Project: The Norway Lemming

Introduction

The Norway lemming (*Lemmus lemmus*) is a small and adorable rodent (see Fig. 10.1) that leads a fascinating life in the Arctic tundra. Known for their cuteness, these creatures play a crucial role in the delicate balance of their ecosystem. With their round faces and compact bodies, Norway lemmings navigate the harsh Arctic landscape where predators like Arctic foxes, birds of prey, and weasels pose constant threats. Their lives are closely tied to the weather and snow conditions, as they build intricate tunnel systems beneath the snow cover, providing them with insulated pathways to forage for food and escape predators. These tunnels not only provide shelter but also shield them from the freezing temperatures above. Remarkably, they can continue their activities beneath the snow, such as foraging for vegetation, even during the darkest winter months. Their diet primarily consists of grasses, mosses, and other vegetation found in the Arctic tundra. During the short Arctic summer, when the snow melts, they take advantage of the lush plant growth to store food for the harsh winter months.
Lemmings exhibit cyclic population dynamics, undergoing periodic fluctuations in numbers, and their breeding patterns are often influenced by environmental cues. Despite the challenges they face, Norway lemmings contribute to the biodiversity of the Arctic and exemplify the resilience of life in extreme climates. This project focuses on modeling the lemming population in different ways. This requires the integration of environmental effects and their estimation from data. Modeling fluctuations and periodic changes in animal populations is an active and highly interesting field. One of the earliest and best-known articles dates back to Charles Elton in 1924 and is – surprise, surprise – about the Norway lemming.

Part 1: A Simple Model for the Lemming Population

The goal is to simulate the population size, i.e. the number of lemmings N with respect to time t. The simplest model for the single-species population dynamics that can achieve long-term stability is the logistic model shown in Eq. 10.4.

Figure 10.1: A Norway lemming during summer. [*Lakkahillo, via Wikimedia Commons*].

Figure 10.2: A Norway lemming in snow [*kgleditsch, via Wikimedia Commons*].

$$\frac{\mathrm{d}N}{\mathrm{d}t} = rN(1 - \frac{N}{k}) \tag{10.4}$$

Here, r is the intrinsic rate of population growth and describes the balance between births and deaths. k describes the so-called carrying capacity and ensures that the population cannot grow indefinitely, e.g. due to limited resources such as food and space.
Eq. 10.4 assumes a linear dependence of the resource limitation ($^r/_k \cdot N$), which is quite limiting for our purposes. An alternative, more general way to describe the population dynamics is therefore given by

$$\frac{\mathrm{d}N}{N\mathrm{d}t} = b - d - f(N) \quad , \tag{10.5}$$

where b and d are the birth and death rates, respectively. The integrated function $f(N)$ represents an additional death rate that depends on the current population size N. Normalizing the rate of change $^{\mathrm{d}N}/_{\mathrm{d}t}$ to N essentially redefines all rates on the right side of Eq. 10.5 to birth and death rates **per individual**, i.e. the probability of a lemming being born or dying **per unit time**. [1]
Let's assume the following facts about the Norway lemming and our observed community:

- Female lemmings produce (on average) 2.5 litters per year. Each litter has 7 newborns. Assume that the sex ratio is constant at 50/50 (male/female).
- The lifespan of a lemming out in the wild is 2 years.
- The community lives in a closed space, i.e. lemmings cannot get in or out. The environment allows exactly 75 lemmings to eat every day. If a lemming goes without food for 5 days in a row, it will starve. When it eats, this counter resets.

[1]We will work with days, i.e. all rates have the unit [d^{-1}].

○ 🏁 Write a Python script `LEMMING_YOURNAMES.py` in the main directory `./` that calculates the evolution of the lemming population for the following parameters:

- Initial population size `N_INIT = 500`
- Simulation time in days `SIM_TIME = 3*365`

Perform all calculations for the birth and death rates b, r and f in separate functions called `return_b()`, `return_d()` and `return_f(N)`. They should incorporate the physical information given above. You can define numerical constants like the number of litters per year *inside* the function. Unfortunately, calculating the *daily* probability of starvation f (in $[\mathrm{d}^{-1}]$) is not trivial, but can be approximated with

$$f(t) \approx \left(\frac{N(t)-F}{N(t)}\right)^D \left[\sum_{i=0}^{D-1} \binom{D-1}{i} \left(\frac{F}{N(t)}\right)^i \frac{1}{D+i}\right] \quad , \tag{10.6}$$

where F is the number of lemmings that can eat per day and D is the number of days without food when a lemming starves. Write these functions in a module `helper_functions.py` located in the subfolder `./mod/`.

Eq. 10.6 neglects the resetting of the eat-counter, which we will come back to. $\binom{n}{k}$ are binomial coefficients that can be computed with `binom(n,k)` from the `scipy.special` package. If you manage to find the full analytical formula for the probability **please**, let us know.

○ 🏁 Visualize the evolution of the population over time (in days) and export the plot as `part1.png` into the subfolder `./exp/` with a resolution of 300 DPI. Make the plot 17cm wide and use Arial, font size 11 throughout.

If your population converges to a steady state of about 145 lemmings, you have done well. Otherwise, you probably need to take another look at your birth and death rates or your numerical solution.

Part 2: Throwing Dices

There is another, fundamentally different approach to calculating the lemming population dynamics $N(t)$ with the physical boundary conditions given above. This approach does not involve solving any differential equations.

○ 🏁 Write a Python module `lemming.py` in the mod subfolder `./mod/` that contains the class `lemming()`. Initialize a lemming with the following attributes:

- `age = 0`. The age of this lemming in days.
- `toughness = 5`. The number of days this lemming can survive without eating.
- `last_eaten = 0`. The number of consecutive days this lemming has gone without food.
- *(optional)* `name = '..'`. Ask Chat-GPT for a list of cute lemming names, save them as `.txt`, import this list and choose one at random. This is just for your personal immersion though ☺.

○ 🏁 Add the following methods to the class that are similar (but not necessarily identical) to `return_b()`, `return_d()` and `return_f(N)` from part 1. Again, you can define numerical constants inside the function and the functions should follow the physical constraints from above.

- `check_dead(self)`. Calculate the probability of death for a given day. Generate a random number and return `True` if the lemming dies.
- `check_reproduce(self)`. Calculate the probability of giving birth for a given day. Generate a random number to see if a birth occurs. Return the number of babies that were born.
- `check_food(self, N)`. Calculate the probability of finding food for a single day based on the population size N. Generate a random number and return `True` if the lemming finds food on that particular day.

○ 🏁 Add the method `live_a_day(self,N)` to the class, which takes the current population size N as input and returns

- `alive` (`bool`). True, if the lemming survives the day
- `reproduce` (`int`). The number of babies born by this lemming.

Use the previously defined methods. Keep track of the lemming's age and the number of days it has gone without food. Make sure the lemming starves based on it's `toughness` attribute.

○ 🏁 Extend the `LEMMING_YOURNAMES.py` script in the main directory `./` to simulate the lemming population based on the `lemming` class: Use the same parameters `N_INIT` and `SIM_TIME` as above. Initialize a list `L` containing all individual lemming instances. For each day, let each lemming in the list `live_a_day()` and adjust the list `L` accordingly. For each day, store the lemming population size N in a numpy array.

○ 🏁 Visualize the evolution of the population over time (in days) and plot it together with the results from part 1. Export the plot as `part2.png` into the subfolder `./exp/` with a resolution of 300 DPI. Make the plot 17cm wide and use Arial, font size 11 throughout.

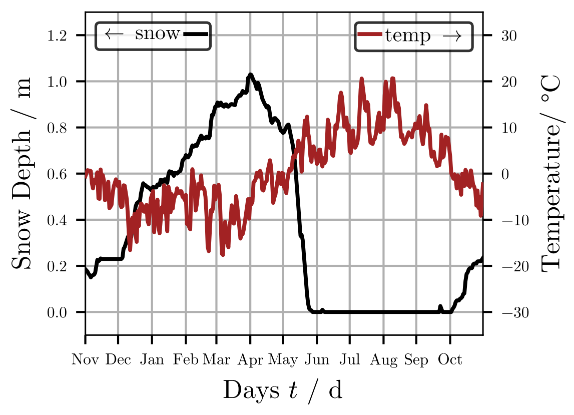

Figure 10.3: Seasonal changes in snow depth and temperature in Varangerhalvøya National Park between 01.11.2022 and 31.10.2023. [open meteo]

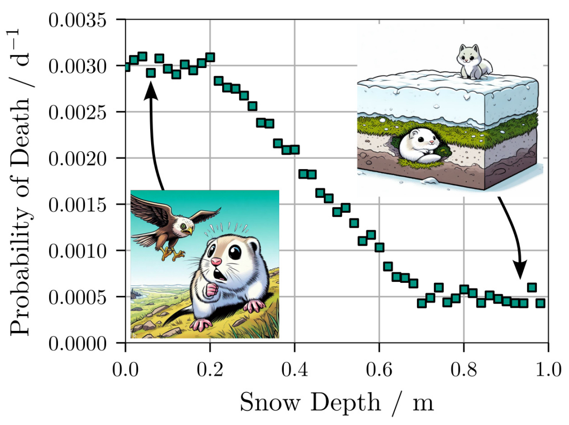

Figure 10.4: “Measured” death rates of Norway lemmings as a function of snow depth.

Compare your results from part 1 and part 2. They should be quite similar with respect to the final population size N. Why do you see fluctuations and why are the results from part 2 a bit higher than part 1 (revisit the assumptions of Eq. 10.6)? Leave a short comment in your code.

Congratulations, you just ran your first Monte Carlo simulation! As you can see, some probabilities are hard to define analytically, but a simple Monte Carlo simulation can solve this problem for you!

Part 3: Modeling Seasonal Dependencies

In reality, none of the kinetic rates in Eq. 10.5 are constant, but depend strongly on seasonal variations. This part of the project is concerned with modeling the dependence of both d and f on the thickness of the snow cover (snow depth) and temperature.

- The file `./data/weather_data.npy` contains a dictionary with daily weather information from Varangerhalvøya National Park between 01.11.2022 and 31.10.2023 [open meteo]. This data is visualized in Fig. 10.3. Write a function `get_weather(start_date, t)` that returns the snow depth and temperature for a give day. The inputs are an initial date as `str` in the format `'MM-DD'` (e.g. `'06-28'` for June 28) and the number of days `t` after this point (as `int`). [2] Include this function in the `helper_functions.py` module. Assume that the weather does not change from year to year. Also, if February 29 occurs in the function, pretend it is February 28 (there is no weather data for `'02-29'`).

[2]If you want to get weather information for the 21st of May, you could use `get_weather('05-21',0)`, `get_weather('05-20',1)`, `get_weather('05-19',2)` and so on.

Defining the function in this way makes it much more elegant to use with `solve_ivp()` later, since time `t` (in days) will always be a required argument for the right hand side function. You might want to take a look at the `datetime` package!

○ 🏁 Your friend Ole Olsen lives and works in the Arctic as a lemming researcher and has been studying how lemming death rates d depend on the snow depth. He sent you the file `./data/d_sd.npy`, which contains a dictionary with corresponding d and snow depth pairs. He also visualized his data in Fig. 10.4. See how the lemmings depend on the snow cover to avoid being eaten by foxes and birds? He also looked at the dependence on temperature, but found no significant effect, except the obvious correlation with snow height. [3]

Look at the data and think of a good model to estimate the relationship. Then, write a function `fit_d_sd()` that fits that model to the data and saves the found parameters in a `.npy` file. Additionally, plot both the raw data and your correlation and save them as `d_sd_fit.png`.

Write a second function `get_d(snow_depth, data=None)` that takes these parameters as optional `data` argument and returns the death probability based on a given snow depth. If `data` is not provided (i.e. `None`), import the model data from a file. Both functions should be located in your `helper_functions.py` module.

Work smarter, not harder: Your model function does not have to be super elaborate at this point!

○ 🏁 Ole is not finished yet: He also investigated how many lemmings actually eat per day depending on the snow depth and temperature. Impressive analytical skills, you have to appreciate his work! Again, he sent you this data `./data/f_sd_T.npy` and visualized it in Fig. 10.5. As you can see, the dependencies are not quite as straightforward, but you can generally say that an increase in snow cover leads to less food (lemmings are slower) and an increase in temperature provides more food (more vegetation).

Write a function `fit_f_sd_T()` that imports the data, fits a (non-linear) regression model and saves it in a `.npy` file. Additionally, plot both the raw data and your correlation and save them as `f_sd_T_fit.png`. Distinguish the training and test data by marker color. Include some form of quality criterion for your model in the figure title.[4]

Write a second function `get_f(snow_depth, temp, data=None)` that takes your model as an optional `data` argument and returns the number of lemmings that eat per

[3] To be clear: d can be estimated solely from snow height and temperature can be neglected here.

[4] The chosen quality criterion and the part of the data you evaluate it on should help reveal potential overfitting!

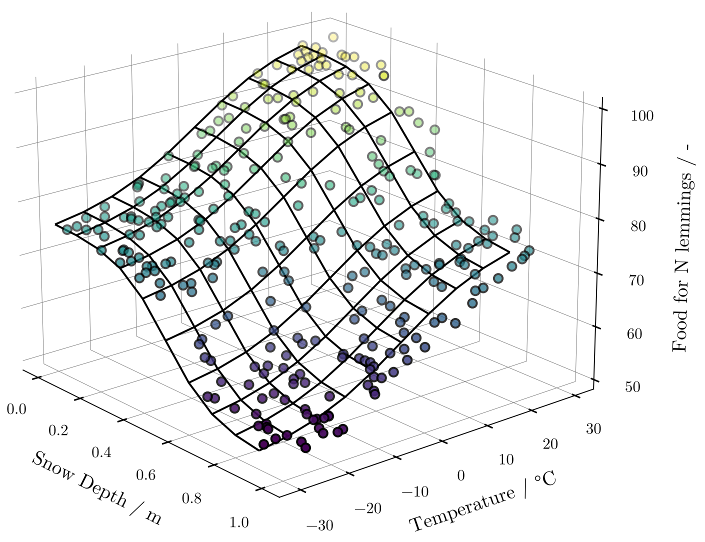

Figure 10.5: Available food depending on snow depth and temperature.

day based on a given snow depth and temperature. If `data` is not provided (i.e. `None`), import the model data from a file. Print a warning if the arguments are outside the calibrated range of values (extrapolation is dangerous!). Both functions should be located in your `helper_functions.py` module.

○ 🏁 Implement your data-driven models for the kinetic rates in both the differential equation from part 1 and the Monte Carlo simulations from part 2. Make sure that all the calculations from parts 1 and 2 still work without modifying the code in the main script above. [5]
Re-calculate both the continuous and the Monte Carlo simulation for the same parameters `N_INIT` and `SIM_TIME`, but include environmental information in the kinetic rates. Start your simulation on 1st January. Visualize the evolution of the population over time (in days). Plot the results of both the solved differential equation and the Monte Carlo model. Export the plot as `part3.png` into the subfolder `./exp/` with a resolution of 300 DPI. Make the plot 17cm wide and use Arial, font size 11 throughout.

✓ If the population of your continuous model fluctuates roughly between 110 and 185 lemmings over the course of a year and the Monte Carlo results are comparable (a little higher), you probably did well. Otherwise, you should take another look at your code.

[5]Work with optional arguments and set the default values accordingly. For example, you can define a variable `CASE` that specifies whether or not environmental parameters should be taken into account.

In this lecture, we have not really covered how to optimize our code for performance. However, re-importing the model parameters for d and f each time can be time consuming, especially for the Monte Carlo simulations. Therefore, import them once in your `LEMMING_YOURNAMES.py` script and pass them via the optional `data` arguments. See how this reduces the computation time!

(Optional) Part 4: Natural Selection

NOTE: The following objectives are not required and are purely optional. However, they might provide you with an additional challenge and some insights into the benefits of Monte Carlo simulations.

- ○ 🏁 Add an optional argument to your `lemming` class that – if set to `True` – makes the initialization of an instance's `toughness` attribute random (within a reaonable range). Make sure to set the default value accordingly, so that all the previous parts still work as intended without changing the code. In addition, think about how you could modify the birth process of a lemming to inherit the properties its parent, while still allowing room for individuality.
- ○ 🏁 Add some opposing effects to the kinetics, where e.g. the `toughness` of a lemming might help it survive days without food but make it slow and more susceptible to being eaten. You could also experiment with other traits that lemmings might have that affect their birth and death rates, such as e.g. `speed`. In addition, it may be necessary to experiment with the start date and simulate a longer evolution time. You can be creative, but if you are very uninspired you could use a model like this:

$$d^* = d \cdot \texttt{toughness} \tag{10.7}$$

- ○ 🏁 Repeat the calculations from part 3, but now also track the average `toughness` value of the population and add it to the plot on a second y-axis. Compare and discuss your results.

Congratulations, in essence, you have just programmed an optimization algorithm to find the optimal properties of a lemming to survive in the wild. Such genetic algorithms are widely used, especially for complex optimization problems. However, they require a lot of cost function evaluations (i.e. lemming life simulations).

A major advantage of Monte Carlo simulations is the ability to track individuals in a population rather than just some averages. This opens up many doors for optimization and detailed investigations and can also be applied to engineering problems: For example, population balance equations (PBE) are able to track the evolution of particle size distributions subject to agglomeration and breakage processes. Solving these PBE with Monte Carlo methods allows for particle-discrete information and an easy integration of multiple particle properties.