Lotka-Volterra

Mira Zoffoli

A.A 2023-2024
Programmazione per la Fisica

Indice

1 Progetto

Scopo del progetto è la costruzione di un programma che simuli l'interazione tra due specie che coesistono in un ecosistema e sono in competizione tra di loro. La prima delle due specie (indicata con x(t) a ogni istante di tempo) è quella delle prede, mentre la seconda specie (y(t)) rappresenta i predatori, che utilizzano, come fonte di nutrimento, le prede.

1.1 Descrizione del sistema

Indicate prede e predatori in un certo istante t rispettivamente con x(t) e y(t), possiamo descrivere il loro andamento nel tempo attraverso le seguenti equazioni differenziali:

$$\frac{dx}{dt} = (A - By(t))x(t)$$

$$\frac{dx}{dt} = (Cx(t) - D)y(t)$$

Dove:

- A indica il tasso di riproduzione delle prede
- B indica il tasso di mortalità delle prede a causa dei predatori
- C indica il tasso di riproduzione dei predatori
- D indica il tasso di mortalità dei predatori. La morte dei predatori è dovuta alla mancanza di prede.

A e C sono da intendersi "in condizione di sostentamento", cioè a patto che ci siano sufficienti risorse per supportare il loro naturale processo di riproduzione. Ciascuno di questi parametri deve essere > 0.

In generale, queste equazioni sono utili per l'analisi di un ecosistema in cui due specie sono in competizione per una stessa risorsa, in modo che più una specie riesce a utilizzare la risorsa meno riesce a farlo l'altra.

Nelle simulazioni al calcolatore di solito si utilizza la seguente versione discretizzata delle equazioni:

$$x_i = x_{i-1} + (A - By_{i-1})x_{i-1}\Delta t$$

$$y_i = y_{i-1} + (Cx_{i-1} - D)y_{i-1}\Delta t$$

1.2 Dinamica del sistema

Partendo da uno stadio iniziale con x(0)$>$ 0 e y(0)$>$ 0, le orbite del sistema sono ristrette a valori positivi di x(t) e y(t). Questo vincolo implica che non esista un cambiamento di segno delle popolazioni nel tempo (viene dunque mantenuto un equilibrio dinamico).

Dalla soluzione del sistema di equazioni differenziali risultano essere due i punti di equilibrio:

- $e_1 = (0, 0)$, questo stato rappresenta l'assenza di prede e predatori nell'ecosistema.
- $e_2 = \left(\frac{D}{C}, \frac{A}{B}\right)$, questo è il punto in cui entrambe le equazioni differenziali sono soddisfatte simultaneamente senza che le popolazioni assumano valori nulli.

Un aspetto chiave del sistema Lotka-Volterra è la presenza di un integrale primo, ovvero una funzione differenziabile che rimane costante lungo le soluzioni del sistema.
Nel caso di Lotka-Volterra, l'integrale primo è associato alla conservazione di una quantità che rappresenta una forma di "energia" del sistema. Tale concetto è analogo al principio dell'energia totale nei sistemi meccanici sotto l'azione di forze conservative.

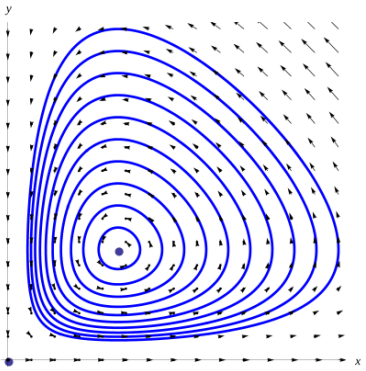

Figure 1: Spazio delle fasi delle equazioni di Lotka e Volterra.

Il punto centrale corrisponde al minimo della funzione H ed è un punto di equilibrio stabile. In tutti gli altri punti il sistema oscilla in continuazione su varie curve di livello. Il grafico in Figura 1 mostra quindi come variano prede e predatori nel corso del tempo. Ci aspettiamo che la nostra simulazione porti a un risultato analogo, concorde alla teoria.

1.3 Obiettivi del codice

Il programma ha l'obiettivo, forniti in input i valori iniziali di x e y, unitamente ai valori dei 4 parametri A, B, C, D che si vogliono utilizzare, di sviluppare una simulazione che utilizzi le equazioni discretizzate di Lotka-Volterra per calcolare, ad ogni step della simulazione, la terna (x_i, y_i, H_i). A partire da questi dati il codice traccia un grafico.

2 Implementazione

2.1 Struttura generale del codice

Il codice è organizzato come segue:

- `Simulation.cpp` e la relativa interfaccia definita in `Simulation.hpp` modellano la classe Simulation, che descrive una singola simulazione e ne mantiene al proprio interno lo stato.
  Un oggetto `Simulation` espone metodi per mandare avanti la simulazione di uno step e visualizzarne i risultati.

- `test_simulazione.cpp` contiene i test unitari per testare il funzionamento della classe Simulation.

- `main.cpp` è il punto di ingresso del programma: crea l'oggetto `Simulation`, manda avanti la simulazione un passo alla volta e stampa a schermo e in un file i risultati intermedi.

`Simulation.cpp`, `Simulation.hpp` e `main.cpp` appartengono alla directory `src`, mentre `test_simulazione.cpp` alla directory `test`.

2.2 Dettagli di implementazione

La classe Simulation, definita all'interno dell'header file `Simulation.hpp` e implementata in `Simulation.cpp`, è il focus principale del progetto; essa gestisce la simulazione dell'interazione tra prede e predatori. La classe contiene i seguenti attributi e metodi principali.

Attributi:

- `double current_x`: il numero attuale di prede.

- `double current_y`: il numero attuale di predatori.

- `double A, B, C, D`: i parametri delle equazioni di Lotka-Volterra, inseriti come input.

- `double t`: il tempo corrente della simulazione.

- `double dt`: il passo temporale della simulazione.

Metodi:

- `Simulation(SimulationParameters const& params)`: costruttore che inizializza la simulazione attraverso i parametri che vengono forniti. I parametri vengono passati raggruppati in una struct di nome `SimulationParameters`.
- `void evolve()`: seguendo l'andamento dettato dalle equazioni di Lotka-Volterra, aggiorna i valori di current_x e current_y.
- `double get_relative_x(), double get_relative_y()`: calcolano i valori relativi rispettivamente di `current_x` e `current_y` rispetto ai loro punti teorici di equilibrio.
  Il metodo inoltre controlla che i valori di x e y non diventino mai negativi, lanciando una eccezione nel caso in cui ciò accada.
- `double get_H()`: calcola l'integrale primitivo H(x, y).
- `std::string print_info()`: ritorna una stringa formattata che rappresenta lo stato attuale della simulazione.

In particolare, il metodo `evolve()` è fondamentale in quanto aggiorna lo stato delle popolazioni di prede e predatori ad ogni istante della simulazione: al suo interno infatti, come riportato in precedenza, vengono utilizzate le equazioni discretizzate di Lotka-Volterra.

```
1  void Simulation::evolve() {
2   t += dt;
3
4   // Aggiorno i valori di x e y
5   double old_x = current_x;
6   double old_y = current_y;
7
8   current_x += (A * old_x - B * old_x * old_y) * dt;
9   current_y += (C * old_x * old_y - D * old_y) * dt;
10
11  // Controllo che il valore corrente di x e y non sia
        negativo
12  if (current_x <= 0 || current_y <= 0) {
13    std::cerr << "Errore: i valori di x e y non possono
          diventare zero o negativi.\n";
14    throw std::runtime_error(
15        "Invalid state: x and y cannot be zero or
            negative");
16  }
17
18  // Controllo che il valore corrente di x e y non sia
        infinito
19  if (std::isinf(current_x) || std::isinf(current_y)) {
```

```
20      std::cerr << "Errore: overflow durante la
            simulazione.\n";
21      throw std::runtime_error("Overflow during
            simulation");
22    }
23  }
```

All'interno del file `main.cpp` è stata invece gestita l'interazione con l'utente da riga di comando.

In particolare questo file:

- gestisce l'inserimento dei parametri iniziali da riga di comando, raggruppati in una struct `SimulationParameters`. I parametri possono anche essere passati al programma come argomenti di invocazione.
- istanzia un oggetto `Simulation` che rappresenta lo stato iniziale delle popolazioni di prede e predatori, oltre ai parametri del modello Lotka-Volterra.

Una volta inizializzata dunque, la simulazione procede per un numero specificato di passi (steps, decisi a loro volta dall'utente). Ad ogni passo, la simulazione fa in modo che le popolazioni di prede e predatori si evolvano utilizzando il metodo `evolve()` della classe `Simulation`, come precedentemente evidenziato.

In `main.cpp` si trova inoltre l'inizializzazione di una finestra grafica, componente ottenuta tramite l'utilizzo di SFML (Simple and Fast Multimedia Library, una libreria multimediale open-source progettata per facilitare lo sviluppo di applicazioni interattive e grafiche).
In particolare, SFML è qui utilizzata per creare e gestire una finestra grafica che visualizza dinamicamente l'evoluzione delle popolazioni nel tempo.

La finestra è stata configurata con una risoluzione basata sul 90% dell'altezza del desktop dell'utente, mantenendo un rapporto proporzionale.
La finestra rimane aperta fino a che l'utente non decide di chiuderla, gestendo gli eventi di input attraverso la funzione `pollEvent`.

Quando sono disponibili, i dati di simulazione sono letti dal file `"simulation_output.txt"` e vengono utilizzati per disegnare forme grafiche. In questo caso i dati delle popolazioni sono rappresentati come cerchi di ridotte dimensioni, posti all'interno della finestra in maniera proporzionale ai valori di x e y.

2.3 Repository su github

Il progetto è disponibile su GitHub a questo indirizzo (https://github.com/mirazoffoli/Lotka-Volterra). Poichè il progetto non è stato sviluppato in gruppo, l'utilizzo di Git e GitHub è stato utile verso la fine della scrittura per tenere più organizzati gli ultimi sviluppi del codice.

3 Logica dei test

I test unitari sono stati implementati attraverso l'utilizzo della libreria doctest e cercano di coprire multipli aspetti della simulazione, in modo tale da avere una garanzia riguardo al corretto funzionamento del codice e della sua gestione dei casi limite.

In particolare, si è scelto di eseguire i seguenti test:

- Controllo dei parametri iniziali della simulazione per prevenire l'inizializzazione di essa con valori non validi.
- Verifica dell'evoluzione delle popolazioni nel tempo, assicurandosi che non diventino negative o infinite.
- Calcolo dei valori relativi e dell'integrale primitivo per confermare la precisione dei calcoli.
- Test per situazioni limite, come parametri negativi o passi temporali nulli.
- Verifica della corretta generazione della stringa di stato della simulazione.

4 Analisi risultati

Impostando una simulazione con i seguenti parametri:

```
initial_x = 1.8;
initial_y = 4.3;
A = 0.1;
B = 0.02;
C = 0.05;
D = 0.1;
dt = 0.001;
steps = 90000;
```

è possibile fare una analisi dei risultati ottenuti: questa è necessaria per comprendere la correttezza (o meno) del codice.

Durante i 90000 step della simulazione, le coordinate x e y hanno mostrato le seguenti variazioni:

- x varia tra un massimo di 2.38188 e un minimo di 1.66135.
- y è varia tra un massimo di 5.95446 e un minimo di 4.15331.

Questi valori rappresentano un chiaro andamento oscillatorio nel tempo per entrambe le coordinate, come indicato dai picchi e dalle valli osservati nei dati.

Tale comportamento è coerente con le previsioni teoriche e suggerisce che il sistema simulato sia dinamico e reagisca in modo sensibile ai parametri impostati.

5 Compilazione del progetto

5.1 Struttura

Il progetto è organizzato in una struttura che include la directory principale e le seguenti sottodirectory e file significativi:

- **src/**: Contiene i file sorgente del programma principale.
- **test/**: Contiene i file di test unitari.
- **CMakeLists.txt**: File di configurazione per CMake, utilizzato per la generazione dei Makefile.
- **.gitignore**: per indicare al sistema di controllo versione (Git) quali file ignorare.

5.2 Compilazione del codice sorgente

- **Creazione della Directory di Build**: All'interno della directory principale, creare una directory denominata "build".
- **Configurazione con CMake**: Utilizzare il comando `cmake ..` per configurare il progetto. In CmakeLists.txt sono specificate le dipendenze e le opzioni di compilazione in base alle quali vengono generati i file necessari per la compilazione.
- **Compilazione del Codice**: Utilizzare il comando `make` per compilare il codice sorgente e generare l'eseguibile.

5.3 Esecuzione del Programma Principale e dei Test Unitari

Una volta compilato con successo, è possibile eseguire il programma principale attraverso il comando `./main` ed i test unitari con il comando `./test_simulazione` .