Volterra Predator-Prey Simulator in C++

Course "Programming for Physics" — Physics Degree, University of Bologna

Giulia Montagnani

Last code modification: August 18, 2025
GitHub Repository

Changes Since Previous Submission

This is the first version of the project; no modifications since any previous submission.

Project Description

This project implements a numerical simulation of the Volterra predator-prey model in C++. The simulation employs a discrete method to solve the coupled differential equations over a specified time interval and generates output files representing the evolution of the populations.

Detailed theoretical explanations of the model are beyond the scope of this document. For more information, please refer to the Lotka–Volterra equations on Wikipedia or any standard textbook on differential equations.

Design and Implementation Choices

The program is split into multiple translation units, with the objects and functions implemented placed inside the project namespace `pf`, following a modular structure to make the code easier to maintain and extend.

The main modules are:

volterra.hpp / volterra.cpp – Simulation Logic

This module implements the discretized Lotka–Volterra equations inside the `Simulation` class. The class is responsible for validating parameters, keeping track of the system state, and evolving it numerically over time.

The data structures used are:

- `SpeciesCount` – minimal struct storing prey and predator counts, used internally to store relative counts;
- `SpeciesState` – extends `SpeciesCount`, adding system energy H and time t, representing the type returned to the user;
- `Parameters` – holds the four constants of the ODE system, stored as a private `const` member in each simulation instance.

The free functions `randomParams()` and `randomInitialConditions()` generate random values for the model parameters and initial populations respectively, using the `std::random` library. Both functions accept optional `min` and `max` arguments to specify the range of random values. If no arguments are provided, the predefined `constexpr` constants from `simulation_opt.hpp` are used as default bounds.

This design provides flexibility during testing, while the control for valid (positive) values is delegated to the `Simulation` constructor.

For efficiency, relative population counts are stored internally in the private member `rel_counts_`, kept separate from absolute states (which are computed only when needed). This avoids unnecessary calculations. Parameters and initial conditions are validated during construction using internal checks based on the `validatePositive()` function. The class throws exceptions if any invalid values are provided.

The `Simulation` class stores `dt_` as a private member, which represents the duration of a single time step in the discretized simulation. In the program, the default constructor value is used in all cases except during testing. Keeping `dt_` as a private member makes testing easier, which is why it was chosen over a global constant.

Helper methods handle conversions between relative and absolute values, as well as computing the first integral H. The class offers two constructors: the second one, which takes separate values instead of a struct, is useful in the test files for improved readability.

Public methods of `Simulation` include:

- `run(double duration)`: runs the simulation for the given duration, calling the private `evolve()` method at each step, which stores the computed relative values in `SpeciesCount` format inside `rel_counts_`. Returns the number of steps and adjusted duration of the run.
- `getAbsStates()`: returns all discrete system states for the current simulation in the form of a `SpeciesState` vector, with prey and predator counts in absolute values.
- A templated `getSeries()` method to extract specific state members, mainly for plotting in the `graph_renderer` module.

Additional getters such as `getDt()`, `getParams()` and `numSteps()`, as well as the default assignment operator, are provided. The class also includes sanity checks during evolution, raising exceptions if populations become non-positive, which should not happen according to the model assumptions.

utils.hpp / utils.cpp – Utility Processing

This module provides helper functions outside the simulation's core logic, implemented as free functions in the `pf` namespace for reuse without adding responsibilities to `Simulation`. They mainly use Standard Library functions and in this project are used to simplify `main`.

Note: Throughout this section, method names are presented without explicit parameter lists for readability.

- `executeSim() -` Allows simulations to return warnings (e.g., duration shorter than `dt`, time rounding applied) through `ExecSimResult` struct, which stores a boolean and a message.
- `generateRandomSim()` — Builds a `Simulation` object using random parameters and initial conditions.
- `writeOnFile()` — Outputs aligned, fixed-width columns for readability.
- `visualizeResult()` — Delegates rendering to `GraphRenderer`, isolating SFML-specific code.
- `askInput()` — Template function for validated user input, with optional min/max limits.
- `formatNumeric()` — Template function for consistent floating-point formatting.

graph_renderer.hpp / graph_renderer.cpp – Visualization

The `GraphRenderer` class handles visualization of simulation results using the SFML Graphics library. It supports two main plot types: time series (population changes over time) and phase-plane orbits (predator-prey relationships). These are managed through an enumeration (`PlotConfig::Type`), making it easy to add new plot types later.

Rendering is split into focused private methods for drawing axes, time series curves, and orbits. Axes include ticks, labels, and titles, with dynamic scaling to fit different data ranges. Consistent margins and coordinate mapping ensure uniform scaling and layout across all plots.

Fonts are loaded once from the project `./assets` directory to keep text styling consistent. The class can draw a single plot or a combined view, splitting the window to show both time series and orbits together.

A `writeToFile()` method is also implemented, separated from the visualization methods.

Additional files

- `main.cpp` – Manages user interaction, sets simulation parameters, and coordinates execution flow. Its design focuses on coordination and exception handling, making it easy to extend or modify user interactions without affecting the simulation or rendering modules.

- `volterra.test.cpp` – Unit tests for class `Simulation` components and related functions, written with the `doctest` framework (see section Testing Strategy). Some private methods of the class are tested indirectly through the public interface to ensure correctness in any part without exposing implementation details.

- `simulation_opt.hpp` – Stores all default parameters, limits, and formatting settings for the simulation. This centralizes configuration, making it easy to adjust model constants (bounds, intervals, time step, etc.). It also specifies the physical meaning of parameters for the predator-prey model.

- `output_opt.hpp` – Stores the visualization options for graphs and text output files. Some examples include:

  - A boolean `combined` to specify whether time series and orbit plots should be displayed and saved together in a single window or separately.
  - A float `display_time` indicating the number of seconds for which the graphical windows remain open before closing automatically.
  - The font path `font_path` used to render text in the plots.

  as well as additional parameters that configure the appearance and layout of the output plots.

- Support files – `doctest.h` for testing and `.clang-format` for coherent formatting throughout the program.

- Font resources – Located in `assets/Open_Sans`, sourced from Google Fonts for text rendering. The location is editable through the `output_opt.hpp` file.

Some general principles followed for designing the program are:

- Splitting the code into simulation logic, utilities, and visualization to enforce separation of concerns.

- Custom types (e.g., `SpeciesCount`, `SpeciesState`, `Parameters`) are used instead of generic tuples or pairs for better code readability and meaning.

- Templates (e.g., `getSeries()`) help reduce repeated code and improve flexibility.
- Naming conventions for clarity and quick recognition of roles: `CamelCase` for class names, `camelCase` for member functions, `snake_case` for free functions, variables, and file names.

How to Run the Program

The project uses *CMake* as a build system generator, in particular relying on the *Ninja build system.* On Ubuntu 24.04, if *Ninja* is not already installed on the device, please run:

```
sudo apt install ninja-build
```

before building the project.

External Libraries

The project uses the SFML library for graphics.You can install it using the command line:

```
sudo apt install libsfml-dev
```

To succesfully run the program, the following commands must be executed in the specified order:

1. CMake has to be initialized using:

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

2. The program can be built using one of the following commands:

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

   depending on the configuration desired between the availables from CMake (`Debug` or `Release`); the `-target test` command execute the tests implemented (see section Testing Strategy), compiling only the files necessary to do this as specified in the `CMakeLists.txt` file.

   These commands generate build files in the `./build` directory.

3. To run the compiled executable `volterra`, after building, execute:

   ```
   ./build/Debug/volterra    % for Debug build
   ./build/Release/volterra  % for Release build
   ```

   Additionally, unit tests are compiled into a separate executable named `volterra.t`. This can be run to verify the correctness of the program:

   ```
   ./build/Debug/volterra.t     % Run tests in Debug mode
   ./build/Release/volterra.t   % Run tests in Release mode
   ```

   If you compiled with `-target test`, you can also consult the detailed logs about the tests executed in the `./build/Testing/Temporary` directory, for example with commands:

   ```
   cd build/Testing/Temporary/
   cat LastTest.log
   ```

Input Parameters and Output Format

Some parameters regarding all parts of the program are not requested at runtime but can be modified by editing the headers `output_opt.hpp` and `simulation_opt.hpp`.

These parameters allow the user to easily adjust the output format, graphical layout, and visual clarity of the simulation results, as well as model details, without modifying the core code.

When the program is executed, the user is asked to choose whether to run the simulation with default parameters, random parameters or to provide custom values.

- In *default mode*, the program instantiates a `pf::Simulation` object with default coefficients and initial populations.
- In *random mode* the program instantiates a `pf::Simulation` object with random coefficients and initial populations.
- In *custom mode*, the user is prompted to enter six values:
  - a, b, c, d (Lotka–Volterra model coefficients)
  - Initial prey population
  - Initial predator population

All these parameters must be positive real numbers, otherwise the program returns an error and exits. Non-integer values for prey and predator populations are allowed, since they represent population density rather than discrete counts.

The user is then asked to provide the simulation duration (in arbitrary time units). If it exceeds the maximum allowed `max_duration`, the program aborts with an error message.

The output is generated in three forms:

1. **Console messages** — status updates (labelled as `[Info]`), warnings, and error messages.
2. **Data file** — the computed simulation results are written to a `.txt` file in the directory `./results`, which also reports he number of discrete steps performed.
3. **Graphical visualization** — a graphical window displaying the predator–prey population curves over time with the first integral H and/or their orbits using SFML. The output is saved to a `.png` file in the directory `./results`.

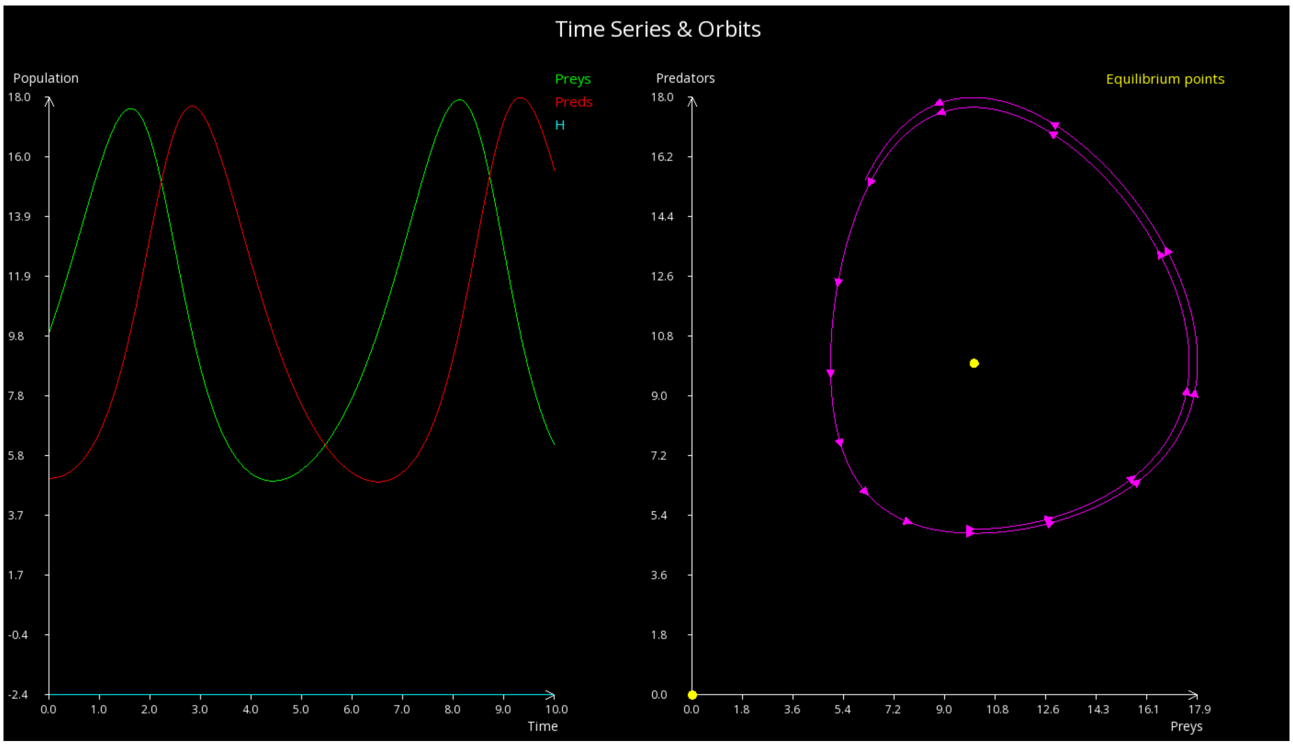
Example run with default parameters:

```
$ ./build/Release/volterra
Run simulation with default values? [y/n]
Press [q] to run a random simulation instead.
Press any other key to exit the program.
y

[Info] Default simulation values set.

Insert duration for the simulation: 10
[Info] Simulation successfully executed.

[Info] Results written to file.

[Info] Visualization in progress. Close the graphic window to terminate the execution.
--- user closes window or time elapses ---
```

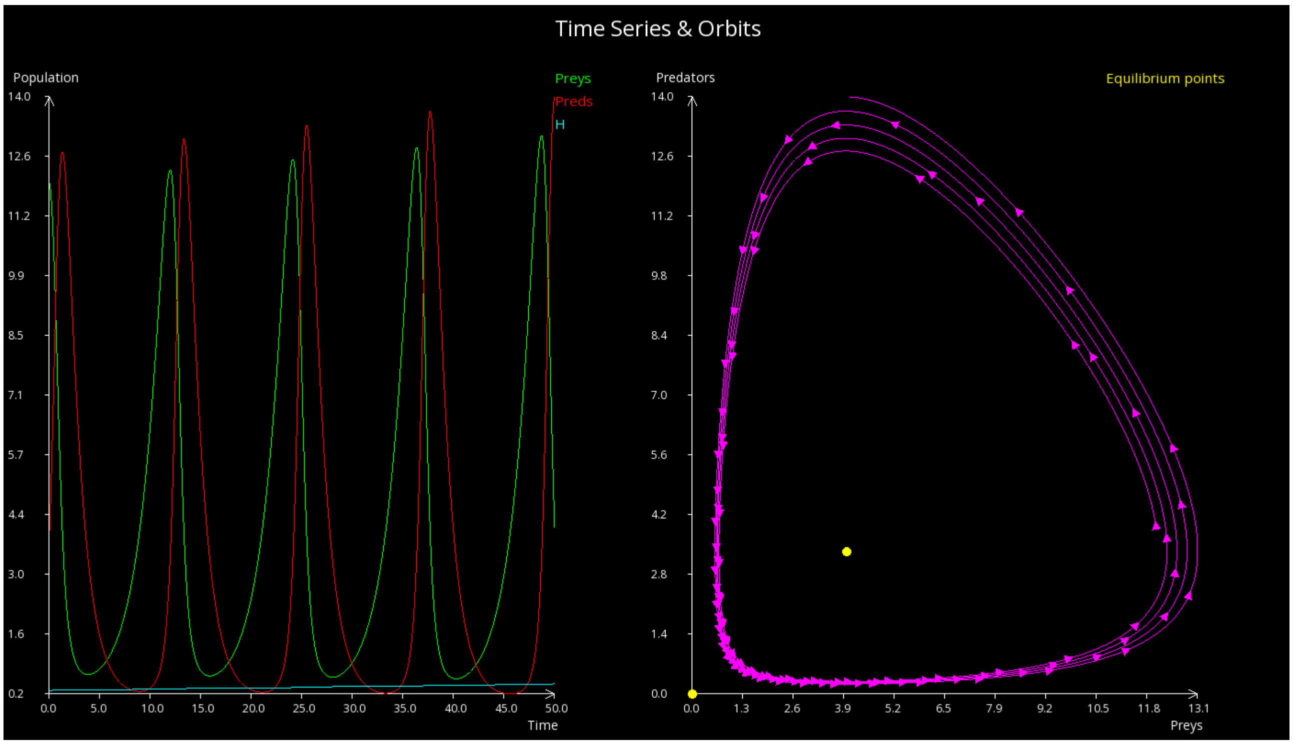
```
[Info] Visualization completed. Image saved to file.

Exiting program.
```

Example run with custom parameters:

```
$ ./build/Release/volterra
Run simulation with default values? [y/n]
Press [q] to run a random simulation instead.
Press any other key to exit the program.
n

Insert the following values for simulation:
a (default 1): 0.5
b (default 0.1): 0.15
c (default 0.1): 0.2
d (default 1): 0.8
Initial number of preys (default 10): 12
Initial number of predators (default 5): 4

[Info] Custom simulation parameters set.

Insert duration for the simulation: 50
...
```



Graphic output - Example run with default parameters



Graphic output - Example run with custom parameters

Result Interpretation

The simulation reproduces known results from the model theory:

- **Graphical visualization** — orbits are closed in the phase-plane for appropriate duration, centered around $(\frac{d}{c}, \frac{a}{b})$.
- **First integral conservation** — the first integral H remains roughly constant; small drift occurs in long simulations due to numerical errors, affecting orbit amplitudes.
- **Time series** — prey and predator populations oscillate alternately when solutions are near equilibrium.
- **Numerical stability** — unbalanced parameters or large time steps can cause deviations from the model, emphasizing the need for an appropriate dt.
- **Population positivity** — prey and predator counts remain positive, ensuring physically meaningful results through implemented checks.

Testing Strategy

The testing strategy aims to ensure the correctness, stability, and robustness of the `Simulation` class, guaranteeing that the simulation produces physically meaningful results under a wide range of conditions.

All tests were implemented using the `doctest` framework and cover the following aspects:

- **Constructor behavior** — both default and parameterized constructors are verified for correct initialization. This includes proper computation of relative and absolute population values and the correct evaluation of the first integral H.

- **Parameter validation** — all model parameters are checked to prevent null or negative values, with exceptions thrown when invalid parameters are provided.
- **Initial conditions validation** — initial population values are required to be strictly positive; invalid conditions trigger exceptions.
- **Time step validation** — the simulation time step dt is verified to avoid null or negative values.
- **Simulation evolution** — the `run()` method is tested for both single-step and multi-step scenarios to ensure correct updates of prey and predator populations, positivity preservation, and acceptable numerical drift in H.
- **Extreme conditions** — simulations with large or highly unbalanced parameter values, large time steps, and extreme initial populations are tested to ensure numerical stability and physically meaningful outputs within the allowed ranges.
- **Series extraction** — the template method `getSeries()` is verified to extract consistent time series for populations, the first integral, and time, ensuring correct alignment with `getAbsStates()`.
- **Individual utility functions** — helper functions, such as `validatePositive()` and random generation ones, are tested independently to guarantee correct exception handling and robustness.

Use of Generative AI

The generative AI system **ChatGPT (OpenAI GPT-5)** was employed exclusively as support in the following aspects of the work:

- Debugging the original code.
- Providing feedback on the implemented solutions, which are original ideas.
- Researching appropriate methods in the C++ Standard Library.
- Supporting the choice of nomenclature and checking compliance with the project's coding conventions, following the C++ Core Guidelines.
- Drafting the skeleton code for the `Graph Renderer` files, which were then independently modified to implement the desired functionalities. No generative AI tools were used to write other parts of the code from scratch.
- Formatting in LaTeX and improving the style of this report.

Additional Notes

Guideline Version and Interactive Version

The version described in this document follows, as closely as possible, the guidelines specified for the exam project.

A modified version of the code, which includes a proposal for an interactive and looped menu (the program does not terminate automatically when an invalid input is entered), is available in the GitHub repository on the branch `interactive-version`.