Winkelmann, R. (2020). The hysteresis of the Antarctic Ice Sheet. *Nature*, *585*(7826), 538–544. https://doi.org/10.1038/s41586-020-2727-5

Giupponi, C., Ausseil, A.-G., Balbi, S., Cian, F., Fekete, A., Gain, A. K., Essenfelder, A. H., Martínez-López, J., Mojtahed, V., Norf, C., Relvas, H., & Villa, F. (2022). Integrated modelling of social-ecological systems for climate change adaptation. *Socio-Environmental Systems Modelling*, *3*, 18161–18161. https://doi.org/10.18174/sesmo.18161

Goll, D., Heitzig, J., & Barfuss, W. (2024). *Deterministic Model of Incremental Multi-Agent Boltzmann Q-Learning: Transient Cooperation, Metastability, and Oscillations* (arXiv:2501.00160). arXiv. https://doi.org/10.48550/arXiv.2501.00160

Hoffman, M., & Yoeli, E. (2022). *Hidden Games: The Surprising Power of Game Theory to Explain Irrational Human Behaviour.* Hachette UK.

Izquierdo, L. R., Izquierdo, S. S., & Sandholm, W. H. (2024). *Agent-Based Evolutionary Game Dynamics.* https://doi.org/10.5281/zenodo.13938500

Lenton, T. M., Armstrong McKay, D. I., Loriani, S., Abrams, J. F., Lade, S. J., Donges, J. F., Milkoreit, M., Powell, T., Smith, S. R., Zimm, C., Buxton, J. E., Bailey, E., Laybourn, L., Ghadiali, A., & Dyke, J. G. (Eds.). (2023). *The Global Tipping Points Report 2023.* https://global-tipping-points.org

Levin, S., & Xepapadeas, A. (2021). On the Coevolution of Economic and Ecological Systems. *Annual Review of Resource Economics*, *13*(1), 355–377. https://doi.org/10.1146/annurev-resource-103020-083100

Macy, M. W., & Flache, A. (2002). Learning dynamics in social dilemmas. *Proceedings of the National Academy of Sciences*, *99*(suppl_3), 7229–7236. https://doi.org/10.1073/pnas.092080099

Marescot, L., Chapron, G., Chadès, I., Fackler, P. L., Duchamp, C., Marboutin, E., & Gimenez, O. (2013). Complex decisions made simple: A primer on stochastic dynamic programming. *Methods in Ecology and Evolution*, *4*(9), 872–884. https://doi.org/10.1111/2041-210X.12082

Meadows, D. H. (2009). *Thinking in systems: A primer*. Earthscan.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Motesharrei, S., Rivas, J., & Kalnay, E. (2014). Human and nature dynamics (HANDY): Modeling inequality and use of resources in the collapse or sustainability of societies. *Ecological Economics*, *101*, 90–102. https://doi.org/10.1016/j.ecolecon.2014.02.014

Müller, B., Hoffmann, F., Heckelei, T., Müller, C., Hertel, T. W., Polhill, J. G., van Wijk, M., Achterbosch, T., Alexander, P., Brown, C., Kreuer, D., Ewert, F., Ge, J., Millington, J. D. A., Seppelt, R., Verburg, P. H., & Webber, H. (2020). Modelling food security: Bridging the gap between the micro and the macro scale. *Global Environmental Change*, *63*, 102085. https://doi.org/10.1016/j.gloenvcha.2020.102085

Müller-Hansen, F., Cardoso, M. F., Dalla-Nora, E. L., Donges, J. F., Heitzig, J., Kurths, J., & Thonicke, K. (2017). A matrix clustering method to explore patterns of land-cover transitions in satellite-derived maps of the Brazilian Amazon. *Nonlinear Processes in Geophysics*, *24*(1), 113–123. https://doi.org/10.5194/npg-24-113-2017

Nowak, M. A. (2006). Five Rules for the Evolution of Cooperation. *Science*. https://doi.org/10.1126/science.1133755

Ostrom, E. (1990). *Governing the commons: The evolution of institutions for collective action*. Cambridge university press.

Ostrom, E., Dietz, T., Dolšak, N., Stern, P. C., Stonich, S., & Weber, E. U. (Eds.). (2002). *The drama of the commons*. National Academies Press. http://www.nap.edu/catalog/10287

Page, S. E. (2018). *The model thinker: What you need to know to make data work for you*. Basic Books.

Polasky, S., Carpenter, S. R., Folke, C., & Keeler, B. (2011). Decision-making under great uncertainty: Environmental management in an era of global change. *Trends in Ecology & Evolution*, *26*(8), 398–404. https://doi.org/10.1016/j.tree.2011.04.007

Raworth, K. (2017). *Doughnut economics: Seven ways to think like a 21st-century economist*. Chelsea Green Publishing.

Reyers, B., Moore, M.-L., Haider, L. J., & Schlüter, M. (2022). The contributions of resilience to reshaping sustainable development. *Nature Sustainability*, 1–8. https://doi.org/10.1038/s41893-022-00889-6

Rockström, J., Gaffney, O., Rogelj, J., Meinshausen, M., Nakicenovic, N., & Schellnhuber, H. J. (2017). A roadmap for rapid decarbonization. *Science*. https://doi.org/10.1126/science.aah3443

Sayama, H. (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)

Scheffer, M., Carpenter, S., Foley, J. A., Folke, C., & Walker, B. (2001). Catastrophic shifts in ecosystems. *Nature*, *413*(6856), 591–596. https://doi.org/10.1038/35098000

Schill, C., Anderies, J. M., Lindahl, T., Folke, C., Polasky, S., Cárdenas, J. C., Crépin, A.-S., Janssen, M. A., Norberg, J., & Schlüter, M. (2019). A more dynamic understanding of human behaviour for the Anthropocene. *Nature Sustainability*, *2*(12), 1075–1082.

Schlüter, M., Baeza, A., Dressler, G., Frank, K., Groeneveld, J., Jager, W., Janssen, M. A., McAllister, R. R. J., Müller, B., Orach, K., Schwarz, N., & Wijermans, N. (2017). A framework for mapping and comparing behavioural theories in models of social-ecological systems. *Ecological Economics*,

131, 21–35. https://doi.org/10.1016/j.ecolecon.2016.08.008

Schultz, W., Stauffer, W. R., & Lak, A. (2017). The phasic dopamine signal maturing: From reward via behavioural activation to formal economic utility. *Current Opinion in Neurobiology*, *43*, 139–148. https://doi.org/10.1016/j.conb.2017.03.013

Smaldino, P. E. (2017). Models Are Stupid, and We Need More of Them. In R. R. Vallacher, S. J. Read, & A. Nowak (Eds.), *Computational Social Psychology* (1st ed., pp. 311–331). Routledge. https://doi.org/10.4324/9781315173726-14

Steffen, W., Broadgate, W., Deutsch, L., Gaffney, O., & Ludwig, C. (2015). The trajectory of the Anthropocene: The Great Acceleration. *The Anthropocene Review*, *2*(1), 81–98. https://doi.org/10.1177/2053019614564785

Steffen, W., Rockström, J., Richardson, K., Lenton, T. M., Folke, C., Liverman, D., Summerhayes, C. P., Barnosky, A. D., Cornell, S. E., Crucifix, M., Donges, J. F., Fetzer, I., Lade, S. J., Scheffer, M., Winkelmann, R., & Schellnhuber, H. J. (2018). Trajectories of the Earth System in the Anthropocene. *Proceedings of the National Academy of Sciences*, *115*(33), 8252–8259. https://doi.org/10.1073/pnas.1810141115

Sterman, J. D., & Sweeney, L. B. (2007). Understanding public complacency about climate change: Adults' mental models of climate change violate conservation of matter. *Climatic Change*, *80*(3), 213–238. https://doi.org/10.1007/s10584-006-9107-5

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second edition). The MIT Press.

Williams, B. K. (2009). Markov decision processes in natural resources management: Observability and uncertainty. *Ecological Modelling*, *220*(6), 830–840. https://doi.org/10.1016/j.ecolmodel.2008.12.023

Part IV

Exercises

Ex | Introduction to Python

This exercises serves as a basic introduction to Python for Scientific Computing.

As with any language, you must speak it to learn it. This holds true for Python as well. In the following, we will cover the basics of Python to get us started [1]. But **expect to keep learning** ways to express your ideas in Python.

Set up

This text is written in a computer file called Jupyter Notebook, typically with the extension `.ipynb`. Jupyter is an interactive platform where you can write code and text and make visualizations. To *run* the notebook, i.e., to execute the Python code inside, you need to connect the notebook to a so-called *Python kernel* or *runtime*. There are two options: you run the kernel locally on your computer or *in the cloud* on the internet.

Cloud-based runtime

The easiest way to run a Jupyter Notebook is to use Google Colab, which requires a Google account. The advantage is that you don't need to worry about installing anything on your machine — other notable cloud options include binder, which is, however, more cumbersome to set up. At Uni Bonn, there is also the option to launch a Jupyter lab via *ecampus*.

Local runtime

The alternative to an internet-based execution of the Python code inside a Jupyter Notebook is to install Python locally on your machine. I recommend doing so. If you are new to Python, we recommend downloading the Anaconda installer and following the installation instructions. Once installed, we'll use the Jupyter Notebook or Jupyter Lab interface to write code.

Exercise

Execute the following code cell:

```
print("Hello, World!")
```

```
Hello, World!
```

---

[1] See also 1 Introduction to Python and Jupyter Notebooks and 01.01-Getting-Started-with-Python-and-Jupyter-Notebooks, which served as valuable sources for this tutorial.

Note that within a Jupyter Notebook, you don't need the `print` function if you simply want to see the output of a cell's last line.

```
"Hello Jupyter"
```

```
'Hello Jupyter'
```

The `print` function is useful for viewing the output of multiple lines.

```
print("Hello Alice?")
print("Hello Bob")
```

```
Hello Alice?
Hello Bob
```

```
"Hello Alice?"
"Hello Bob"
```

```
'Hello Bob'
```

Getting help

A convenient way to get help on almost anything in Python is to add a `?` behind the Python object. Try it with the `print` function in the cell below.

```
# ...
```

Also, be not afraid of Python errors. They often give you handy tips on how to resolve a problem.

Pro tip. Familiarize yourself with the keyboard shortcuts for a more convenient notebook experience, e.g., execute a code cell and advance by pressing `Shift+Enter`, use `a` for inserting a code cell above, `b` for below, `dd` for deleting a cell, `z` for undoing your last move, `Enter` for entering into a cell, `Esc` for leaving a cell, `m` for changing a cell to a Markdown cell, and `y` for changing it back to a code cell.

Basics

Everything in Python is an object. Every number, string, data structure, function, class, module, etc., exists in the Python interpreter as a Python object. An object may have attributes and methods associated with it. For example, let us define a variable that stores an string:

```
var = "Alice"
```

Exercise

A convenient way to see which methods and attributes are associated with an object is to use tab completion. Write `var.` and press TAB in the cell below.

```
# ...
```

Transform the text in the variable `var` to uppercase. Use tab-completion to find the appropriate method.

```
# ...
```

Comments

In Python code cells, comments begin with `#`. They serve to clarify your code. Nonetheless, it's best not to use them excessively. Your code should ideally be clear and comprehensible on its own, without the need for excessive comments.

Exercise   Write a comment in the cell below.

```
# ...
```

Variables

A variable is created as soon as a value is assigned to it. We don't have to define the type of the variable explicitly, as we do in other programming languages, because Python can automatically guess the type of data entered (dynamically typed). For example,

```python
x = 5
y = 3.14
name = "Alice"
is_student = True
```

You can assign values to several variables simultaneously by using commas to separate the variable names and their corresponding values.

```python
color1, color2, color3 = "red", "green", "blue"
```

You can assign the same value to several variables by chaining multiple assignment operations in one statement.

```python
color4 = color5 = color6 = "magenta"
```

Any data or information stored within a Python variable has a *type*.

Exercise

View the data type stored within each variable form above using the `type` function.

```
# ...
```

Rules for variable name

Variable names can be short (`a`, `x`, `y`, etc.) or descriptive (`my_favorite_color`, `profit_margin`, `the_3_musketeers`, etc.). However, I recommend using descriptive variable names to make it easier to understand the code.

The rules below must be followed while naming Python variables: * A variable's name must start with a letter or the underscore character `_`. It cannot begin with a number. * A variable name can only contain lowercase (small) or uppercase (capital) letters, digits, or underscores (`a-z`, `A-Z`, `0-9`, and `_`). * Variable names are case-sensitive, i.e., `a_variable`, `A_Variable`, and `A_VARIABLE` are all different variables.

DeepDive | Call by Reference

Python employs a mechanism called *Call by Object Reference.* When an object is assigned to a variable name, the variable acts as a reference to that object. For instance, consider the following assignment:

```python
x = [5, 3]
```

Here, the variable name `x` points to the memory location of the object `[5, 3]`. Now, if we assign `x` to a new variable `y`:

```python
y = x
```

In this case, the variable `y` now also points to the same object `[5, 3]`. This means the object `[5, 3]` is **not** duplicated in a different memory location for `y`. To illustrate this, let's add an element to `y`:

```python
y.append(4)
y
```

```
[5, 3, 4]
```

```python
x
```

```
[5, 3, 4]
```

When we modified `y`, it is important to note that `x` also updated to the same object. This demonstrates that `x` and `y` refer to the same object rather than independent copies.

Basic Operations

Python includes fundamental arithmetic operations by default. Below are a few examples. Specifically, the `**` operator is used for exponentiation.

```
a = 12
b = 2

print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(a ** b)
print(a % b) # modulo operator
```

```
14
10
24
6.0
144
0
```

Data structures

Python includes various built-in containers or data structures [2] for data storage.

Lists

A list is a sequence of Python objects with two key characteristics

1. the number of objects is variable, i.e., objects can be added or removed from a list, and
2. the objects are mutable, i.e., they can be changed.

Lists can be defined as a sequence of Python objects separated by commas and enclosed in square brackets []. For example, below is a list consisting of three integers.

```
list_example = [4,7,3,5,7,1,5,87,5]
```

We can check the data type of a Python object using the in-built function `type()`. Let us check the data type of the object `list_example`.

```
type(list_example)
```

```
list
```

Indexing   Indexing means accessing the elements of a list by their respective position. Indexing in lists includes both

- positive indexing (starting from 0 for the first element) and
- negative indexing (starting from -1 for the last element).

For example,

---

[2]see 2 Data structures – Introduction to Data Science with Python for a more detailed exposition.

```
list_example[0]
```

4

```
list_example[-1]
```

5

Slicing a list   List slicing is a technique in Python that allows you to extract a portion of a list by specifying a range of indices. It creates a new list containing the elements from the original list within that specified range. List slicing uses the colon : operator to indicate the `start`, `stop`, and `step` values for the slice. The general syntax is:

```
new_list = original_list[start:stop:step]
```

Here's what each part of the slice means:

- `start`: The index at which the slice begins (inclusive). If omitted, it starts from the beginning (index 0).
- `stop`: The index at which the slice ends (exclusive). If omitted, it goes until the end of the list.
- `step`: The interval between elements in the slice. If omitted, it defaults to 1.

Exercise . Try it out.

```
# ...
```

Adding and removing elements in a list   For removing an element from the list, the `pop` and `remove` methods may be used. The `pop` method removes an element at a particular index, while the `remove` method removes the element's first occurrence in the list by its value.

Exercise . Try it out.

```
# ...
```

List comprehensions   List comprehensions offer a clear and concise method to generate new lists by applying an expression to each element in an iterable (such as a list or range) and can filter items based on a specified condition. The fundamental syntax for a list comprehension is as follows:

```
new_list = [expression for item in iterable if condition]
```

- `expression`: This is the expression that is applied to each item in the iterable. It defines what will be included in the new list.
- `item`: This is a variable that represents each element in the iterable as the comprehension iterates through it.
- `iterable`: This is the source from which the elements are taken. It can be any iterable, such as a list, range, or other iterable objects.

266

- **condition** (optional): This filter can be applied to control which items from the iterable are included in the new list. If omitted, all items from the iterable are included.

For example, to create a list of even numbers from 1 to 20, we can write

```
[x for x in range(1, 21) if x % 2 == 0]
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Exercise . Use a list comprehension to create a list that has squares of natural numbers from 5 to 15.

```
# ...
```

Tuples

A tuple is a sequence of Python objects with two key characteristics:

1. the number of objects is fixed, and
2. the objects are immutable, i.e., cannot be changed.

A tuple can be defined as a sequence of Python objects separated by commas and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

We can check the data type of a Python object using the in-built function `type()`. Let us check the data type of the object `tuple_example`.

```
type(tuple_example)
```

```
tuple
```

Tupe indexing is identical to list indexing.

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked, and each variable will be assigned a value in the order in which it appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
print(x)
print(y)
print(z)
```

```
4.5
this is a string
('Nested tuple', 5)
```

267

Dictionaries

Unlike lists and tuples, a dictionary is an unordered collection of items. Each item stored in a dictionary has a key and value. You can use a key to retrieve the corresponding value from the dictionary. Dictionaries have the type `dict`.

Dictionaries are created by enclosing key-value pairs within braces or curly brackets { and }, colons to separate keys and values, and commas to separate dictionary elements.

The dictionary keys and values are Python objects. While values can be any Python object, keys need to be immutable Python objects, like strings, integers, tuples, etc. Thus, a list can be a value, but not a key, as elements of a list can be changed.

```python
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping'}
```

Elements of a dictionary can be retrieved by using the corresponding key.

```python
dict_example['India']
```

```
'Narendra Modi'
```

```python
dict_example.keys()
```

Viewing keys and values

```
dict_keys(['USA', 'India', 'China'])
```

```python
dict_example.values()
```

```
dict_values(['Joe Biden', 'Narendra Modi', 'Xi Jinping'])
```

```python
dict_example.items()
```

```
dict_items([('USA', 'Joe Biden'), ('India', 'Narendra Modi'), ('China', 'Xi Jinping')])
```

Adding and removing elements in a dictionary New elements can be added to a dictionary by defining a key in square brackets and assigning it to a value:

```python
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```python
del dict_example['Countries']
```

```python
dict_example
```

```python
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```python
dict_example.pop('USA')
```

```python
'Joe Biden'
```

```python
dict_example
```

```python
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan': 'Fumio Kishida'}
```

New elements can be added, and values of existing keys can be changed using the `update` method:

```python
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi
↪  Jinping','Countries':3}
dict_example
```

```python
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```python
dict_example.update({'Countries': dict_example['Countries']+1, 'Japan':'Fumio
↪  Kishida'})
dict_example
```

```python
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

Functions

If a piece of code or an algorithm is utilized multiple times within a program, it can be defined separately as a function. Doing so improves both the code structure and clarity.

To create a new function, use the `def` keyword. For instance,

269

```python
def greet(name):
    return f"Hello, {name}!"
```

Here, `name` is called an argument or parameter of the function.

Indentation: Python heavily utilizes indentation (white space preceding a statement) to establish code structure. This feature enhances readability and comprehension of Python code. Improper indentation can lead to issues. To indent your code, position the cursor at the beginning of the line and press the `Tab` key once. Pressing `Tab` again will create an additional indent, while `Shift+Tab` will decrease the indentation by one level.

```python
greet("Alice")
```

```
'Hello, Alice!'
```

```python
greet("Bob")
```

```
'Hello, Bob!'
```

Variable scope

When defining functions, understanding variable scope is crucial.

Local Variable. Variables declared within a function have a local scope, meaning they can only be accessed inside that function. These are referred to as local variables.

Global Variable. Conversely, a variable declared outside of any function is called a global variable. This means a global variable can be accessed both inside and outside the function.

Named and optional arguments

Calling a function with numerous arguments can be confusing and lead to mistakes. Python allows the use of named arguments for improved clarity. Additionally, you have the option to distribute the function call across several lines.

```python
def create_profile(first_name, last_name,
                   age=None, email=None, phone=None, address=None, city=None,
 ↪   state=None, zip_code=None,
                   country="Germany"):
    """
    Creates a user profile with the given information. Some fields have default
 ↪   values.
    """
    print(f"First Name: {first_name}")
    print(f"Last Name: {last_name}")
    print(f"Age: {age}")
    print(f"Email: {email}")
    print(f"Phone: {phone}")
    print(f"Address: {address}")
```

```
    print(f"City: {city}")
    print(f"State: {state}")
    print(f"Zip Code: {zip_code}")
    print(f"Country: {country}")
```

Here, all arguments except for the first two are optional. You can invoke the function with or without these arguments; if no arguments are provided, the default value will be applied. Functions that include optional arguments provide greater flexibility in their usage.

```
user_profile = create_profile(
    "Jane",
    last_name="Doe",
    email="jane.doe@example.com",
    city="Bonn")
```

```
First Name: Jane
Last Name: Doe
Age: None
Email: jane.doe@example.com
Phone: None
Address: None
City: Bonn
State: None
Zip Code: None
Country: Germany
```

DeepDive | *args and **kwargs

Using special symbols, we can pass a variable number of arguments to a function. There are two special symbols:

1. *args (Non-Keyword Arguments)
2. *kwargs (Keyword Arguments)

```
def myFun(name, *args, **kwargs):
    print("Name:", name)
    print("args: ", args)
    print("kwargs: ", kwargs)
```

Now we can use both ***args** and ****kwargs** to pass arguments to this function

```
myFun('John',22,'cs',city="Bonn",age=22,major="cs")
```

```
Name: John
args:  (22, 'cs')
kwargs:  {'city': 'Bonn', 'age': 22, 'major': 'cs'}
```

Control flow

Similar to other programming languages, Python includes built-in keywords that facilitate conditional control flows in code.

Conditional statements

One of the most powerful features of programming languages is branching, which enables making decisions and running alternative statements depending on whether specific conditions are true. In Python, we can use the `if`, `else`, and `elif` (short for *else if*) keywords. For example,

```python
def what_is_x(x):
    if x > 0:
        print("x is positive")
    elif x > -10:
        print("x is negative")
    else:
        print("x is below -10")
```

```python
what_is_x(x=-9)
```

```
x is negative
```

Iteration (Loops)

Another powerful feature of programming languages, closely associated with branching, is the ability to repeatedly execute one or more statements. This capability is commonly known as iteration or looping. There are two primary methods to achieve this in Python: `for` loops and `while` loops.

For loops   A `for` loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and ranges. For example,

```python
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for day in days:
    print(day)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
```

The `range` function is used to create a sequence of numbers that can be iterated over using a `for` loop. It can be used in 3 ways:

- `range(n)` - Creates a sequence of numbers from 0 to `n-1`
- `range(a, b)` - Creates a sequence of numbers from `a` to `b-1`

- `range(a, b, step)` - Creates a sequence of numbers from `a` to `b-1` with increments of step

Exercise . Try it out.

```
# ...
```

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

```python
a_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for i in range(len(a_list)):
    print('The value at position {} is {}.'.format(i, a_list[i]))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

Here, the `format` method of the string inserts its arguments at the positions where the `{}`s serve as a placeholder.

Another way to achieve the same result is by using the `enumerate` function with `a_list` as an input, which returns a tuple containing the index and the corresponding element.

```python
for i, val in enumerate(a_list):
    print('The value at position {} is {}.'.format(i, val))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

While loops   A `while` loop executes a block of code for as long as a condition is true. For example, let's try to calculate the factorial of 100 using a while loop. The factorial of a number `n` is the product (multiplication) of all the numbers from 1 to `n`, i.e., `1*2*3*...*(n-2)*(n-1)*n`.

```python
result = 1; i = 1 # We can define multiple variables in one line by separating them
↪   with a ;

while i <= 100:
    result = result * i
    i = i+1

'The factorial of 100 is: {}'.format(result)
```

```
'The factorial of 100 is: 93326215443944152681699238856266700490715968264381621468592963895217
```

273

Be aware of **infinite loops**. Suppose the condition in a while loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is stuck in an infinite loop during execution, just press the "Stop" button on the toolbar (next to "Run") or select "Kernel > Interrupt" from the menu bar. This will interrupt the execution of the code. The following two cells both lead to infinite loops and must be interrupted.

DeepDive | `break` and `continue` statement

These statements alter the flow of a loop.

We can use the `break` statement within the loop's body to immediately stop the execution and break out of the loop.

```
i = 1
result = 1

while i <= 100:
    result *= i
    if i == 42:
        print('Magic number 42 reached! Stopping execution..')
        break
    i += 1

print('i:', i)
print('result:', result)
```

```
Magic number 42 reached! Stopping execution..
i: 42
result: 1405006117752879898543142606244511569936384000000000
```

With the `continue` statement, we break out of the current iteration and continue to the next one.

```
i = 1
result = 1

while i <= 10:
    i += 1
    if i % 2 == 0:
        print('Skipping {}'.format(i))
        continue
    print('Multiplying with {}'.format(i))
    result = result * i

print('i:', i)
print('result:', result)
```

```
Skipping 2
Multiplying with 3
Skipping 4
```

```
Multiplying with 5
Skipping 6
Multiplying with 7
Skipping 8
Multiplying with 9
Skipping 10
Multiplying with 11
i: 11
result: 10395
```

Libraries

There are several built-in functions in Python like `print()`, `abs()`, `max()`, `sum()` etc., which do not require importing any library. We can extend these functions by importing external libraries. Here, we will cover some popular libraries for basic scientific computing.

NumPy

NumPy is a fundamental library for **numerical computing** in Python. It provides support for arrays, matrices, and mathematical functions, making it essential for scientific and data analysis tasks. It is mostly used for performing numerical operations and efficiently storing numerical data.

```python
import numpy as np
```

```python
vector = np.array([1, 2, 3])
matrix = np.array([[3, 1],
                   [0, 2]])
print(vector)
print(matrix)
```

```
[1 2 3]
[[3 1]
 [0 2]]
```

SciPy

SciPy is used for performing scientific computing such as solving differential equations, optimization, statistical tests, etc. In particular, its eigenvalue computation method seems to be more stable than NumPy's.

```python
import scipy as sp
```

Exercise : Eigenvector and values

Use the `scipy.linalg` module to calculate the eigenvalues and eigenvectors of the `matrix` defined above. If you need to refresh your knowledge about eigenvectors and eigenvalues, I recommend watching this video by 3Blue1Brown. Use your knowledge from above on how to get help inside a Python Jupyter Notebook.

```
# ...
```

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, or interactive plots and visualizations. It is commonly used for data visualization and exploration.

```
import matplotlib.pyplot as plt
```

For example, we can create a scatter plot of 1000 random sample, generate from the `numpy.random` module.

```
np.random.seed(42) # The seed makes the result reproducible
plt.scatter(np.random.randn(1000), np.random.randn(1000));
```
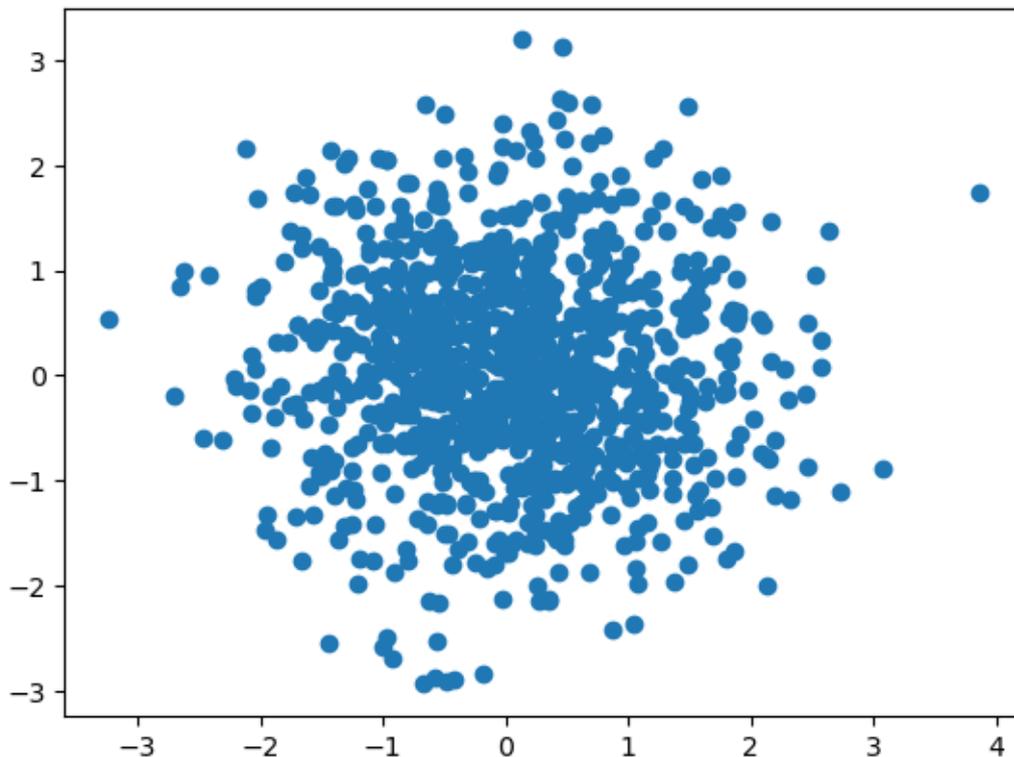

Sympy

SymPy is a Python library for symbolic mathematics, providing capabilities to perform algebraic manipulations, solve equations, and work with calculus, among other mathematical tasks. It is a

computer algebra system written in pure Python, designed to be easily extensible and to provide a comprehensive set of tools for symbolic computation.

```
import sympy as sym
```

For example, we can calculate the eigenvectors and values from the matrix above also computer-analytically

```
eigenvalues_and_vectors = sym.Matrix(matrix).eigenvects()

print("The eigenvalues are {} and {}".format(eigenvalues_and_vectors[0][0],
↪   eigenvalues_and_vectors[1][0]))
print("The eigenvalues are {} and {}".format(eigenvalues_and_vectors[0][2][0],
↪   eigenvalues_and_vectors[1][2][0]))
```

```
The eigenvalues are 2 and 3
The eigenvalues are Matrix([[-1], [1]]) and Matrix([[1], [0]])
```

Another use case is to differentiate a function using SymPy in Python using the `diff` function. Here's an example:

```
# Define the variable
x = sym.symbols('x')

# Define the function
f = x**3 + 2*x**2 + x + 1

# Differentiate the function with respect to x
sym.diff(f, x)
```

$3x^2 + 4x + 1$

Recap | Eigenvectors and eigenvalues

DeepDive | Recap EEV

To represent a vector (x) using the eigenvectors of a matrix (A) as a basis, follow these steps:

1. Eigenvalue and Eigenvector Decomposition: If (A) is a square matrix, and it has a full set of linearly independent eigenvectors, we can decompose it as: [A = V V^{-1}] where: - (V) is the matrix whose columns are the eigenvectors of (A), - ( ) is the diagonal matrix containing the corresponding eigenvalues of (A), - (V^{-1}) is the inverse of (V) (assuming the eigenvectors are linearly independent).

2. Using Eigenvectors as a Basis: The eigenvectors of (A) form a basis for the vector space. To represent (x) in this new basis, express (x) as a linear combination of the eigenvectors of (A).

Let (v_1, v_2, ..., v_n) be the eigenvectors of (A). You want to express: [x = c_1 v_1 + c_2 v_2 + ...+ c_n v_n] where (c_1, c_2, ..., c_n) are the scalar coefficients that correspond to the projection of (x) onto each eigenvector (v_i).

3. Find the Coefficients (c): If (V) is the matrix whose columns are the eigenvectors of (A), then you can find the coefficients (c) by solving the following equation: [x = V c] where (c = [c_1, c_2, …, c_n]^T).

To solve for (c), multiply both sides by (V^{-1}): [c = V^{-1} x] Thus, the coefficients (c) represent the vector (x) in the eigenvector basis.

4. Reconstructing (x) from the Eigenvector Basis: Once you have the coefficients (c), you can reconstruct (x) as: [x = V c] where (V) is the matrix of eigenvectors and (c) is the vector of coefficients.

Summary: To represent (x) in the eigenvector basis of (A): 1. Compute the matrix (V) of eigenvectors of (A). 2. Find the coefficients (c = V^{-1} x). 3. (x) can then be written as (x = V c), where (c) represents (x) in the eigenvector basis.

Exercise : Sustainability Systems Science Generator

There are many word combinations in the literature that broadly refer to the kind of science we will be exploring in this course. Take, for example, the word combination,

Coupled Social-Ecological Systems Modeling.

Your task is to write a Python function that randomly generates alternative names for this kind of science. Tip: Start by considering alternatives for each part of the word combination above.

```
# ...
```

Saving notebooks

The easiest way to save a notebook as a PDF file for sharing is to print it via your browser's print dialogue.

Ex | Nonlinearity

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

In this exercise, we will investigate the questions when feedback and overshoot lead to prolonged oscillations and when they do not.

Model 1 | Human-Nature interactions

In the lecture, we asked the question, whether the osscilations we observed in the interaction model between human economic captial (y) and natural capital (x),

$$x_{t+1} = x_t - ay_t \tag{.14}$$
$$y_{t+1} = y_t + bx_t \tag{.15}$$

are due to a special set of parameters (a and b), or whether they are a general feature of the system's structure.
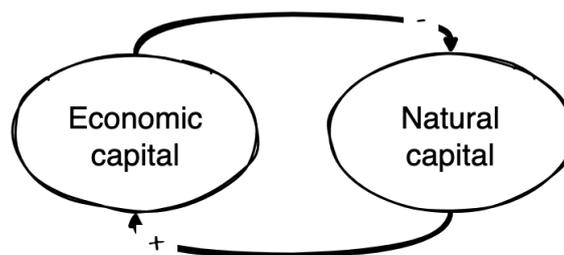
Figure 12: 02.01-EconomyNature.dio.png

Now, you will use Python's `sympy` library to investigate this question. `sympy` is a powerful library for symbolic mathematics. It allows you to define variables and equations symbolically, and then solve these equations symbolically. This is very useful for investigating the properties of mathematical models.

```
import sympy as sp
```

First, we define the symbols for the variables in the model:

```
x, y, a, b = sp.symbols('x y a b')
```

Step 1: Coefficient matrix

Formulate a `sympy.Matrix` object `A` that represents the coefficient matrix of the system of equations, $\mathbf{x}_{t+1} = A\mathbf{x}_t$.

```
# ...
```

Step 2: Calculate the eigenvalues of the matrix A

Now, calculate the eigenvalues of the matrix `A` using the `A.eigenvals()` method.

```
# ...
```

Step 3: Interpret the results

What do the eigenvalues tell you about the stability of the system? Are the oscillations in the system due to a special set of parameters, or are they a general feature of the system's structure? Write your answer in the markdown cell below.

…

As discussed in the lecture, the fact that economic and natural captial may enter negative values is not very intuitive. Therefore, we will now refine the model, turning it into a model with nonlinear changes.

Model 2 | Lotka-Volterra equations

During this part, you will use Python to implement and investigate the Lotka-Volterra equations, also known as the predator-prey model. The predator-prey equations are an iconic model in population ecology, which describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. It is the foundation for many dynamic system models of human-enviornment interactions, where human societies are the predators and the natural environment the prey. The Easter Island model (Brander & Taylor, 1998) or the HANDY (human and nature dynamics) model (Motesharrei et al., 2014) are prototypical examples of such models; (see also Section 2 of the review by Farahbakhsh et al., 2022).

In discrete time, the Lotka-Volterra equations read

$$\Delta x = x_{t+1} - x_t = \alpha x_t - \beta x_t y_t \tag{.16}$$
$$\Delta y = y_{t+1} - y_t = \delta x_t y_t - \gamma y_t \tag{.17}$$

where, in our case, x denotes the health of the natural environment and y the level of development of a human society. The parameters α, β, δ, and γ are positive constants that determine the dynamics of the system. The parameter α represents the natural growth rate of the environment, β the rate at which the human society depletes the environment, δ the rate at which the human society grows by exploiting the environment, and γ the natural decay rate of the human society.

Model 2 | The Lotka-Volterra equations

You will use the following set of parameters as default values:

```
# Parameters
alpha = 0.1  # Growth rate of prey
beta = 0.02  # Rate at which predators destroy prey
gamma = 0.3  # Death rate of predators
delta = 0.01 # Rate at which predators increase by consuming prey

# Initial conditions
X = 40  # Initial prey population
Y = 9   # Initial predator population
```

Step 1 | Implement the model

Write a function called `lotkavolterra` that implements the update of the Lotka-Volterra equations.

```
# ...
```

Write (or copy-paste-and-adjust) an `iterate_model` function that iterates a dynamic systems model forward in time, given an initial state and a function (plus its parameters) that updates the state.

```
# ...
```

Write (or copy-paste-and-adjust) a `plot_stock_evolution` function that plots the time series of the state variables. Label the two-dimensional output of our system as "Natural environment" and "Human society".

```
# ...
```

Step 2 | Plot the time series

Visualize the time series of the natural environment and the human society for the default parameter values for 100 time steps.

```
# ...
```

You should observe oscillations in the time series of the natural environment and the human society. Crucially, natural environment and human socitey values (almost) never become negative.

Optional | Feel free to interact with the time series plot using the `interact` function.

```
# ...
```

Step 3 | Visualize the dynamics in the phase space

Adjust the `plot_flow` function from the lecture material to visualize the dynamics of the system in the phase space. The updated `plot_flow` function should take the following arguments (and default values):

- `update_func`: the function that updates the state of the system
- `xextent=10`: the x-axis extent of the phase space plot
- `yextent=10`: the y-axis extent of the phase space plot
- `nr_points=11`: the number of points in the phase space plot
- `ax=None`: the axis object to plot on
- `**update_params`: additional keyword arguments for the `update_func`

```
# ...
```

Visualize the dynamics of the Lotka-Volterra equations in the phase space for the default parameter values and an `xextent` of 70.

```
# ...
```

Now, adjust the `plot_flow_trajectory` function from the lecture material to visualize the trajectory of the system in the phase space next to the time series plot. The updated `plot_flow_trajectory` function should take the following arguments (and default values):

- `nr_timesteps`: the number of time steps to simulate
- `initial_value`: the initial value of the state variables
- `update_func`: the function that updates the state of the system
- `xextent=10`: the x-axis extent of the phase space plot
- `yextent=10`: the y-axis extent of the phase space plot
- `nr_points=11`: the number of points in the phase space plot
- `**update_params`: additional keyword arguments for the `update_func`

```
# ...
```

Visualize a trajectory from the initial values ($x_0 = 28, y_0 = 5$) of the system in the phase space next to the time series plot for the default parameter values and 250 time steps. The `xextent` should be 70.

```
# ...
```

You should observe that the system diverges with ossiclations around an equilibrium point that are more complex than simple sin and cosin fuctions.

Optional | Feel free to interact with plot using the `interact` function.

```
# ...
```

But where exactly are the equilibirum points of the system?

Step 4 | Equilibrium points

Calculate the analytical solutions for the equilibrium points x_e of the Lotka-Volterra system, i.e., how x_e and y_e depend on the parameters α, β, δ, and γ.

You can do this either by hand or using the `sympy` library.

```
# ...
```

You should have found two equilibrium points. Include both in the visualization of the phase space trajectory from the initial values $(x_0 = 28, y_0 = 5)$ for the default parameter values and 250 time steps. The `xextent` should be 70.

```
# ...
```

Step 5 | Derive the Jacobian matrix

The Jacobian matrix of the descrete-time system $(\mathbf{x}_{t+1} = F(\mathbf{x}_t))$ is the multidimensional version of the derivative. It is a matrix of all first-order partial derivatives of a vector-valued function. The Jacobian matrix of a two-dimensional system is given by

$$J = \begin{bmatrix} \frac{\partial F_x}{\partial x} & \frac{\partial F_x}{\partial y} \\ \frac{\partial F_y}{\partial x} & \frac{\partial F_y}{\partial y} \end{bmatrix}, \tag{.18}$$

where F_x and F_y are the two functions that describe the dynamics of the x and y component of the system.

You can do this either by hand or using the `sympy` library.

…

$$J = \begin{bmatrix} 1 + \alpha - \beta y & -\beta x \\ \delta y & 1 + \delta x - \gamma \end{bmatrix}. \tag{.19}$$

Step 6 | Eigenvalues at the equilibrium points

Compute the eigenvalues of the Jacobian matrix at the two equilibrium points using the symbolic mathematics library `sympy`.

```
# ...
```

Similarly as in the first model, you should observe that for positive parameter values, you always have to take the square root of a negative number, leading to complex numbers which indicates that the system oscillates around the equilibrium points.

In the basic Lotka-Volterra model, the natural environment would grow indefinitely without human society, which is not very realistic. Therefore, we will now refine the model, make the growth of the natural environment logistic.

Model 3 | Extended Lotka-Volterra model

We extent the Lotka-Volerra model such that the natural environment has a finite carrying capacity C and its natural growth follows the logistic map,

$$\Delta x = x_{t+1} - x_t = \alpha x_t(1 - \frac{x_t}{C}) - \beta x_t y_t, \tag{.20}$$

$$\Delta y = y_{t+1} - y_t = \delta x_t y_t - \gamma y_t. \tag{.21}$$

All other parameters remain the same.

Step 1 | Implement the model

Write a function called `lotkavolterraX` that implements the update of the extended Lotka-Volterra equations.

```
# ...
```

Step 2 | Visualize the dynamics in the phase space

For the default parameters and a carrying capacity of $C = 40$, visualize a trajectory of the dynamics of the extended Lotka-Volterra equations in the phase space, using the `plot_flow_trajectory` function from above. The initial values should be $x_0 = 40$ and $y_0 = 0.001$, representing a natural environment in equilibirum and a human society that is just starting to grow. Simulate the system for 250 time steps. Set the `xextent` to 50, and the `yextent` to 3.

```
# ...
```

Briefly describe and interpret the dynamics you observe in the markdown cell below.

...

Last, plot the evolution of the human society trajectory alone. What do you observe?

```
# ...
```

Optional step | Eigenvalues at the equilibrium points

Use `sympy` to calculate the eigenvalues at the equilibirum points. What do they tell about the stability of the system?

```
# ...
```

Ex | Tipping elements

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

Robustness of the tipping elements model

In this exercise, we will investigate the robustness of the tipping elements model. We will do this changing the functional forms of the reinforcing and balancing feedback loops.
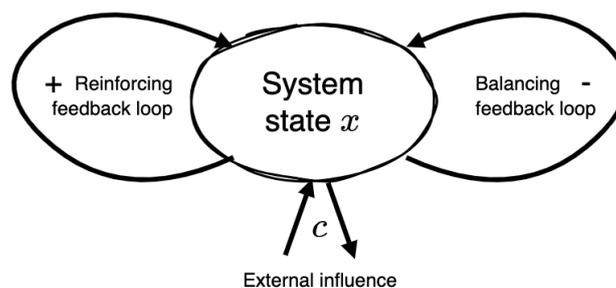
Figure 13: 02.02-TippingModel.dio.png

$$\Delta x = (x^3 - ax^5 + c)\frac{1}{\tau},$$

where, as in the model of the lecture, τ represents the typical time scale of the system, and thus, inverse strength of the system's change, and a is a parameter that determines the strength of the balancing feedback loop in relation to the reinforcing feedback loop (with unit stength). Compare this form with the one in the lecture. What is the difference?

Step 1 | Time evolution

Implement the `update_function` for this new model in Python.

```python
# ...
```

Showcae the bistability of this model by plotting the time evoltion of the system state for different initial conditions.

```
# ...
```

Visualize the same inital conditions in a cobweb plot.

```
# ...
```

Step 2 | Bifurcation analysis

Conduct a bifurcation analysis accodring to the one in the lecture, including the calculation and plotting of the equilibirum points and their stability.

```
# ...
```

…

```
# ...
```

Step 3 | Potential function

Derive and plot the potential function for this model

```
# ...
```

…

```
# ...
```

Step 4 | Conclusion

Briefly summarize what your findings mean for the robustness of the tipping elements model.

…

The subcritical pitchfork bifurcation

Bifurcation theory orders different kinds of bifurcations. A so-called **subcritical pitchfork bifurcation** is defined as one where an unstable equilibrium point split into three, two unstable and one stable. Its difference equation (in normal form) is given by

$$\Delta x = x^3 - cx,$$

where c is a parameter that controls the system's stability and x is the system state.

Your task is to implement this model and conduct a bifurcation analysis. We will also use this model in the next lecture.

Step 1 | Stability analysis

Find the critical thresholds of the parameter c at which bifurcations occur and study the stability of each equilibrium point in dependence of the parameter value.

```
# ...
```

...

```
# ...
```

Step 2 | Bifurcation diagramm

Draw an analytical bifurcation diagram of this model for $-1 < c < 2$, showing the equilibrium points and their stability.

```
# ...
```

Now, you should be able to observe why this bifurcation is called a pitchfork bifurcation.

Step 3 | Potential function

Derive and plot the potential function for this model. Also include the equilibirum points and their stability.

```
# ...
```

Another kind of bifurcation in the logistic map

In this exercise we will investigate another kind of bifurcation in the logistic map. Let's revisit the logistic map in the form,

$$x_{t+1} = cx_t(1 - x_t),$$

where $c > 0$.

Step 1 | Stability analysis

Find the critical thresholds of the parameter c at which bifurcations occur and study the stability of each equilibrium point in dependence of the parameter value.

```
# ...
```

...

```
# ...
```

Step 2 | Simulations

Simulate the model with several selected values of to confirm the results of analysis.

```
# ...
```

…

```
# ...
```

Step 3 | Bifurcation diagramm

Draw (simulate) a bifurcation diagram of this model for $0 < c < 4$.

```
# ...
```

Step 4 | Sensitivity to initial conditions

Draw two trajectories from almost the same but a different initial condition for a paramter value $3.7 < c < 4.0$. Draw the two trajectories ontop of each other with different color and linestyle.

```
# ...
```

You should see that the trajectories diverge after a while. This is called **sensitive dependence on initial conditions**, a key property of so-called chaotic systems.

Ex | Resilience

```
import numpy as np
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)
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;
```

The resilience phenomena we discussed are very generic. They don't depend on the exact formulation of the equation. These exercises will show that this claim is not entirely wrong.

Discontinous systems

In this note, we used the classical dynamcial systems from bifurcation theory which are all continuous, meaning that we can write their functional form as one continuous mathematical equation.

The pictorial resilience models often portray a simple cup, such as this,

```
def G(x, a): return np.where(np.abs(x)<a, a/2*x**2, None)

def plot_potential(a=1.0):
    xs=np.linspace(-2,2,301)
    plt.plot(xs, G(xs, a), color='blue')
    plt.ylim(-0.1, 1.1); plt.xlim(-2, 2)

interact(plot_potential, a=(0, 2., 0.01),);
```

```
interactive(children=(FloatSlider(value=1.0, description='a', max=2.0, step=0.01), Output()),
```

Let's interpret this *cup* function as a *quasi-potential* for the following difference equation,

$$
x_{t+1} = \begin{cases} x_t - \frac{dG}{dx}(x_t) + bn_t = x_t - ax_t + bn_t & \text{if } -a \le x \le a, \\ -10 & \text{if } x < -a \\ +10 & \text{if } x > a. \end{cases}
$$

```python
def F(x, a, b):
    if x<-a:
        return -10.0
    elif x>a:
        return 10.0
    else:
        return x - a*x + b*np.random.randn()
```

Showcase robustness resilience with this system

```python
# ...
```

Showcase adaptation resilience with this system

```python
# ...
```

Showcase the critical slowing with this system

```python
# ...
```

Heavy tailed shocks

So far, we assumed the unpredictable and external shocks are distributed according to a normal distribution with mean zero.

Real-world shocks may not exhibit this property. They often come with so-called **heavy tails**, meaning that **large** shocks are more probable compared to a normal distribution.

The Student's t-distribution (or simply the t-distribution) is a continuous probability distribution that generalizes the standard normal distribution. Like the latter, it is symmetric around zero and bell-shaped. The t-distribution has one more parameter than the normal distribution, called the *degrees of freedom*, df.

- When df$\to \infty$, the t-distribution becomes the normal distribution.
- When df$= 1$, the t-distribution becomes the so-called Cauchy distribution.

```python
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import t, norm, cauchy

x = np.linspace(-5, 5, 100)
degrees_of_freedom = [1, 2, 5, 100]  # Varying degrees of freedom

# Plotting T-distribution curves for different degrees of freedom
for df in degrees_of_freedom:
```

```
    y = t.pdf(x, df)  # Using default location and scale parameters (0 and 1)
    plt.plot(x, y, label=f"Degrees of Freedom = {df}")

z = norm.pdf(x)
plt.plot(x, z, 'k--', label='Normal')
z = cauchy.pdf(x)
plt.plot(x, z, 'k:', label='Cauchy')

plt.xlabel('x'); plt.ylabel('PDF'); plt.legend();
plt.title('T-Distribution with Varying Degrees of Freedom');
```
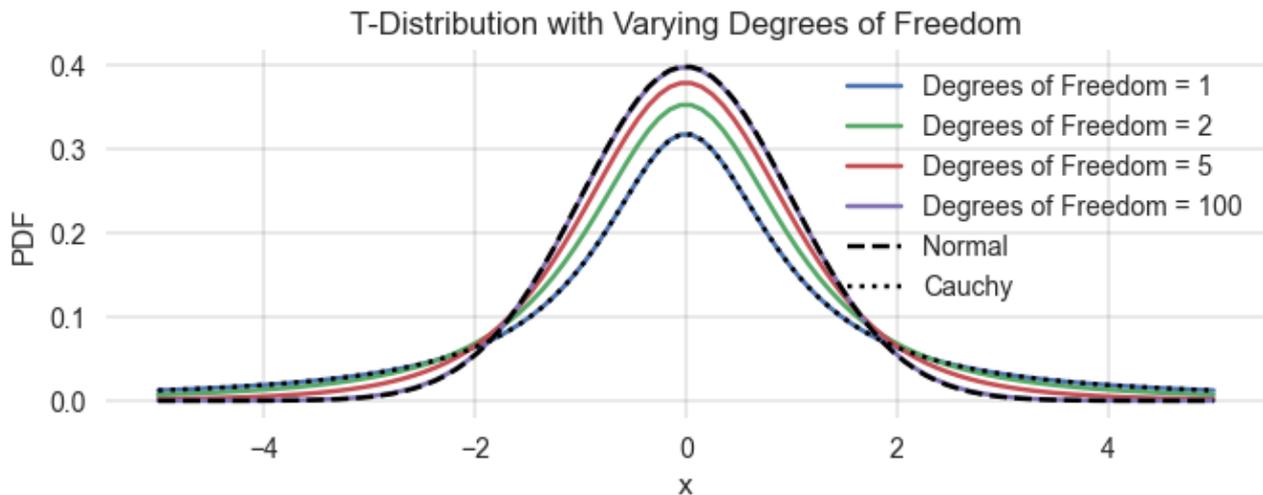

Investigate the impact of heavy-tailed shocks on resilience.

Tip: Define a difference equation with t-distributed shocks and a degree-of-freedom parameter to control the shocks' heavy-tailedness.

```
# ...
```

Autocorrelation with heavy-tailed shocks

Investigate how shocks' heavy-tailedness impacts the lag-1 temporal autocorrelation early-warning indicator.

```
# ..
```

Ex | State transitions

In this exercise, you will apply the compentencies shown in the lecture to a Markov chain with 3 states as shown below.
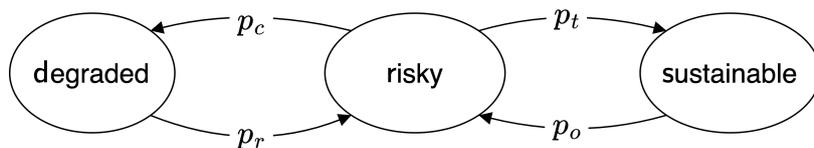
Figure 14: 02.04-ExcerciseExample.dio.png

This model extents the example shown in the lecture between a prosperous and a degraded environmental state by adding a third state. The previous prosperous state is now split into two states: a risky state with a collapse probability p_c to move to the degarded state from which the system transitions back to the risky state with recovery probability p_r, as before.

Now the risky state has also a *transformation probability* p_t to transform the system into a sustainable state. Think of a combination of technological innovation and policy changes that allow the system to transition to a sustainable state. However, due to rebound effects, there is also a *overusage probability* p_o that the system transitions back to the risky state.

```python
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
```

Step 1 | Transition matrix

Set up the transition matrix using the variable with the following default values,

```python
pc = 0.005 # collapse probability
pr = 0.001 # recovery probability
pt = 0.05 # transformation probability
po = 0.01 # overusage probability (relapse to prosperous state)
```

Tip: Self-transitions are not shown in the diagram, but they are possible.

```
# T = ...
```

Make sure and show below that the rows of the transition matrix sum up to 1, as required for a probabilistic transition matrix of a Markov chain.

```
# ...
```

Step 2 | Simulation

Compare the time-evolution of an ensemble stochastic runs with the time-evolution of the state distribution.

Step 2.1 | Stochastic run

First, simulate a stochastic run of the system for 100 time steps with the risky state as the initial state.

```
# ...
```

Visulaize the results in a plot.

```
# ...
```

Step 2.2 | Ensemble

Second, create an ensemble of 300 stochastic runs of the system for 100 time steps with the risky state as the initial state.

```
# ...
```

Make sure that your variable, into which you stored the runs of the ensemble, is a two-dimensional `numpy` array with shape `(300, 100)`.

```
# ..
```

Step 2.3 | Visualize the ensemble

Visualize each of the 300 trajectories in a time series plot by looping through the trajectories of the ensemble and plotting them in the same plot with a low alpha value (`plt.plot(..., alpha=0.05)`) to make the individual trajectories visible.

```
# ...
```

You should realize, that we cannot visulaize the average of the ensemble in a single line plot, as we did in the lecture, because the state space has three states.

Takeing the average of the numerical values representing the states does not make sense in this case. For example, if the risky state is represented by the number 1, the sustainable state by the number 0, and the degraded state by the number 2, the average of the ensemble could not distinguish between all probability in the risky state and 50% in the sustainable and degraded state.

But we still can visualize the evolution of the distribution of the ensemble at each time step.

To do so, we need to transform our ensemble of 300 stochastic runs into a distribution of the ensemble at each time step. We can do so by counting the number of state visits in the ensemble at each time step. The `numpy.histogram` function can help us with this task. Suppose that the ensemble is stored in the variable `ensemble` with shape `(300, 100)`. Then, the following code snippet will count the number of visits of each state in the ensemble at the first time step:

```
stochastic_evolution = np.array([np.histogram(samp, bins=3, range=(0,2))[0] for samp
↪   in ensemble.T])
```

The loop through `ensemble.T` iterates through each time step. The `np.histogram` function counts the number of visits of each state in the ensemble at each time step. The `bins=3` argument specifies that we have three states, and the `range=(0,2)` argument specifies that the states' numerical representations are in the range from 0 to 2. The `[0]` at the end of the list comprehension extracts the counts of the states from the histogram function.

Apply this code snippet to the ensemble and make sure that the variable `stochastic_evolution` has the shape `(100, 3)`.

```
# ...
```

Finally, visualize the state distribution over time using the `plt.imshow` function. The x-axis should represent the time steps, and the y-axis should represent the states. The color intensity should represent the number of visits of each state at each time step. A code like this should do the job:

```
plt.imshow(stochastic_evolution.T, aspect='auto', interpolation='None')
```

```
# ...
```

Step 2.4 | State distribution evolution

To compare this result to the time evolution of the state distribution, you first need to compute the latter.

```
# ..
```

Check that the sum of your state distribution evolution at each time step is equal to 1.

```
# ..
```

Finally, visualize the state distribution evolution over time using the `plt.imshow` function as above.

```
# ...
```

You should observe that both plot look similar, but the state distribution evolution is smoother than the ensemble plot. This is because the ensemble plot results from the individual trajectories, while the state distribution evolution plot shows the average of the ensemble at each time step.

Step 3 | Stationary distribution

Compute the stationary distribution of the system by computing the eigenvector of the transition matrix corresponding to the eigenvalue 1, first numerically and then analytically.

Step 3.1 | Numerical solution

Use the `numpy.linal.eig` method to compute the stationary distribution of the system. Make sure to normalize the correct eigenvector to sum up to 1, as it represents a probability distribution.

```
# ...
```

How does the result compare to the time-evolution above?

Does the system reach the stationary distribution at the end?

If not, what does that say about the time evolution simulations above?

...

Step 3.1 | Symbolic solution

Construct the transition matrix symbolically using the `sympy` package and compute the stationary distribution of the system analytically.

```
# ...
```

Interpret your result

...

Ex | Sequential Decisions

The Markov Decision Process below illustrates the tension between opting for a cautious policy, a short-term risky policy, and a sustainable transformation policy with potential overuse and rebound effects.

The environments consists of three states, $\mathcal{S} = \{\mathsf{s}, \mathsf{r}, \mathsf{d}\}$, a **s**ustainable, a **r**isky, and a **d**egraded state.

In each state, the agent has three actions to choose from, $\mathcal{A} = \{\mathsf{l}, \mathsf{t}, \mathsf{h}\}$, a **l**ow-intensity action toward the environment, a **t**ransformation action, and a **h**igh-instensity action toward the environment.

Immediate rewards r and transition probabilities p are written on the transition arrows as $p|r$.
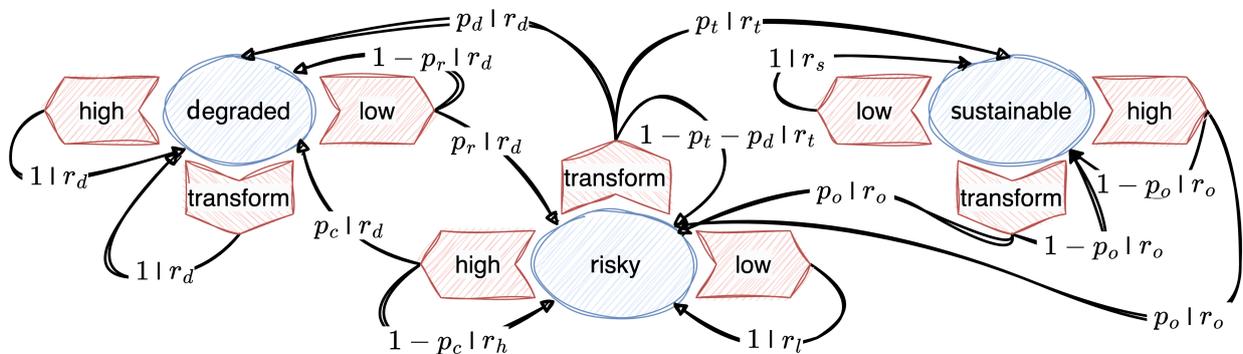
Figure 15: 03.01-ExcerciseExample.dio.png

The **low-intensity action at the risky state** guarantees to stay in the risky environment but requires a reduction of immediate welfare (low-intensity reward $r_l <$ high-intensity reward r_h).

High immediate welfare (r_h) via the **high-intensity action at the risky state** comes at the cost of over-using Nature, which may tip into a degraded state (with collapse probability p_c), where only a low-intensity course of action may recover Nature eventually (with recovery probability p_r).

Whenever the **degraded state** is involved, the agent only receives the lowest degradation reward r_d.

An alternative to the cautious low-intensity action at the risky state is the **transformation action**, where eventually high levels of sustainable wellbeing $r_s > r_h$ can be achieved (after the transition to the sustainable state with transition probability p_t).

However, the transition to the sustainable state requires investment into the transformation technology and thus lower immediate wellbeing during the transition (transition reward $r_t < r_l$).

It also may not guarantee to stay within the risky state and may cause Nature to degrade (with probability p_d).

Rebound effects. At the sustainable state, however, there is the risk of overusing the transformation technology (if another action than the low-intensity is taken) since this yields the highest overuse reward r_o, which risks (with probability p_o) the relapse back into the risky environmental state.

```python
import numpy as np
import matplotlib.pyplot as plt
```

Step 1 | Transition and rewards tensors

Transform the description of the MDP into two `Python` functions that can be called with the model parameters and return a transition and reward tensor, respectively. Make sure to write these functions to be used both with `numpy` and `sympy`.

```
# ...
```

Test that both functions work and check that the transition tensors are proper probability distributions for three exemplary parameter combinations.

```
# ...
```

Step 2 | State values

Write a `Python` function to numerically compute the MDP's state values, given a `policy_Xsa`, a `transitions_Tsas` tensor, a `rewards_Rsas` tensor, and a discount factor `dcf`.

```
# ...
```

Test that your function works with some arbitrary values for the `policy_Xsa`, `transitions_Tsas` tensor, `rewards_Rsas` tensors, and discount factor `dcf`.

```
# ...
```

Step 3 | Policies

Formulate four different policies and represent them in `Python`:

The **cautious policy** always chooses the low-intensity action.

```
# ...
```

The **risky policy** chooses the high-intensity action in the risky and sustainable state and the low-intensity action in the degraded state.

```
# ...
```

The **transformation policy** chooses the transformation action in the risky state and the low-intensity action in the sustainable and degraded state.

```
# ...
```

The **overuse policy** chooses the transformation action in the risky and sustainable state and the low-intensity action in the degraded state.

```
# ...
```

Step 4 | Optimal policy

What are the state values of the risky state for all the four policies at the parameter combination $p_c = 0.2$, $p_r = 0.01$, $p_t = 0.04$, $p_d = 0.005$, $p_o = 0.02$, $\gamma = 0.98$, $r_o = 1.2$, $r_s = 1.0$, $r_h = 0.9$, $r_l = 0.7$, $r_t = 0.65$, $r_d = 0.0$?

```
pc = 0.2; pr = 0.01; pt = 0.04; pd = 0.005; po = 0.02;
ro = 1.2; rs = 1.0; rh = 0.9; rl = 0.7; rt = 0.65; rd = 0.0
```

```
# ...
```

Which is, therefore, the optimal policy for that parameter combination?

```
# ...
```

Step 5 | Optimal policies with uncertainty

Given the other above parameters, how does the optimal policy change with a varying discount factor γ? Create a plot that shows the state values of the risky state for the four policies at $\gamma \in [0.001, 0.9999]$ and interpret your result.

```
# ...
```

Ex | Strategic Interactions

This exercise will revisit the tragedy dilemma, agreements, and threshold public goods but use slightly different modeling assumptions. Doing so will give us a more robust understanding of the strategic interactions in multi-agent action situations.

```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
```

Step 1 | Tragedy Dilemma

Next to the model presented in the lecture, another common parametrization of the tragedy dilemma is the following: Actors can either cooperate or defect. Each cooperator contributes $c > 0$ to the public good at an individual cost of c. The sum of all contributions is multiplied by a synergy factor r and then equally distributed among all actors. The payoff functions are given by:

$$R_c = \frac{rc(N_c + 1)}{N} - c, \tag{.22}$$

$$R_d = \frac{rcN_c}{N}, \tag{.23}$$

with N_c being the number of other actors cooperating and N being the total number of actors.

Step 1.1 | Visualization

Plot the payoff functions for cooperators and defectors as a function of N_c for different values of r. Compare the results to the tragedy dilemma model presented in the lecture.

```
# ...
```

Step 1.2 | Conditions

Give the conditions the parameters must hold for this model to be a tragedy dilemma.

```
# ...
```

Step 2 | Agreements

Apply the reasoning from the lecture to compute how the critical participation levels depend on the parameters of the model r, c, and N.

```
# ...
```

Step 3 | Threshold Public Goods

Let us now consider a variant of threshold public good, where the catastrophic impact m occurs probabilistically, and each polluting actor increases the probability of collapse. We assume that if all actors pollute, there is a probability q_c of collapse; if all actors cooperate, there is zero probability of collapse. The collapse probability increases linearly with the number of polluting actors N_p, i.e.,

$$p_c = q_c \frac{N_p}{N}.$$

Furthermore, we assume that if the collapse occurs, the actors won't receive the payoffs from the public. The payoff functions are given by:

$$R_a(N_a; r, c, q_c, m, N) = (1 - q_c N_p/N) \cdot (rc(N_a + 1)/N - c) + q_c N_p/N \cdot m, \tag{.24}$$

$$R_p(N_a; r, c, q_c, m, N) = (1 - q_c(N_p + 1)/N) \cdot (rcN_a)/N) + q_c(N_p + 1)/N \cdot m, \tag{.25}$$

where N_a is the number of other actors abating and N_p is the number of other actors polluting. Thus, it must hold that the total number of actors $N = N_a + N_p + 1$. Furthermore, r is the synergy factor, c is the cost of cooperation, q_c is the probability of collapse if all actors pollute, and m is the catastrophic impact.

Step 3.1 | Visualization

Plot the payoff functions for abating and polluting actors as a function of N_a for $f = 4$, $c = 5$, $m = -5$ $qc = 0.4$, and $N = 5$. Compare the results to the threshold public goods model presented in the lecture.

```
# ...
```

Step 3.1 | Conditions

Calculate the three critical conditions for this game of

1) *Dilemma*, i.e., the actors are indifferent between all abating and all polluting,
2) *Greed*, i.e., the actors are indifferent between abating and polluting, give all others abate, and
3) *Fear*, i.e., the actors are indifferent between abating and polluting, given all others pollute.

Solve the conditions for the collapse impact m.

You may do this by hand or by using the sympy library. I recommend the latter.

```
# ...
```

Visualize the critical conditions for the collapse impact m as a function of q_c for $r = 1.2$, $c = 5$, and $N = 2$. Interpret the results briefly.

```
# ...
```

Visualize the critical conditions for the collapse impact m as a function of $N \in [2, 3, \ldots, 15]$ for $r = 3$, $c = 5$, and $qc = 0.5$. Interpret the results briefly.

```
# ...
```

Ex | Dynamic Interactions

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
```

In the chapter on dynamic interactions, we saw how being embedded in the same environment could resolve the tragedy into a comedy of the commons. In this exercise, you will investigate the **ecological micro-foundations of social dilemma traps**. That is when there is no game to begin with in the short term; what kind of incentive regimes can emerge when the agents are embedded in the same environment? The model is very similar to the one presented in the chapter, and a detailed analysis can be found in (Barfuss, Donges, et al., 2024).
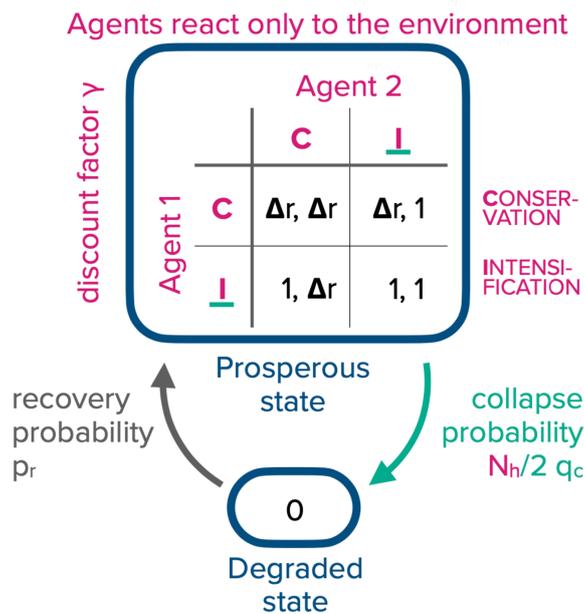
Barfuss et al. (2024) Ecologically-mediated collective action in commons with tipping elements

Figure 16: 03.03-EcologicalMicroFoundation.dio.png

Model description

Two representative decision-makers repeatedly interact (in discrete time) within an environment of two states, a prosperous and a degraded one. In the prosperous state, each agent can choose between a conservation and an intensification action, which affects the environment. For example, intensification corresponds to emitting a large, business-as-usual amount of carbon into the atmosphere, a high rate of rainforest deforestation, or a high level of freshwater extraction. In contrast, conservation corresponds to a significantly reduced amount of greenhouse gas emissions, rate of deforestation, or level of fresh-water extraction within the recovery capacity of the environment.

Provisioning goods and immediate benefits. In a prosperous environment, the biosphere's maintenance and regulating services function sufficiently well, and the environment can deliver provisioning goods to the agents. We assume these goods are entirely private, i.e., their benefits do not depend on the other agent's action. Each agent opting for conservation receives a benefit b_C. Each agent opting for intensification receives a higher benefit $b_I > b_C$. In the degraded state, however, the biosphere's maintenance and regulating services are not functioning. Thus, whenever the environment collapses into or is in the degraded state, all agents receive only a benefit $b_D < b_C$ independent of their action.

To reduce the number of free model parameters, we summarize these three benefits into a single relative benefit ratio, $\Delta r = f(b_C) \in [0, 1]$ by transforming all benefit parameters according to $f(x) = (x - b_D)/(b_I - b_D)$. Δr describes how much less immediate benefit conservation delivers (in relation to the collapse impact b_D) compared to intensification. Thus, a larger Δr denotes either a larger conservation benefit, a smaller, more severe degradation benefit, or a smaller intensification benefit.

Ecological tipping and transitions. However, each agent employing intensification also increases the probability of triggering the tipping element and collapsing the environment into the degraded state. We parameterize the overall collapse probability p_c by the collapse leverage $q_c \in [0, 1)$ each intensification actor exerts on the environment. The total collapse probability $p_c = 0$, if no actor chooses intensification, $p_c = q_c/2$ if one actor chooses intensification, and $p_c = q_c$ if both do. In the degraded state, the agents cannot influence the environment and have to wait on average for $1/p_r$ rounds, parameterized by the recovery probability p_r, until they enter the prosperous state again.

Decision model. We assume that each agent's conservation or intensification strategy is conditional solely on the current environmental state. They do not take longer histories, their own past choices, nor the choices of the other agent into account - either because they do not have the cognitive resources for more complex strategies or because they cannot observe or make sense of the other agent's actions. But we assume that agents can plan their course of action into the future and that they care for future rewards but exponentially discount them with their discount factor $\gamma \in [0, 1)$.

Note that all direct social interaction in the model is deliberately stripped away. The agents' choices do not influence the immediate benefit of the other agent. Agents are self-interested and do not consider the other agent's actions for their strategy. Social interaction is only mediated indirectly through the environment. This is not to say that such direct social interaction does not exist.

Our model aims to isolate the cooperation-promoting effects of the actors' shared ecological embeddedness. There is already good evidence for the beneficial effect of direct social interaction on collective action (Nowak, 2006; Ostrom, 1990). We aim to assess the prospects for collective cooperation when such mechanisms cannot work, either because of the anonymity or the scale of the problem.

In summary, our theoretical model is determined by only four parameters, two ecological and two social ones: the collapse leverage q_c, the recover probability p_r, the discount factor γ, and the relative benefit ratio Δr. All of them are in the range between 0 and 1.

Task

Visualize the critical curves where the social dilemma incentive regimes change - solving for the discount factor γ as a function of the collapse leverage q_c (assuming agents with identical parameters). The other two parameters are $\Delta r = 0.5$ and $p_r = 0.1$. Briefly interpret your plot.

```
# ...
```

Ex | Behavioral Agency

```python
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
```

Schelling's segregation model demonstrates how individual preferences can lead to racial segregation, even when people are comfortable in mixed neighborhoods. In this exercise, you will explore how the model's outcome depends on some of its parameters. You will investigate what it takes to achieve a well-mixed society.

Task 1 | Implement Schelling's model

Define a Python function `run_model(agents)` that simulates Schelling's model for a given iterable of agents (`agents`), the number of agents regarded as neighbors `num_neighbors` and the number of neighbors required to be of the same type `require_same_type` such that an agent is happy. The function should return an iterable of agents after the model has converged. The model should run until no agent wants to move anymore. Print out the current cycle number while the model is running.

```python
# ...
```

Task 2 | Run the model

Test your implementation by running the model with 500 agents, 10 neighbors, and 5 neighbors of the same type required. Initialize the agents with 250 agents of type 0 and 250 agents of type 1. Plot the distribution of agents at the beginning, and the end of the model run in a matplotlib figure with two axes next to each other.

```python
# ...
```

Task 3 | Performance metric

Having a visual understanding of the model's behavior is essential. However, it is also useful to have a quantitative measure of the model's performance. Define a Python function `homogeneity(agents, num_neighbors)` that returns the average homogeneity of the agents. The homogeneity of an agent is the fraction of its neighbors that are of the same type. The average homogeneity is the average of the homogeneity of all agents.

```
# ...
```

Test your implementation by calculating the homogeneity of the initial and final agent distributions from the simulation above.

```
# ...
```

Convince yourself that the homogeneity is a useful measure of the model's performance by repeating the simulation with the same parameters as above, except the number of neighbors required to be happy is set to 9. Visualize and calculate the homogeneity of the final agent distribution.

```
# ...
```

Task 4 | Sensitivity analysis

Perform a simulation to investigate how the final homogeneity depends on the number of neighbors considered, `num_neighbors`, and the number of neighbors of the same type required to be happy, `require_same_type`.

First, we keep the number of neighbors required to be happy fixed to be exactly half the number of neighbors considered. How sensitive is the final homogeneity to the number of neighbors considered? Run the model for `num_neighbors` in the range from 2 to 20 in increments of two. As the simulation is stochastic, run each simulation precisely five times. Plot the final homogeneities (plus their averages) versus the number of neighbors considered. Briefly interpret your result.

```
# ...
```

Finally, run the model for `num_neighbors` in the range from 2 to 14, and for each `num_enighbors`, the `require_same_type` from 1 to `num_neighbors`-1. Plot the final homogeneity as a heatmap.

Tip: Store the final homogeneities in a two-dimensional NumPy array

```
final_homogeneities=np.NaN * np.ones((15, 15)),
```

where the first dimension corresponds to the number of neighbors considered and the second dimension to the number of neighbors of the same type required to be happy. The `np.NaN` values will be useful for plotting the heatmap as the plotting functions ignore the 'NaN' values. You can you use the `imshow` function from matplotlib to plot the heatmap,

```
plt.imshow(final_homogeneities, cmap='RdBu', vmin=0, vmax=1.0, origin='lower').
```

Include a color bar and a line indicating where the number of neighbors considered is equal to half the number of neighbors of the same type required to be happy.

Interpret your results regarding the maximum number of neighbors required to be happy to achieve a well-mixed society. What happens to the final homogeneity when the number of neighbors required to be happy exceeds half the number of neighbors considered?

```
# ...
```

Ex | Individual Learning

```python
import numpy as np
import sympy as sp
import pandas as pd
import matplotlib.pyplot as plt
from copy import deepcopy
```

Task 1 | Learning the risky policy

In the lecture, we explored how the agent learns a cautious policy within the risk-reward dilemma. Investigate the learning process for parameter combinations that make the risky policy optimal (`DiscountFactor=0.6`, `CollapseProbability=0.1`, `RecoveryProbability=0.1`, `SafeReward=0.5`, `RiskyReward=1.0`, `DegradedReward=0.0`). What parameters of the learning process, such as learning rate and choice intensity, allow the agent to consistently learn the risky policy?

```python
# ...
```

How does the learning process change if you change the transition probabilities to `CollapseProbability=0.05`, `RecoveryProbability=0.005`?

```python
# ...
```

Task 2 | Ecological public good

Implement the ecological public good from Lecture 03.03 as a reinforcement learning environment. Ensure your `EcologicalPublicGood` class inherits from the base Environment class.

```python
# ...
```

Let two agents learn in it and visualize the learning process.

```python
# ...
```

Briefly discuss your findings.

Ex | Learning Dynamics

```
import numpy as np
import matplotlib.pyplot as plt
```

Task 1 | Social dilemma flows

Visualize the flow plots for all four social dilemma environment we discussed in the course: Tragedy Prinsoner's Dilemma, Divergence Chicken, Coordination Stag Hunt, and Comedy Harmony.

You can use the `pyCRLD` environment `SocialDilemma` by impporting

```
from pyCRLD.Environments.SocialDilemma import SocialDilemma
```

```
# ...
```

Task 2 | Critical transition

We consider the following model: Two agents can either cooperate or defect. A cooperator contributes a benefit b, which all agents receive. However, a cooperator must pay c for the contribution. A defector does not contribute and does not pay a cost. Thus, the payoff matrix is

	Cooperate	Defect
Cooperate	$2b-c$, $2b-c$	$b-c$, b
Defect	b, $b-c$	$0,0$

Let us re-normalize the payoffs, devide all payoffs by b and express in the cost-to-benefit ratio $r = c/b$.

	Cooperate	Defect
Cooperate	$2-r$, $2-r$	$1-r$, 1
Defect	1, $1-r$	$0,0$

Simulate the reinforcement learning dynamics in the game from 25 random initial joint policies for values of r in the range $[0.5, 1.5]$. Record the final joint policy for each initial policy and plot the critical transition from defection to cooperation as a function of r. Also, visualize how long, on average, it takes for the agents to reach the final joint policy. Show a critical slowing down.

```
# ...
```