Lotka–Volterra Prey–Predator Simulation

Tommaso Federici
0001173262

February 7, 2026

1 Introduction

This project is a C++ application developed for academic purposes.
It implements a **Lotka–Volterra prey–predator simulation**, allowing the user to define initial conditions and model parameters and to study the system's evolution through a **discrete-time model**. It includes unit tests (with **Doctest**) and graphical rendering (with the **SFML** library).

2 Implementation

Hereinafter, the main project decisions are described and motivated.

2.1 Structure

The project is organized in a hierarchical, tree-like directory structure.
The root directory, named `project/`, contains configuration and utility files, while the source code is divided into 4 dedicated subdirectories:

- **include/**: header files (class and function declarations);
- **io/**: input/output implementation files (handling user interaction and data writing);
- **src/**: main source files, including numerical simulation and rendering logic;
- **test/**: unit test files (Doctest-based);
- `CMakeLists.txt`: build configuration file (for CMake and Ninja);
- `doctest.h`: testing framework header;
- `font.ttf`: font resource file (used by renderer).

2.2 Design choices

The project is modular, separating numerical simulation, rendering and input/output handling.
All Lotka–Volterra code resides in the `lotka_volterra` namespace, I/O in `io`.
Parameters are validated at run time to ensure physical and numerical consistency and errors are reported via `throw` statements to avoid invalid states.

This design choice allows configuration errors to be detected early and reported with informative messages, preventing the simulation and the renderer from operating on inconsistent or non-physical states. In particular, run-time validation is preferred over compile-time checks to support both interactive execution and programmatic configuration.

The `Simulation` class stores the state at each time step, model parameters and the relative variables x_{rel}, y_{rel}, together with stability information. Private methods handle parameter checking, single-step evolution and energy computation, while the public interface allows access to data and evolution over steps or time intervals. The time evolution is based on a discretized version of the Lotka–Volterra equations, where populations are updated through small time steps Δt. This reflects the discrete nature of the simulation.

The `Renderer` class uses SFML to display the trajectory. Two views (`ui_view` for interface, `world_view` for simulation) allow independent scaling. Private methods manage scaling, tick computation and incremental drawing; public methods provide drawing up to a step or the full trajectory. Points are colored by relative energy variation and equilibrium points are highlighted.

Input/output functions allow both interactive terminal input and programmatic configuration, and simulation data can be exported to CSV.

3 Input–Output

Interactive execution requests:

- time step $dt \in [0.0001, 0.01]$;
- strictly positive model parameters A, B, C, D;
- positive initial populations x_0, y_0;
- total simulation time T (multiple of dt);
- window $size \in [800, 1000]$.

All inputs are validated; invalid entries raise exceptions.
An example of valid interactive input is summarized in Table 1.

Parameter	Value
dt	0.001
A	10
B	6
C	4
D	12
x_0	4
y_0	3
$size$	1000
T	10

Table 1: example of valid interactive input values for the simulation and the rendering.

Outputs include a graphical window of the x–y trajectory (animated or full) and/or a CSV file containing time, populations and energy.

4 Compilation

To compile the project, make sure you have a C++20–compliant compiler. If your compiler does not support C++20, please update it or use an alternative one that does.
The following dependencies are required:

- **SFML** (version 2.6 or later)
- **CMake** (version 3.28 or later)
- **Ninja**

To install them, please follow the commands below.
On Linux:

```
$ sudo apt install libsfml-dev
$ sudo apt install cmake
$ sudo apt install ninja-build
```

On macOS:

```
% brew install sfml
% brew install cmake
% brew install ninja
```

Next, unzip `project.zip` and navigate to the project directory:

```
$ unzip project.zip
$ cd project
```

Then, create and configure the `build/` directory using the `CMakeLists.txt` file located in the current directory:

```
$ cmake -S . -B build -G"Ninja Multi-Config"
```

Finally, compile the program and run the tests in either **Debug** or **Release** mode:

```
$ cmake --build build --config Debug
$ cmake --build build --config Debug --target test
$ cmake --build build --config Release
$ cmake --build build --config Release --target test
```

The executable can then be run in either configuration:

```
$ ./build/Debug/project
$ ./build/Release/project
```

5 Results

For valid inputs (e.g. Table 1), the program produces correct graphical and numerical outputs. Trajectories remain near equilibrium when initial conditions are close to $e_2 = \left(\frac{D}{C}, \frac{A}{B}\right)$. Degenerate cases (zero populations) are handled consistently, without generating new populations.

For sufficiently small time steps, the discrete-time model qualitatively reproduces the expected closed orbits of the continuous system, with minor numerical energy drift due to the time discretization. This behavior is consistent with the qualitative properties of the Lotka–Volterra system.

Graphical output (e.g. Figure 1) shows energy deviations via point color; CSV files store the full simulation history.

6 Testing

Unit tests use the Doctest framework to validate the numerical core, I/O and rendering. **Simulation tests** check initialization, parameter validation, time evolution, extinction scenarios, energy conservation, convergence and numerical accuracy.
Renderer tests verify window size validation, execution without errors and handling of

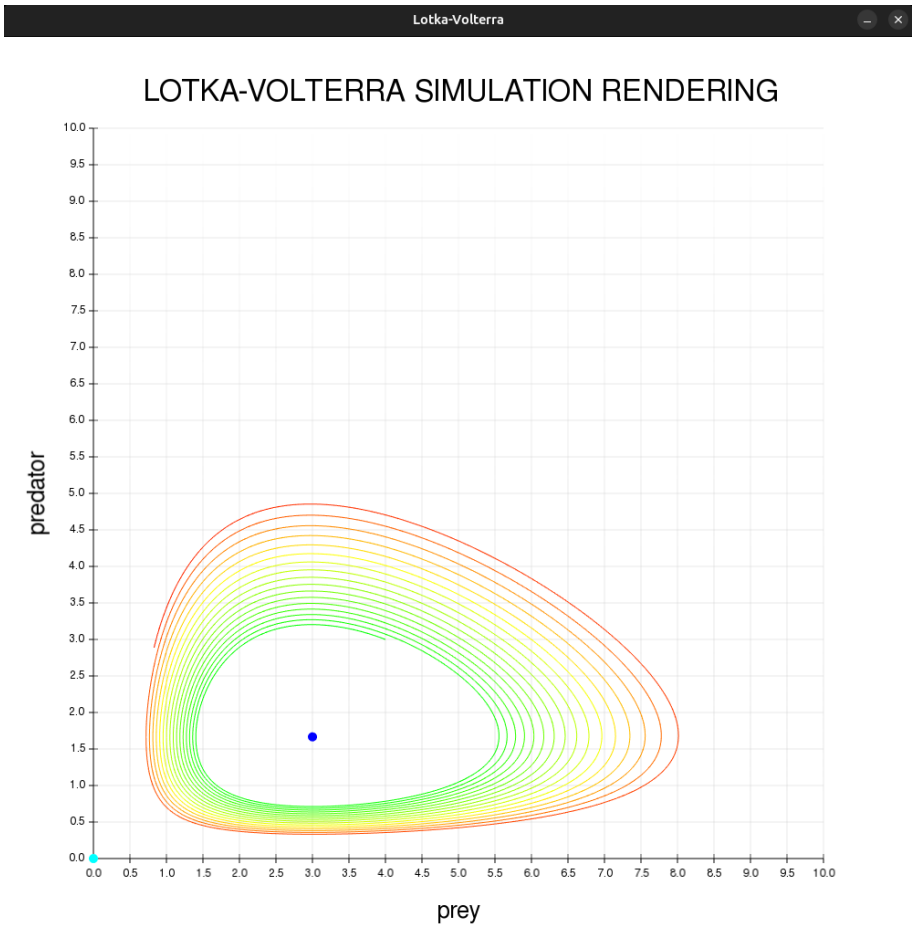

Figure 1: example of rendered trajectory from input values in Table 1

full/partial trajectories.
I/O tests ensure correct construction of objects and CSV export.

Tests are executed automatically during the build process, ensuring immediate detection of regressions and implementation errors. Together, these tests provide confidence in the correctness, robustness and numerical reliability of the implemented model and its supporting components.

7 Use of AI

Artificial intelligence tools were used as auxiliary support during the development of the project.

AI assistance was employed to help resolve specific implementation issues in the graphical rendering subsystem (e.g. view handling and coordinate scaling) and to obtain minor guidance on `CMakeLists.txt` modifications aimed at avoiding compilation errors.
AI was also used for support with technical English, as well as for occasional help with LaTeX and Markdown syntax, as the project and its documentation are also hosted on GitHub [1].

All design choices and final implementations were independently reviewed and validated by the author.

References

[1] Project GitHub repository