Title: Visualizing *C. elegans* Population Dynamics

Abstract:

Understanding population dynamics of the model organism *Caenorhabditis elegans* (*C. elegans*) is an important step in understanding the general principles that drive populations in fluctuating environments. Our team introduces a novel software tool designed to improve the accessibility and interpretability of *C. elegans* population dynamics simulations. This tool complements an existing simulator, systematic tracking each worm’s progression through developmental stages such as egg, larva, adult, dauer, and parlad as well as death.

Featuring a user-friendly web interface, the software caters to both tech-savvy and non-technical users. It enables multiple simulation instances, allows manipulation of run-time parameters, and provides functionality for downloading and graphically presenting simulation data. This interactive approach empowers users to explore diverse scenarios and parameters, fostering greater comprehension of *C. elegans* population dynamics.

Introduction

WormPOP is a multiagent simulation that simulates the life history of *C. elegans*. The worms are born and transition through the egg, larval adult, parlad, and dauer stage until they die [Scharf et al., 2022]. The software transforms simulation data into a visually intuitive representation, using a directed graph. This graph not only delineates *C. elegans* life cycle transitions but also dynamically adapts to illustrate population density at each stage. Node sizes reflect the number of worms in respective life stages, while directed arrows adjust proportionally to signify transition volumes between stages.
Our program deals with simulation data instead of data involving live *C. Elegans*. But why do we need simulations? Simulations are useful for several reasons. The main reason is that experiments that may take several weeks/months in the lab can be simulated in a few hours. We can run multiple simulation experiments simultaneously if computational resources are available. The fact that we can run multiple simulations makes it easier to test certain starting conditions.

Simulations also have disadvantages because simulations are ultimately approximations, and we must make a lot of assumptions about the real-world system. And of course, simulations are computationally intensive. Simulations produce huge amounts of data that need to be stored, organized and interpreted. Running a simulation requires some familiarity with using a command line interface. A command line (or terminal) can be intimidating without training.
Most people do not interact with a computer through a command line. Instead, a web application is the standard way for users to interact with computational services. To solve the problem of organizing and storing data, we created a web server that manages each simulation instance. To enhance usability, we created an application that runs in a web browser with an intuitive interface that visualizes the data generated by the simulation.

By mitigating technical barriers to obtain high-quality simulation data, our objective is to enhance the overall accessibility of scientific research, fostering an inclusive environment for exploration and discovery in the biological sciences.

**Methods**

A Web application has two components. Client-side code and Server-side code. Client-side code runs in a web browser, also known as the client. Server-side code runs remotely in a central web server. A web server can distribute data to multiple clients.

**Server Specification**

Our web server runs multiple instances of a simulation according to user specified parameters. We also need to monitor the process executing the simulation in case the user wants to stop it. We also need to stop the simulation and delete the data. Operating systems track processes by a process id, also known as a PID. However, tracking simulations using the process ID is not reliable since multiple processes can be assigned to the same process ID in the event the server is rebooted. We assign a simulation instance
to a GUID, a long string of characters that is randomly generated. We store these on a table.

To expose operations that can be done on an operation, we expose API routes. An API stands for an Application Programming Interface. It allows external agents like a user agent to perform operations to data stored in a web server. A web server exposes an API through routes. Each route has a set of associated verbs. For example, the GET request performed at the “/” route retrieves the home page of a web server. This is known as the index page. We expose several routes that perform the following operations:

1. **Create a simulation**
   a. GET ‘/’
      i. Retrieves the page that allows user to submit simulation
   b. POST ‘/run-simulation’
      i. Creates a new simulation instance based on the data submitted by the user
      ii. Returns a GUID to the caller that uniquely identifies the simulation

2. **Stop a simulation by killing the process running it**
   a. POST ‘/stop-simulation/<guid>’
      i. Here <guid> represents the simulation instances
      ii. This GUID is returned by the call to ‘/run-simulation’

3. **Delete a simulation by deleting all the data simulation generates**
   a. POST ‘/delete-simulation/<guid>’
      i. Here <guid> represents the simulation instances
      ii. This GUID is returned by the call to ‘/run-simulation’

**Client-Side specification**

The client’s purpose is to provide a user interface for the user to interact with the simulation processes. It also provides the means for the user to visualize the data.

- Pull simulation results from the server and graph it
- Display all running simulation instances
- Start/Stop/Delete simulation
- Graph all simulation data as a function of time
- Animate the stage transitions
Results

The home page of our application contains the forms necessary to submit a new simulation instance. To make sure our application is extensible this web page is dynamically generated from the simulation parameters defined in ‘contants.json’.

![Parameter Table]

*Fig 3. The user can adjust the starting conditions of the simulation before pressing the “Run Simulation” button at the bottom. The user can also give a name for the simulation.*
After the user presses run simulation, they are redirected to a table that shows all simulation instances. From here they can control whether to stop or delete the simulation. Stopping the simulation simply kills the process running it. All data is preserved. Deleting the simulation also deletes all the data.

Fig 4. After the user submits a simulation, they are shown this table which lists all the simulations the user has previously submitted. It contains the Job ID, which uniquely identifies each instance. The Name is supplied to the user and helps to remind the user of the parameters they changed. It contains the start time, and how long the simulation has been running for. The status indicates whether the simulation is Running or Stopped. The Action bar contains the operations that can be done to the simulation. For example, Download will create a zip file of all the data generated by the simulation.

The graph button allows the user to visualize the relative population sizes of all worm life stages. This graph was generated from a DOT file, a file format that is used by the ‘dot’ program to generate graphs. The DOT file is turned into an SVG file, which is sourced by the web application and loaded into the browser. The browser uses JavaScript to manipulate the SVG elements, which changes the size of the nodes and edges.
Fig. 5. This is an animated graph. The user can start the animation, stop the animation, reset it, or go through it step by step. The user can also choose a specific timestep and visualize the relative population sizes at each time step.

The next feature is the data dashboard. It allows the user to visually compare the results between multiple runs of the simulation. The simulation generates four aggregate files.

1. `summary.tsv` - This file contains a summary of the simulation at each time step. For example, how much food is currently in the system and the total number of worms. This is an aggregate file, so it counts all the existing entities at each time step.
2. `stage_transitions.tsv` - This file counts the individuals that are transitioning from one stage to the next. W
3. `death_transitions.tsv` - This file records the exact manner of death of each worm
4. `variant_count.tsv` - This file records the number variants per worm.
Fig 6. Section (a) Contains buttons for all the simulation instances. Section (b) contains the attributes for each data file. Section (c) contains the parameters for each simulation instance. The row is highlighted yellow if the parameters are different. Section (d) is the graph.

Each file gets its own card in the dashboard. The user can plot all the columns of each data file and compare it against each instance of the simulation. This makes data interpretation a lot easier because the user can instantly see any differences a parameter will cause. In the Figure above, we compare two simulation instances. The default simulation (Test3) and a simulation instance where we cull 90% of eggs at specified intervals. As we can see in the plot, when we remove eggs from the population, more worms die of old age. This makes sense since older worms have access to more food since fewer eggs mature to larval stage.
Technical Stack

In software development, the technical stack, or tech stack, is the set of tools used to develop an application. This section describes the tools we used to write our application.

1. Git – Git is a version control software that allows collaboration between individuals. The most powerful feature of git is branching. The ‘main’ or ‘master’ branch is the official version of the code. A branch is a parallel copy of the code that can be edited without affecting the main branch. Upon approval, the branch version can be merged into the main branch.

2. Flask – Flask is a simple backend server framework that allows the user to specify an action when a route is indicated.

3. KnockoutJS – A simple frontend framework that allows the user to define a data model, then bind the HTML to the data model. Changing the data model also changes the HTML. The idea is to separate the data layer from the presentation layer.

4. Tailwind CSS – A styling library that allows user to define styling attributes as shorthand in the HTML

5. Plotly.JS – A graphing library in javascript.
Discussion

Data visualizations are crucial in research because they turn complex data into understandable graphics. When research is showcased through a web application, it becomes more accessible. Users can interact with the data, like adjusting variables to see different outcomes, making the science more hands-on. This interactivity also simplifies tasks like running simulations, which might otherwise require advanced computational skills. For researchers seeking funding, a website that clearly displays their work's impact through visualizations can help potential funders grasp the significance of their research easily, enhancing their chances of securing financial support.

Studying population dynamics, especially through simple organisms like C. elegans, provides valuable insights into broader biological processes, including those in more complex, long-lived species like humans. Despite its simplicity, C. elegans shares essential biological characteristics with humans, making it an effective model for understanding aging and demographic changes. By researching the conditions under which C. elegans begins to show signs of aging and mortality, scientists can unravel the underlying mechanisms of aging, potentially applying these findings to human health and longevity. This research not only advances our knowledge of life span determinants but also contributes to our understanding of population dynamics and demographic shifts in larger ecosystems.

Acknowledgements

This section details the individual contribution of all members of the OURE team.

1. **Dr. Andrea Scharf – Principal Investigator**: Dr. Scharf advised the development of the application and provided the original code for the Python simulation. She suggested features that need to be added.

2. **Kevin Lai – Technical Lead**: To bring the web application to life, Kevin organized team coding sessions once a week to develop the code. He designed the system architecture and taught the other OURE members how to use the tech stack. Kevin created some of the backend server routes and designed the database schema to map process to simulation. He also created an animated graph with nodes that changes size based on population size.

3. **Yug Patel – Frontend/Backend Developer** – Yug created the API routes to start, delete, and download the simulation jobs. He integrated Plotly.JS with the application to graph simulation data. He created the logic needed to select simulation instance and graph data attributes. He created the table that displays all simulation instances.
4. **John – Web design:** John wrote CSS to make the application look presentable. He also stylized the landing page, or the page where you submit an instance of the simulation. He stylized the buttons that control the animated graph.

5. **Chubi – Frontend Developer:** Chubi made logic for the table that compares parameters of the simulation. He also created the guide on how to setup and run the simulation through Windows terminal.

6. **Nathan – Simulation Developer:** Nathan added sanity check to make sure mass is conserved. He made changes to command line interface to toggle on/off verbose logging and created the guide on how to setup and run the simulation through Linux terminal.
References


4. "ChatGPT Conversation on Data Visualizations, Population Dynamics, and Technology References." Personal conversation, 26 Mar. 2024. https://chat.openai.com/share/1a113df7-7fd7-4e97-81e6-150d0454618a

My experience in OURE felt more like an exercise in leadership than research. I’m an aggressive go-getter. When there’s a thing I want to get done, I do it, even with minimal outside guidance. I learned that most people aren’t like that. Our OURE group had 4 people. Here they are with a summary of my assessment.

1. Me – Lots of programming and problem-solving experience. I was writing code for 3 years before starting college.
2. Yug – Pretty good at programming and problem solving (though he didn’t officially do an OURE this year)
3. Chubi – Has a job outside of school, doesn’t code a lot. Can’t really solve problems.
4. John – Doesn’t code a lot.

My expectation for everyone at the start of the OURE was that everyone has a decent idea of how to do what needs to be done and then do them. For example, John wanted to rewrite the simulation program. Chubi wanted to make a database. I wanted to add more reporting features to the simulation, so more simulation data was made available in a spreadsheet. At this point, Yug wasn’t in the group yet.

Here’s how they did. John underestimated the amount of knowledge and time needed to rewrite the program. The simulation is quite complex and requires a decent knowledge of how python works. Chubi doesn’t really know how to make a database, nor did he know how the simulation works. What I learned about observing them make a goal and fail to meet them is that breaking down a problem into small, bite sized chunks, is hard. Chubi and John don’t have the programming experience needed to accurately judge what they can and cannot do. On the other hand, my task for the first semester of the OURE was just to extract simulation data. It’s a clear, concise goal that wasn’t too ambitious and doing so required me to familiarize myself with how the simulation works. I was able to trace how data was being reported by the simulation and integrate new data logging functionality. I took for granted the fact that my years of programming experience taught me exactly how to approach a problem.

The second semester of the OURE, I decided to give them more guidance. At this point, Yug joined the group. I knew the fact that everyone had trouble breaking down tasks into small pieces. So I did that part for them. I decided that we should create a web application to visualize the data because creating a web app requires a broad array of skills ranging from easy to very complex. Given the difference in programming skills, I could find ways for everyone to contribute. I gave John some CSS to do. CSS is the technology used to change how websites look. Chubi I gave a task that he was unprepared for, but I was hoping that his maturity would help him rise to the challenge. Unfortunately, he relied too much on ChatGPT which doesn’t help if you can’t code. Yug did the bulk of the work. Unlike Chubi who has a job outside of school, Yug was able to spend a lot more time doing things and figuring out. For example, I wanted Yug to figure out how to integrate data plotting with the website. He worked in it for like 8 hours. That’s time that Chubi doesn’t have.

How is research conducted in my field? Coming from a programming background, I see how research relies a lot on programming. Having worked in industry on internships, I can also see the difference in programming practice between research and industry. For example, we don’t use
containerization. It’s a technology used to distribute programs that can run anywhere. Adopting containerization seems to be institution dependent. For example, WashU uses containers. Their supercomputer requires applications to be in a docker container before running. I read some papers from WashU where the code is distributed in Singularity/Docker containers. While S&T super computers supports containers through Singularity, it’s not widely adopted in the biology department. It’s new even for our Computer Science department.

One resource that isn’t being used in my opinion is building web applications as supplemental material for research. Scientific communication is important, especially for stakeholders in government who may not understand why we are researching something. A well-developed web application makes the research interactive and publicly accessible. It also makes collaboration between researchers easier. At the research lab I worked at in WashU, the lab had a website called CIVIC where researchers can curate instances of clinically significant mutations. They were able to get funding for two full-time software developers. It’s not financially feasible for some labs, but it does show how important a web developer these days in research.