UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Brenno Lemos Melquiades

Desenvolvimento de um software científico para a modelagem e simulação computacional

São João del-Rei

2024

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Brenno Lemos Melquiades

Desenvolvimento de um software científico para a modelagem e simulação computacional

Qualificação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação do curso de mestrado do Programa de Pós Graduação em Ciência da Computação (PPGCC) da UFSJ.

Orientador: Alexandre Bittencourt Pigozzo

Universidade Federal de São João del-Rei — UFSJ

Pós-Graduação em Ciência da Computação

São João del-Rei

2024

Brenno Lemos Melquiades

Desenvolvimento de um software científico para a modelagem e simulação computacional

Qualificação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação do curso de mestrado do Programa de Pós Graduação em Ciência da Computação (PPGCC) da UFSJ.

São João del-Rei, 16 de fevereiro de 2024

Alexandre Bittencourt Pigozzo
Orientador

Rafael Sachetto Oliveira
Convidado 1

Marcelo Lobosco
Convidado 2

São João del-Rei
2024

Agradecimentos

Gostaria de agradecer e dedicar este trabalho:

Aos meus pais, pelas oportunidades que me deram, pelo apoio nas escolhas que fiz para chegar aqui, pelo companheirismo e pelo amor incondicional que me proporcionaram.

Ao meu orientador, Alexandre Pigozzo, que me introduziu ao meio científico, ao tema deste trabalho e me auxiliou não somente neste trabalho, mas na minha trajetória desta universidade.

Aos servidores do DCOMP-UFSJ, que mantém um alto nível de educação e demonstram respeito e empatia pelos docentes do curso. Em especial, um agradecimento aos servidores(as) Carolina Xavier, Leonardo Rocha, Marcos Laia, Michelli Loureiro e Douglas Dinalli.

Aos meus amigos de Guarapari, que sempre estiveram ao meu lado.

À minha namorada, Keina Takashiba, minha melhor amiga, que me ajuda a me organizar e me apoia em tudo que for.

Aos amigos que fiz aqui, na UFSJ, Arthur Gabriel Matos, Ana Clara Medina, Lucas Grandolpho e Wesley Guimarães, meus grandes companheiros de trabalhos.

À FAPEMIG, que apoiou este e os projetos anteriores.

À todos aqueles que apoiam, defendem e lutam por uma educação gratuita e de qualidade para todos.

Resumo

Palavras-chaves: modelagem, simulação, modelos computacionais, modelos matemáticos, software de interface gráfica.

As áreas de Modelagem Matemática e Computacional têm se tornado cada vez mais importantes no mundo atual, no qual estudos científicos devem trazer resultados cada vez mais rápidos. Os modelos matemáticos e computacionais surgem como ferramentas poderosas no estudo e compreensão de sistemas complexos e que podem ser usados por pesquisadores de diversas áreas diferentes. Contudo, os modelos comumente requerem extensivo conhecimento matemático para serem criados, o que resulta em uma grande barreira de entrada para cientistas sem formação relacionada diretamente com a matemática, como biólogos, e estudantes iniciando carreiras acadêmicas. Apesar de existirem softwares para esses fins, estes frequentemente apresentam interfaces muito complexas em sua busca para se tornarem genéricos o suficiente. Enquanto estes softwares têm seus casos de uso, este trabalho visa entregar uma alternativa simplificada mas funcional, voltada aos iniciantes sem comprometer o funcionamento para usuários avançados. Para isto, neste trabalho foi desenvolvido um software que facilita a construção e simulação de Equações Diferenciais Ordinárias (EDOs). EDOs são alguns dos modelos computacionais mais comuns, que conseguem representar com acurácia diversos fenômenos. O software apresenta uma interface gráfica simples e intuitiva, que possibilita o usuário exportar uma simulação, realizá-las interativamente e, inclusive, gerar o código que implementa computacionalmente o modelo.

Abstract

Key-words: modelling, simulation, automata, computational models, mathematical models, graphical interface software.

The Computational and Mathematical modelling areas have been increasingly becoming more important nowadays, in which scientific studies need to bring results faster by the day. These models serve as powerful tools in the study and comprehension of complex systems, and can be used by researchers of different areas. However, these models commonly require extensive mathematical knowledge in order to be created, which results in a big entry barrier for scientists with no mathematical-related background, like biologists, and students ingressing in the academical field. Although softwares exist for these purposes, they often bring complex interfaces in their attempt of becoming generic enough for any usage. While theses softwares certainly have their use cases, the work presented here aims to bring a simpler but functional alternative, aimed towards beginners without compromising functionality for advanced users. In order to do that, a software was developed for making it easier to construct and simulate ordinary differential equations (ODEs). ODEs are some of the most common computational models, which can represent with accuracy different phenomenons. The software presents a simple and intuitive interface, which enables the user to export a simulation, simulate interactively and even generate the code that implements the model.

Lista de ilustrações

Lista de abreviaturas e siglas

EDO	Equação Diferencial Ordinária
EDP	Equação Diferencial Parcial
RI	Representação Intermediária
GUI	*Graphical User Interface*, Interface Gráfica de Usuário

Sumário

1 Introdução

Modelos matemáticos e computacionais estão se tornando cada vez mais relevantes no mundo atual. Diversas áreas do conhecimento se beneficiam da simulação computacional e é através das simulações computacionais que fenômenos em várias áreas estão sendo compreendidos em diferentes escalas espaciais e temporais. Sem o computador não seria possível, por exemplo, simular as reações entre milhões de células.

As simulações computacionais podem auxiliar experimentos *in vitro* e *in vivo* na explicação de diversos fenômenos. As simulações permitem testar vários cenários diferentes, o que poderia ser muito custoso ou até inviável de ser testado *in vitro* e *in vivo*. Além disso, elas podem fornecer previsões testando hipóteses que ainda não foram investigadas.

O desenvolvimento de modelos computacionais requer um conjunto de etapas importantes que começam com a definição de seu objetivo e terminam com sua validação, passando por múltiplas iterações e melhorias. Dentre estas etapas, uma das mais desafiadoras é a implementação do modelo. Essa etapa requer o conhecimento de métodos numéricos, programação, estruturas de dados, bibliotecas, entre outros recursos. Um erro na implementação pode comprometer todo o trabalho. Visando a simplificação desta etapa, neste trabalho é apresentado um software que foi desenvolvido para auxiliar pesquisadores na implementação de alguns tipos de modelos computacionais.

O principal objetivo deste trabalho é desenvolver um software para facilitar a criação e simulação de modelos computacionais, dimiuindo a barreira de entrada na área de Modelagem Computacional e permitindo que seja necessário investir menos tempo na implementação do modelo. A partir do "desenho" de um modelo feito na interface gráfica, o software é capaz de gerar o código que implementa o modelo, simulá-lo e até exportar gráficos.

Entre as contribuições deste trabalho, destacam-se as seguintes:

- A criação de uma representação gráfica intuitiva para modelos matemáticos baseados em Equações Diferenciais;
- O desenvolvimento de um gerador de código baseado em *templates* para gerar os códigos que implementam os modelos;
- A construção de modelos de Equações Diferenciais Ordinárias (EDOs) sem a necessidade de iniciar a implementação computacional dos modelos do zero;

Uma das aplicações do software que foi desenvolvido é no processo de ensino-aprendizagem de modelagem computacional. Os usuários poderão aprender sobre mode-

lagem computacional por meio de um processo de aprendizado ativo com a construção e simulação dos modelos na prática.

O restante do texto deste trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados alguns conceitos teóricos para um melhor entendimento do trabalho. No Capítulo 3, são descritos os trabalhos relacionados. O software desenvolvido é descrito no Capítulo 4. No Capítulo 5 são apresentados e discutidos os resultados e, por fim, a conclusão e os trabalhos futuros são descritos no Capítulo 6.

2 Referencial Teórico

Neste capítulo, são apresentados os principais conceitos que formam a base teórica do trabalho.

2.1 Equações Diferenciais Ordinárias

Uma Equação Diferencial Ordinária (EDO) é um tipo de modelo matemático que descreve como as populações modeladas evoluem ao longo do tempo. Os modelos de EDOs são bastante difundidos e utilizados em diversas áreas como, por exemplo, na medicina e na neurociência (ADOMIAN, 1995), no estudo do Câncer (SPENCER et al., 2004; TALKINGTON; DANTOIN; DURRETT, 2018), no estudo da resposta imune à infecções virais (REIS et al., 2021), no estudo da eletrofisiologia cardíaca (VIGMOND et al., 2008; BUCELLI et al., 2022), entre outros exemplos.

A seguir, é apresentado um exemplo de modelo clássico da literatura.

O modelo predador-presa, também conhecido como modelo Lotka-Volterra, modelo a dinâmica da interação entre uma presa e um predador. O modelo é descrito pelas seguintes EDOs:

$$\begin{aligned} \frac{dH}{dt} &= r.H - a.H.P \\ \frac{dP}{dt} &= b.H.P - m.P \end{aligned} \tag{2.1}$$

As populações do modelo e as taxas são:

H	Presa
P	Predador
r	Taxa de reprodução da presa
m	Taxa de mortalidade dos predadores
a	Taxa de predação
b	Taxa de reprodução dos predadores

Neste modelo, temos os seguintes processos sendo modelados:

- Reprodução das presas (termo $r.H$);
- Predação (termo $a.H.P$);
- Reprodução dos predadores (termo $b.H.P$);

- Morte dos predadores (termo $m.P$).

Termos de replicação, predação e morte como os vistos neste modelo são muitos comuns e são empregados na maioria dos modelos. Esses termos são construídos com base no princípio da Lei de Ação de Massas (*Mass Action Law*) que diz que "O número de interações entre duas partículas depende da concentração de ambas". A Lei de Ação de Massas pode ser aplicada, a princípio, a qualquer sistema de qualquer área onde as seguintes hipóteses são consideradas:

- O sistema é bem homogêneo (*well-mixed system*);
- As partículas/substâncias se movimentam de forma aleatória;
- Não é considerada nenhuma estrutura espacial (*lack of spatial structure*) no modelo.

Na Figura 1, é dado um exemplo de resultado obtido com a simulação do modelo predador-presa.

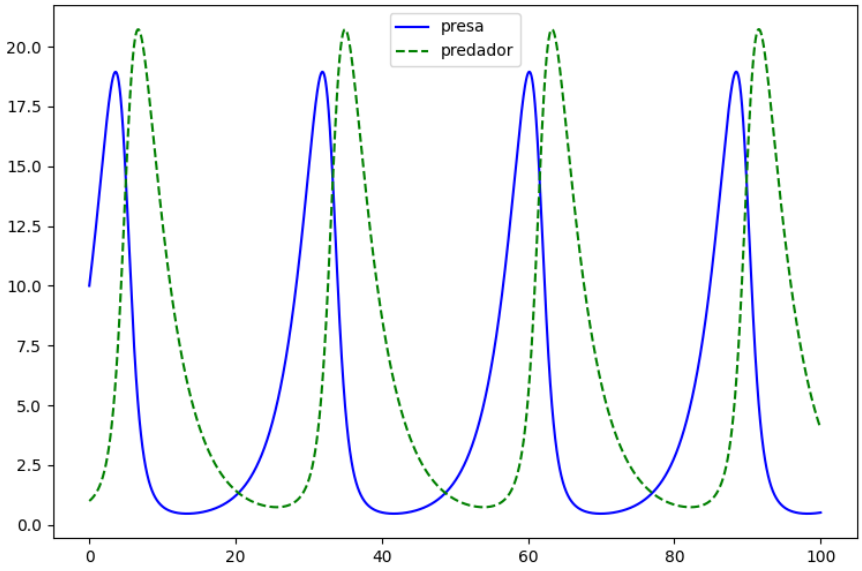

Figura 1 – Exemplo de resultado do modelo Predador-Presa clássico.

2.2 Programação Visual e Editores baseados em Nós

Programação visual é uma maneira do usuário programar a máquina por meio de elementos gráficos que abstraem instruções do computador. Comumente, estes elementos representam múltiplas operações por vez, com o objetivo de facilitar a programação.

Um dos primeiros exemplos da aplicação prática da programação visual foi com a linguagem GRAIL (*GRAphical Input Language*) (ELLIS; HEAFNER; SIBLEY, 1969), desenvolvida em 1969 junto a um dispositivo semelhante a uma caneta e uma tela sensível ao seu toque. O conjunto do *hardware* e do *software* permitia que o usuário desenhasse letras e formas geométricas que se traduziriam em elementos de fluxograma. O objetivo era realizar um estudo sobre a comunicação humano-computador.

O conceito de utilizar elementos visuais para a assistência da programação não se restringiu somente a dispositivos especializados, contudo, e hoje se encontra em diversos *softwares* de finalidades distintas que podem ser instalados em qualquer computador. Como um exemplo de aplicação de propósito similar ao GRAIL, a linguagem Scratch (MALONEY et al., 2010) foi criada com o objetivo de facilitar o ensino de lógica de programação para crianças sem comprometer em funcionalidades. A linguagem apresenta comandos, funções e estruturas de controle como blocos que podem ser "montados" para compor um programa.

Outras aplicações se especializam em algum nicho da computação, como é o caso do simulador lógico Logisim (BURCH, 2002), que abstrai componentes lógicos e permite o usuário criar circuitos em uma espécie de *protoboard* digital.

Muitas vezes, *softwares* que incluem programação visual para a abstração de operações utilizam uma interface conhecida como editor de nós:

- O *software* de modelagem, animação e edição de vídeos, Blender (TAKALA; MäKäRäINEN; HAMALAINEN, 2013), utiliza um editor de nós para abstrair operações de manipulação de imagens e materiais;
- O motor de jogos proprietário Unreal Engine[1] utiliza editores de nós para múltiplas finalidades e, entre elas, está a escrita de *shaders* e a programação de lógica geral dos jogos.

Existem diversos motivos pelos os quais a programação visual se encontra em tão amplo uso. Principalmente, pode-se atribuir o sucesso de sua aplicação à facilidade de utilização. Em geral, não se faz necessário a leitura de manuais da mesma forma que se faz com linguagens de programação tradicionais; todas as operações possíveis no software estão modeladas como elementos visuais com parâmetros de entrada e saída bem definidos. Implementações exemplares também proibirão o usuário de realizar ligações inválidas diretamente na interface, realizando uma etapa mais comum em compiladores ou analizadores estáticos de código, que geralmente permitem que o usuário cometa um ou mais erros e só descubra-os posteriormente.

[1] Documentação da Unreal Engine

2.3 Geração de código

A fase de geração de código geralmente é a última fase da compilação. Na maioria dos compiladores, o gerador de código recebe como entrada uma ou mais representações intermediárias do código/texto de entrada e retorna como saída o código/texto na linguagem alvo (AHO et al., 2006).

Como este trabalho tem o objetivo de possibilitar a geração de código em múltiplas linguagens alvo diferentes, surgiu a necessidade de abstrair mais esta fase, de forma que a implementação de linguagens alvo adicionais seja mais simples dado que ao menos uma já foi implementada.

Para tal abstração, seguiu-se um padrão utilizado por diversos compiladores diferentes, que consiste na separação do *back-end* (que trabalha com a linguagem alvo) do *front-end* (que trabalha com a linguagem fonte) através da utilização de uma ou mais representações intermediárias.

Nas próximas subseções, serão apresentados alguns conceitos importantes para o entendimento do processo de geração de código.

2.3.1 Representação intermediária

As representações intermediárias (RIs) são usadas em um compilador por várias razões (AHO et al., 2006), dentre as quais destaca-se:

- Separar o *front-end* do *back-end*. Neste caso, o *front-end* não precisa se preocupar com detalhes da linguagem alvo e o *back-end* não precisa conhecer detalhes da linguagem fonte;
- Permitir que sejam realizadas otimizações independente de máquina ou otimizações independente da linguagem alvo;
- Para facilitar a tradução e geração do código alvo.

Algumas RIs usadas em compiladores incluem:

- Árvore de Sintaxe Abstrata (ASA);
- Grafos Acíclicos Dirigidos (GAD);
- Código de Três Endereços.

As Árvores de Sintaxe Abstratas são muito usadas na representação do código fonte de entrada, representando comandos e expressões em cada escopo. Os Grafos Acíclicos Dirigidos são muito usados na fase de otimização do código. O Código de Três Endereços

é uma representação que está próxima da linguagem Assembly e pode ser usada como base para várias otimizações independentes da máquina. Geralmente, o Código de Três Endereços otimizado é a entrada recebida pelo gerador de código Assembly. O Código de Três Endereços também pode ser utilizado pelo algoritmo de alocação de registradores como uma abstração inicial para a escolha e atribuição de registradores.

Como uma alternativa mais abstrata ao Assembly (que é dependente da arquitetura alvo), existe a linguagem LLVM IR (LLVM *Intermediate Representation*), que por sua vez pode ser transpilada para o Assembly da máquina alvo. A LLVM é uma infraestrutura para o desenvolvimento de compiladores com o objetivo de facilitar sua criação sem se preocupar com a arquitetura de computadores. Considerando que uma linguagem possa ser transpilada na representação intermediária da LLVM (LLVM IR), ela pode ser compilada em qualquer arquitetura suportada pela infraestrutura que incluem, mas não estão limitadas a, x86, AMD64, ARM, ARM64, WebAssembly e RISC-V.

Para facilitar o processo de geração de código e realizar uma maior separação entre o *front-end* (interface gráfica) e o *back-end* (gerador de código) do software, foi criada e utilizada uma representação intermediária (RI). A RI é a entrada para o gerador de código e também é utilizada para salvar e carregar os modelos. A RI desenvolvida será apresentada e explicada na Seção 4.1.1.

2.3.2 Sistemas baseados em *templates*

Em várias aplicações, a geração de código é feita com base em *templates*. Desde o início da World Wide Web, *templates* têm sido usados por *frameworks* para ajudar no processo de construção de páginas Web. Informalmente, podemos dizer que um *template* é um esqueleto (uma estrutura) que serve de referência para todo o processo de geração de código.

Em sua essência, *templates* são arquivos com marcadores especiais que podem ser substituídos por outros valores dinamicamente, de forma similar ao pré-processador de C. Adicionalmente, alguns motores de *templates* introduzem estruturas de controle de fluxo, como condicionais, laços de repetição e chamadas de funções.

Para ilustrar a ideia, suponha o *template* abaixo criado com base em um código que implementa um sistema de Equações Diferenciais Ordinárias (EDOs):

```
1 from scipy.integrate import solve_ivp
2 import numpy as np
3 import pandas as pd
4
5 def odeSystem(t, P, {% for key, value in params %}{%if loop.is_last%} {{
    key}} {%else%} {{key}}, {%endif%}{%endfor%}):
6 ## for key, value in vars
7     {{key}} = P[{{loop.index}}]
```

```
## endfor
## for key, value in odes
    d{{key}}_dt = {{value}}
## endfor
    return  [{% for key, value in odes %}{%if loop.is_last%} d{{key}}_dt
    {%else%} d{{key}}_dt, {%endif%}{%endfor%}]
```

O *template* acima utiliza os marcadores definidos pela biblioteca Inja[2], e demonstra algumas das estruturas de controle mencionadas, na forma das palavras-chave `for`, `if` e `else`.

2.4 Ensino-aprendizagem de Modelagem Computacional

Os modelos são abstrações do mundo real. Com a evolução dos computadores e o seu uso massivo, os modelos computacionais têm contribuído significativamente para o avanço do conhecimento em diversas áreas. Eles nos ajudam a compreender os fenômenos de interesse permitindo "rastrear" e entender as mudanças que estão ocorrendo em um sistema complexo e visualizar, por exemplo, o efeito de uma pequena mudança no restante do sistema.

Simulações computacionais são ferramentas fundamentais não só para a pesquisa científica, mas também para a educação. Elas são frequentemente usadas como laboratórios virtuais para promover a compreensão dos alunos sobre os conceitos teóricos que estão na base dos sistemas simulados. [Imagining the School of the Future Through Computational Simulations: Scenarios' Sustainability and Agency as Keywords].

Após a Segunda Guerra Mundial, o leque de campos disciplinares em que as simulações começaram a ser utilizadas tem vindo a expandir-se até os dias de hoje, de modo que é quase impossível nomear qualquer disciplina que não tenha utilizado ou desenvolvido ferramentas computacionais e simulações para avançar a fronteira do conhecimento (Borrelli e Wellmann, 2019).

O uso de modelos computacionais abre novas maneiras para os alunos aprenderem sobre diferentes disciplinas. Os alunos podem aprender através de um processo iterativo de construção de modelo, simulação, modificação de modelo para simulação novamente, explorando todas as características do modelo em si [Imagining the School of the Future Through Computational Simulations: Scenarios' Sustainability and Agency as Keywords]. Plataformas como, por exemplo, a Netlogo [ref] permitem este tipo de aprendizagem iterativa e interativa.

Um dos objetivos do software que foi desenvolvido neste trabalho é auxiliar o aprendizado de conceitos importantes nas áreas de modelagem matemática e computaci-

[2] https://github.com/pantor/inja

onal e ajudar o aluno a desenvolver a habilidade de modelar problemas através da prática de construção e simulação de modelos. O uso combinado do software com alguma metodologia ativa de ensino permitirá aos alunos aprender na prática como construir modelos para diferentes fenômenos de interesse e entender melhor os fenômenos estudados através das simulações computacionais realizadas de forma interativa através da interface gráfica do software.

Considerando o uso do software na prática e uma visão geral do chamado ciclo da modelagem apresentado na Figura ..., destaca-se que o software auxilia o passo 2 e automatiza os passos 3 e 4 facilitando o processo de modelagem por estudantes e pesquisadores.

1) Estudo do problema. 2) Formulação do modelo. 3) Implementação do modelo. 4) Simulação computacional. 5) Validação do modelo.

O aprendizado, em sala de aula, pode ser conduzido, por exemplo, através de um processo incremental que começa investigando e trabalhando com modelos mais simples até se chegar em modelos mais complexos. Algumas vantagens incluem: 1) a apresentação de determinados conceitos e técnicas é facilitada em modelos mais simples; 2) em muitos casos reais, modelos complexos utilizam certas expressões, equações e "estratégias" na modelagem que estão presentes também em modelos mais simples. Então, muitas vezes, o aluno conseguirá entender melhor um modelo mais complexo entendendo que determinadas partes do mesmo já apareceram em modelos mais simples previamente estudados.

Uma possível abordagem de ensino seria o professor trabalhar com um problema para apresentar e discutir alguns conceitos e/ou técnicas de modelagem de forma semelhante à metodologia Problem Based Learning (PBL) [refs]. O seguinte conjunto de passos é sugerido a seguir:

1. Descrição do contexto e do problema a ser modelado;
2. Formulação das hipóteses pelos alunos e discussão entre eles;
3. Formulação do modelo considerando o conhecimento de modelos existentes e que foram estudados previamente;
4. Construção do modelo no software;
5. Simulação do modelo no software com a criação de diferentes cenários para teste e análise dos comportamentos obtidos.
6. Caso o comportamento desejado ou esperado não seja obtido, voltar para o passo 2.

As metodologias ativas de ensino colocam o aluno como protagonista no processo de construção do próprio conhecimento. ...

3 Trabalhos relacionados

Com o avanço de tecnologias para o desenvolvimento de aplicações gráficas e também da internet, ocorreu um aumento expressivo na quantidade de *softwares* desenvolvidos para modelagem e simulação computacional. Entrentanto, poucos destes *softwares* permitem que o usuário visualize e interaja com o código gerado para a simulação de seu modelo. Mais comumente, a única maneira com a qual o usuário pode interagir com o modelo é através da abstração provida pela interface, o que causa uma dependência direta do *software*.

A seguir, na Tabela ... é apresentado um quadro comparativo entre o software deste trabalho e softwares similares. Para realizar a comparação, foram selecionadas algumas características consideradas relevantes no contexto de softwares para modelagem e simulação computacional.

O software desenvolvido possui alguns diferencias em relação a outros desenvolvidos com propósitos similares:

- Em sua interface gráfica é apresentado um editor baseado em nós que fornece ao usuário uma forma de programação visual 2.2, possibilitando um grau de liberdade limitando apenas operações consideradas incorretas;
- Em seu formato de representação intermediária para salvamento e carregamento dos modelos, o software utiliza uma estrutura extensível para outros tipos de modelos como, por exemplo, equações diferenciais parciais. Esta estrutura também foi projetada para ser humanamente legível, a ponto de ser possível que uma pessoa decodifique o modelo representado apenas lendo o arquivo. Assim, caso o software se prove insuficiente ou indisponível, ainda é possível reverter o arquivo nas equações que o compõe.

A seguir, são descritos em mais detalhes os trabalhos relacionados.

3.1 Snoopy

Um *software* que serviu como inspiração para este trabalho foi o software para construção, animação e simulação de redes de Petri chamado Snoopy (HEINER; RICHTER; SCHWARICK, 2008; HEINER et al., 2012; LIU, 2012). Além da rede de Petri clássica, o software permite modelar e simular vários tipos de redes de Petri como, por exemplo, redes de Petri estocásticas e coloridas que são extensões da rede clássica. O modelo construído é representado como um grafo que tem dois tipos de nós chamados *places*

(locais) e *transitions* (transições) e quatro tipos de arestas (estocástica, imediata, determinística e planejada) com suas semânticas associadas. Os *places* representam populações de um certo tipo e transições representam eventos que podem aumentar ou diminuir as quantidades das populações que estão armazenadas nos locais.

O modelo gráfico utilizado pelo software Snoopy serviu de inspiração para a criação da representação utilizada neste trabalho. A partir do software Snoopy, foi possível observar que modelos gráficos, além do atrativo visual, facilitam a construção de um modelo e um melhor entendimento do que está sendo feito, o que está sendo modelado. A possibilidade de simular interativamente as Redes de Petri estocásticas foi um fato que serviu de inspiração para a ideia da simulação interativa no software desenvolvido neste trabalho.

3.2 Insight maker

O InsightMaker (FORTMANN-ROE, 2014) é um *software* de simulação baseado em nuvem. Todo o modelo é feito através de um navegador de internet e, quando executado, é simulado no próprio servidor, sem a necessidade de que o usuário possua uma máquina potente. O InsightMaker (FORTMANN-ROE, 2014) permite a construção de modelos utilizando diagramas de Dinâmica de Sistemas e modelos baseados em agentes. O site possui diversas facilidades para a construção do modelo em um Canvas. A partir do modelo de Dinâmica de Sistemas construído no Canvas, o InsightMaker (FORTMANN-ROE, 2014) gera internamente um modelo de EDOs e executa este modelo utilizando o método de Euler ou o método de Runge-Kutta de quarta ordem conforme escolha do usuário. Através da interface web, também é possível definir os valores de todas as variáveis e parâmetros, e é possível realizar as simulações computacionais. Os resultados são exibidos de forma interativa onde é possível controlar a velocidade com que os valores são exibidos nos gráficos.

3.3 VCell

O software VCell (Virtual Cell) (MORARU et al., 2008; SCHAFF et al., 1997) é uma plataforma para modelar sistemas biológicos celulares que é construída em torno de um banco de dados central e disseminada como um aplicativo da web. Os modelos são construídos com base em regras (*Rule-based modelling*). VCell tem os seguintes tipos de simulações disponíveis: determinística (EDO compartimental ou EDP de reação-difusão-advecção com suporte para cinemática 2D), reações estocásticas (solvers de SSA), estocástica espacial (reação-difusão com Smoldyn), híbrida determinística/estocástica e simulações baseadas em agentes. O software permite simular vários processos metabólicos que ocorre dentro das células e através das membranas celulares. Com relação às mem-

branas, há suporte para os seguintes processos: Eletrofisiologia, suporte para o fluxo da membrana e difusão lateral da membrana.

3.4 EpiFire

O EpiFire (HLADISH et al., 2012) não é um *software* completo, mas uma biblioteca em C++ para simulação de modelos de redes complexas com aplicação na área de Epidemiologia. Ele é uma interface de programação de aplicativos (API) implementado em C++, projetado para gerar de forma eficiente redes com uma distribuição de grau especificada, medir características fundamentais da rede e realizar simulações computacionais eficientes das redes geradas com base nos métodos *percolation* e *chain-binomial*.

3.5 Cytoscape

O software Cytoscape (SHANNON et al., 2003) é uma plataforma de software de código aberto para visualizar redes de interação molecular e vias biológicas e integrar essas redes com anotações, perfis de expressão gênica e outros dados. Atualmente, o Cytoscape é utilizado como uma plataforma geral para análise e visualização de redes complexas.

4 Software para modelagem e simulação computacional

O *software* desenvolvido neste trabalho consiste em uma interface gráfica de usuário [2] (mais comumente referida pelo acrônimo em inglês, GUI, que significa *Graphical User Interface*) que apresenta um editor baseado em nós. Os nós representam abstrações de componentes comumente usados na construção de EDOs, como constantes e variáveis. Em 4.1, serão discutidos os tipos de nós presentes na interface.

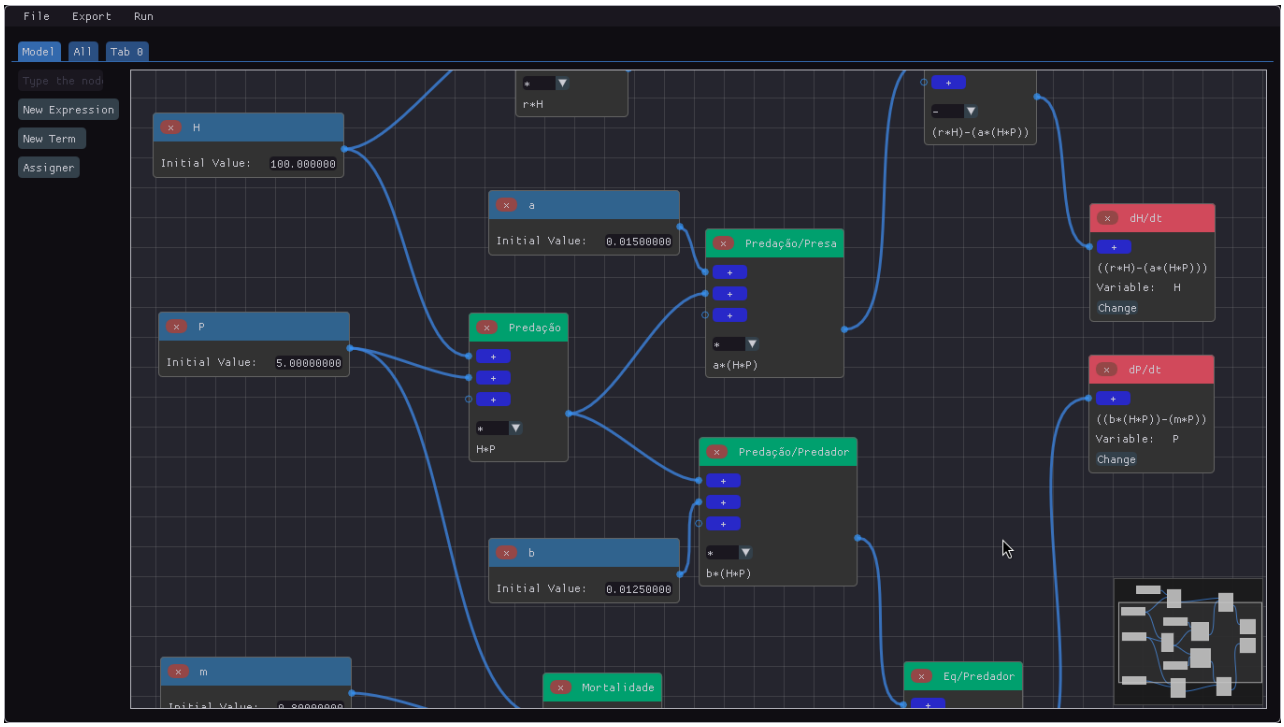

Figura 2 – Exemplo do modelo Predador-Presa (2.1) construído no *software*.

O modelo é construído a partir da criação de nós e suas ligações. Uma equação pode ser expressada por um ou mais termos e combinações, e uma variável alvo para a atribuição do valor.

Após construído o modelo, o usuário poderá simulá-lo diretamente pela GUI [3], exportar um PDF dos resultados, ou exportar um código de Python equivalente para o modelo desenhado [.1]. A experiência de usuário pode ser resumida ao fluxograma 4.

Dessa forma, o usuário pode criar e simular modelos sem a necessidade de iniciar a implementação computacional do zero e também sem a necessidade de ter conhecimentos dos métodos que são utilizados na implementação. Pelo contrário, o usuário terá acesso aos códigos gerados e poderá aprender com estes e utilizá-los, por exemplo, como base para desenvolver o seu próprio código.

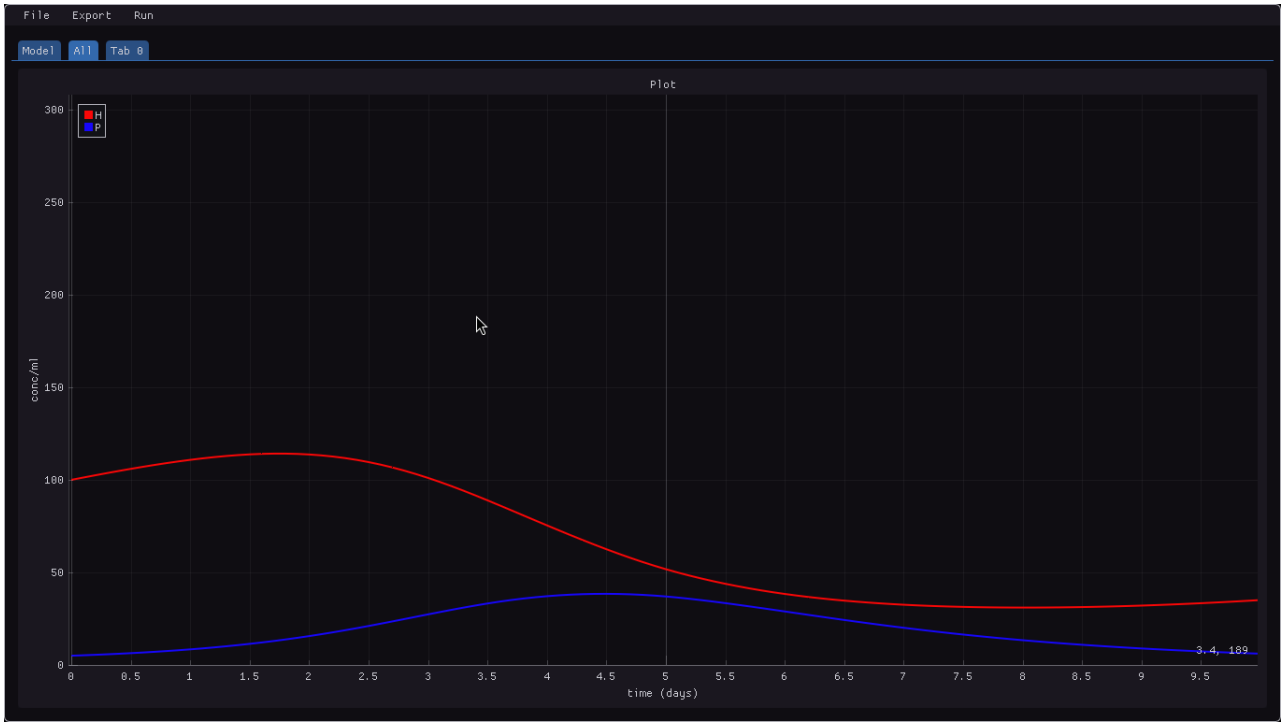

Figura 3 – Plotagem dos resultados da simulação do modelo Predador-Presa.

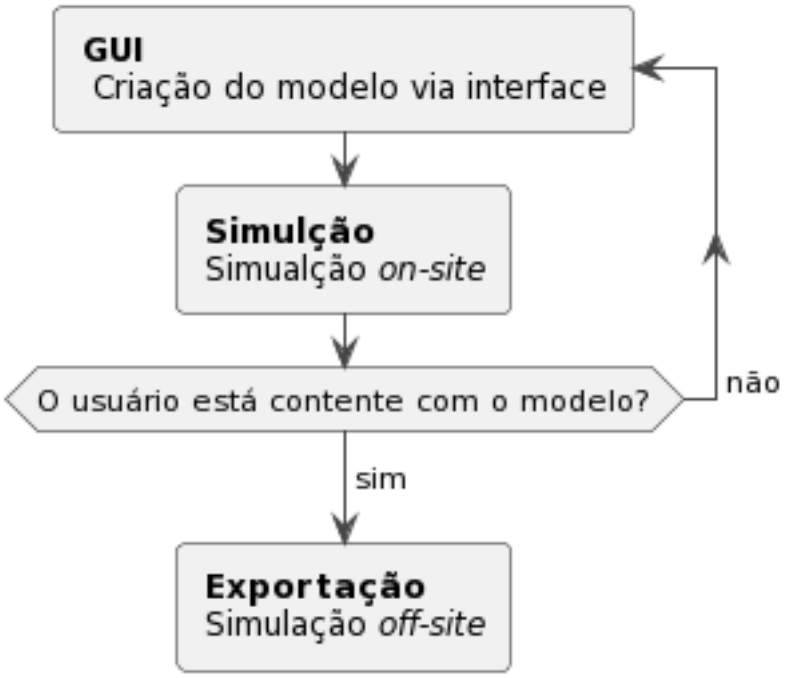

Figura 4 – Fluxograma da experiência de usuário esperada para o software.

A simulação das EDOs, tanto pela GUI quanto pelo código de Python exportado, é feita utilizando métodos especializados da biblioteca científica `scipy` (VIRTANEN et al., 2020).

Tipo de nó	Usado para representar	Exemplo
Termo	Variáveis, parâmetros e constantes.	Em uma expressão $k.x.y$, os termos são o parâmetro k e as variáveis x e y.
Expressão	Qualquer expressão matemática para a construção das equações do modelo.	Na equação da presa [2.1], as expressões são: 1) $r.H$; 2) $a.H.P$; e 3) $r.H - a.H.P$ (combinação de 1 e 2).
Equação	É um nó especial usado para atribuir uma expressão, que representa todo o lado direito de uma EDO, a uma variável.	A expressão $r.H - a.H.P$ pode ser usada como entrada de um nó de equação associado à variável H.

Tabela 1 – Tipos de nós presentes no software.

4.1 Uma representação visual para EDOs

Uma representação visual foi desenvolvida para minimizar a necessidade de um conhecimento técnico prévio pelo usuário, assim como a dependência de documentos e manuais. Para atingir este objetivo, foi necessário modelar esta representação utilizando conceitos que fossem naturais para o público-alvo e a forma com a qual este desenvolve EDOs.

Entende-se que EDOs são construídas seguindo padrões. Por exemplo, dado duas populações A e B, caso seja desejado simular uma condição que seja mais provável de acontecer dado altas concentrações de ambas as populações, é possível representá-la simplesmente multiplicando ambas as populações. Opcionalmente, adiciona-se uma constante para controlar a taxa de ocorrência.

No sistema de EDOs 2.1, é possível visualizar este fenômeno: na equação da presa, a expressão $-a.H.P$ é utilizada para controlar a taxa de predação, isto é, o quanto a concentração desta população é afetada negativamente pela atividade de predação realizada pelos predadores. Naturalmente, a predação ocorrerá mais frequentemente dado altas concentrações de ambos presa e predador, assim como descrito acima.

A partir desta premissa, pode-se resumir o desenvolvimento de EDOs à combinação das expressões que representam as interações das populações entre si e o meio. Assim, a representação visual foi modelada em um editor de nós, de forma que todas as interações visassem a construção destas expressões. a tabela 1 descreve os tipos de nós presentes no software e sua relação com o objetivo de construir as expressões.

4.1.1 Uma representação intermediária humanamente legível

Neste trabalho, a representação intermediária (RI) foi utilizada com dois propósitos: (1) Separar o *front-end* (interface gráfica) do *back-end* (gerador de código) e (2) Permitir que os modelos construídos sejam salvos e carregados da unidade de armazena-

mento.

A RI evita que mudanças realizadas no código da interface gráfica tenham um grande impacto na implementação do gerador de código. Desta forma, o gerador de código somente será impactado por alguma mudança na *GUI* quando a mudança tiver como efeito a inserção, modificação ou remoção de informações da representação intermediária.

A alternativa a esta abordagem seria passar as informações em memória do modelo diretamente para o gerador de código. Neste caso, o acoplamento dos módulos seria maior, e o impacto disto seria maior tempo de desenvolvimento para cada nova transformação adicionada (por exemplo, novas linguagens de programação).

Para permitir que o usuário salve e carregue os modelos, a RI assumida se baseia num formato JSON que representa de forma direta as classes de C++ que formam a GUI. Para a serialização e deserialização em JSON, utilizou-se uma biblioteca da linguagem Rust chamada *serde* (que será explicada em detalhes na Seção 4.3), que permite que esta conversão seja feita de forma declarativa. Na listagem a seguir, é mostrado um exemplo de arquivo JSON que é referente ao modelo Predador-Presa (Figura 2). As EDOs desse modelo foram apresentadas na Equação 2.1.

```
{
    "meta_data": {
        "start_time": 0,
        "end_time": 10.50,
        "delta_time": 0.1
    },
    "nodes": {
        "1": {
            "id": 1,
            "name": "Population 1",
            "related_constant_name": "Population 1_0",
            "links": [
                {
                    "sign": "+",
                    "node_id": 30
                }
            ],
        },
        "2": {
            "id": 2,
            "name": "Population 2",
            "related_constant_name": "Population 2_0"
```

```
23          },
24          "30": {
25              "id": 30,
26              "name": "Pop1 + Pop2",
27              "operation": "+",
28              "inputs": [1, 2]
29          }
30      },
31      "constants": [
32          {
33              "name": "gravity",
34              "value": 9.81
35          },
36          {
37              "name": "Population 1_0",
38              "value": 100
39          },
40          {
41              "name": "Population 2_0",
42              "value": 100
43          },
44          {
45              "name": "a",
46              "value": 1.6
47          }
48      ]
49  }
```

Neste exemplo de RI, pode-se observar 3 seções:

- `"meta_data"`: Esta seção inclui os meta dados da simulação que não fazem parte de nenhuma construção específica. Entre estes dados, estão o tempo inicial da simulação (`"start_time"`), tempo final (`"end_time"`) e o passo de tempo (`"delta_time"`);
- `"nodes"`: Esta seção inclui os nós como vistos interface gráfica. Estes nós podem ser do tipo "população" ou "combinação".

  No primeiro caso, o nó define as relações que ele possui com outros nós pelo atributo `"links"`. As ligações possuem os dados de sinal (se a relação é positiva (+) ou negativa (-) para a população) e o identificador do nó combinatório relacionado.

No segundo caso, o nó define o atributo `"inputs"`, que é uma lista dos identificadores dos nós que se ligam a este. Diferentemente do sub-atributo `"node_id"` do nó anterior, estes identificadores podem ser de qualquer tipo de nó.

A partir desta estrutura, é possível montar as equações de qualquer população através de uma resolução recursiva de identificadores. Em 4.1.1 é apresentado um *template* na linguagem Jinja2[1] capaz de realizar a construção das equações como descrito;

- `"constants"`: Esta seção inclui as constantes que podem ser utilizadas em equações, com seus nomes e valores.

```
Equations:
{% for node_id, node in model.nodes|items -%}
 {% if node.related_constant_name %}
  - d{{ node.name }}/dt =
  {%- for link in node.links -%}
   {%- set link_node = model.nodes[link.node_id] -%}
   {%- if link_node.related_constant_name -%}
    ({{ link.sign }} {{ link_node.name }}) {% if not loop.
   last %} + {% endif %}
   {%- else -%}
    {{ link.sign }} (
     {%- for input in link_node.inputs recursive -%}
      {%- set inner_link_node = model.nodes[input] %}
      {%- if inner_link_node.related_constant_name -%}
       {{ inner_link_node.name }} {% if not loop.last %} {{
   link_node.operation }} {% endif %}
      {%- else -%}
       {%- set old_link_node = link_node -%}
       {%- set link_node = inner_link_node -%}
       {{ loop(inner_link_node.inputs) }}
       {%- set link_node = old_link_node -%}
       {% if not loop.last %} {{ link_node.operation }} {%
   endif %}
      {%- endif -%}
     {%- endfor -%}
    ) +
   {%- endif -%}
```

[1] Jinja2 Homepage

```
25    {%  endfor  -%}
26   {%  endif  %}
27 {%-  endfor  -%}
```

4.2 Geração de Código e Simulação interativa

Além de apresentar uma interface capaz de criar, salvar e carregar sistemas de EDOs, o software também possibilita o usuário de simular as equações (plotando na interface), gerar o código equivalente em Python ou exportar um PDF com os resultados.

A geração de código é feita por meio da interpolação e substituição de *strings* em *templates*. Estes *templates* incluem o código base necessário para a simulação e plotagem com as bibliotecas corretas inclusas, ecrito de maneira sucinta e legível com o objetivo de servir como um material de aprendizagem estensível.

Este código gerado, além de ser aprensentado como opção de exportação do modelo para o usuário, também é utilizado internamente pelo software para produzir os resultados que são plotados na interface. Como benefício desta abordagem, os gráficos plotados tanto pelo software quanto pelo código exportado são idênticos. Por outro lado, a dependência de Python para a simulação traz complicações quanto à distribuição do software, que será discutida em mais detalhes em 4.4 assim como será discutida a solução implementada.

4.3 Implementação e Tecnologias utilizadas

O software principal que contém a interface gráfica está sendo construído na linguagem C++ utilizando, principalmente **OpenGL**[2] e **ImGui**[3].

O contexto de OpenGL é gerenciado pela biblioteca multi-plataforma **GLFW**[4]. A interface foi desenvolvida com ImGui, uma biblioteca de gráficos de modo imediato. Ao contrário da maioria dos *frameworks* de interface gráficas que utilizam pesadamente de orientações a objeto e vários níveis de heranças, a biblioteca ImGui fornece uma interface imperativa, na qual os componentes são definidos (em sua maioria) por funções e a ordem de aparição na tela é definida pela ordem de chamada dessas funções.

Para o editor de nós, a extensão **ImNodes**[5] foi utilizada. Ela apresenta uma interface que segue a mesma filosofia da ImGui, permitindo a definição de nós e pinos de forma imperativa, facilitando a manipulação da interface.

[2] https://www.opengl.org/
[3] https://github.com/ocornut/imgui
[4] https://www.glfw.org/
[5] https://github.com/Nelarius/imnodes

Foi feito o uso de classes e subclasses para definir os nós e pinos utilizados na interface. Nessa estrutura, os nós são donos dos pinos de saída e entrada, e as passagens de informações são feitas ou entre nós e seus pinos, ou entre pinos de diferentes nós, como na Figura 5.

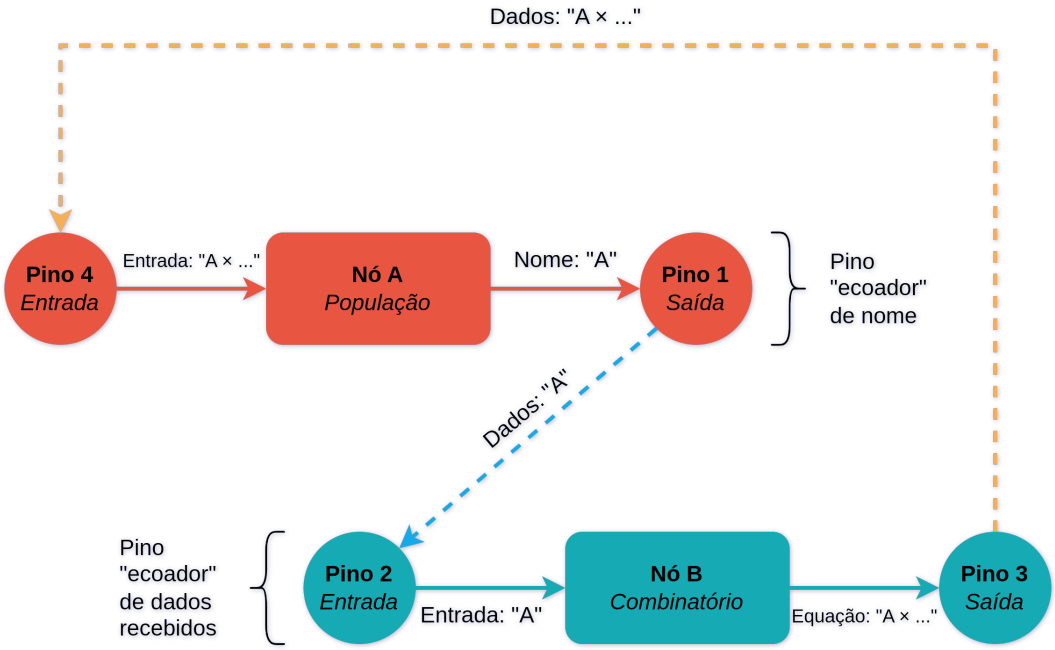

Figura 5 – Fluxo da passagem de dados entre nós e pinos na interface gráfica.

A Figura 5 ilustra como o nome da população "A" é levada até um nó combinatório que monta a expressão "A * ..." (assumindo nós omitidos) e retorna-o para a população, por meio de conexões entre pinos. É importante que as populações recebam as expressões montadas pelos combinadores para que o usuário consiga facilmente observar as equações completas a qualquer momento.

4.3.1 A biblioteca de RI

Para realizar todas as conversões mencionadas ao longo deste texto, foi desenvolvida uma biblioteca em Rust. A linguagem Rust foi escolhida por trazer maior segurança, confiabilidade e facilidade de desenvolvimento. Como Rust é uma linguagem compilada, é possível compilar bibliotecas estáticas ou dinâmicas para utilizar em C ou C++. No caso deste software, a biblioteca expõe algumas `structs` e funções utilizando a *FFI* (acrônimo para Interface de Funções Estrangeiras, em inglês) de C, e os arquivos *headers* são gerados automaticamente pelo programa **cbindgen**[6] e **GitHub Actions**[7].

[6] https://github.com/eqrion/cbindgen
[7] cbindgen-action

O software utiliza a biblioteca **serde**[8] para declarativamente criar as conversões das `structs` de/para o formato JSON por meio do macro *derive* de Rust. Este macro permite que estruturas implementem certos métodos automaticamente, puramente pela leitura dos campos que esta possui. Este é o caso das `structs` `Model`, `MetaData`, `Node` e `Constant`.

Uma outra vantagem da linguagem Rust é possuir um ecossistema de bibliotecas com pequenos escopos e bem integradas. Neste caso, o motor de *templates* utilizado, **minijinja**[9], recebe como entrada para suas conversões estruturas que implementam métodos de serialização e desserialização da biblioteca serde.

Pelos motivos mencionados acima, a implementação das conversões de/para JSON e para o modelo de EDOs foi tão simples quanto declarar as estruturas e escrever o *template* de EDOs, apresentado em 4.1.1.

4.3.2 Testes unitários

O software atualmente também implementa testes unitários em C++ utilizando o *framework* de testes **Catch2**[10]. O objetivo dos testes é garantir que a biblioteca em Rust é compilável e possui estruturas que passam de forma segura pela FFI de C. Adicionalmente, os testes realizam garantias como checagem de campos em JSON antes e pós conversão para estruturas.

Estes testes tornam a vida do desenvolvedor mais fácil, uma vez que garantem que alterações no código não quebram funcionalidades pré-existentes. Os testes possuem execução automatizada por meio de **GitHub Actions**[11], que fornecem máquinas em nuvem sob demanda, assim impedindo que o desenvolvedor se descuide e esqueça de executá-los.

4.4 Distribuição do software

[8] https://serde.rs/
[9] https://github.com/mitsuhiko/minijinja
[10] https://github.com/catchorg/catch2
[11] GitHub Actions

5 Resultados

6 Conclusão e trabalhos futuros

Desde o início do Mestrado até o presente momento, foram realizadas as seguintes atividades:

- Iterações de design para layout da interface gráfica;
- Prototipagem da interface gráfica;
- Implementação das principais funcionalidades e classes da interface gráfica utilizando a biblioteca ImGui;
- Implementação da Representação Intermediária e salvamento/carregamento de arquivos do computador;

Espera-se que, até o final deste ano, o software seja capaz de realizar o fluxo completo de salvamento/carregamento, criação do modelo, simulação *onsite*, geração de código e exportação dos resultados das simulações em PDF.

Um trabalho futuro é desenvolver uma versão Web do software para que ele possa ser utilizado por um número maior de pessoas. A criação do modelo será feita através do navegador Web e todo o processamento incluindo a geração de código, execução, geração dos resultados e imagens será feita no servidor. As tecnologias usadas pelo projeto até então são todas compatíveis com a Web; para as linguagens C e C++ existe o projeto **Emscripten**[1], um compilador baseado na LLVM que é capaz de gerar WebAssembly. Já o Rust, também baseado na LLVM, possui suporte nativo à WebAssembly.

Outro trabalho futuro é a implementação da geração de código para modelos de Equações Diferenciais Parciais (EDPs) utilizando o método de diferenças finitas. Essa implementação iria exigir uma mudança maior na interface e na representação intermediária para acomodar características e recursos que são próprios de modelos de EDPs.

Por último, destaca-se que uma outra possibilidade de extensão do trabalho é a implementação de funcionalidades para realizar a estimativa de parâmetros do modelo de EDOs e a análise de sensibilidade do modelo. Essas novas funcionalidades seriam integradas ao software existente.

[1] https://emscripten.org/

Referências

ADOMIAN, G. Solving the mathematical models of neurosciences and medicine. *Mathematics and Computers in Simulation*, v. 40, n. 1, p. 107–114, 1995. ISSN 0378-4754. Disponível em: <https://www.sciencedirect.com/science/article/pii-/0378475495000218>.

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. Hardcover. ISBN 0321486811. Disponível em: <http://www.amazon.ca-/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0321486811>.

BUCELLI, M. et al. *A mathematical model that integrates cardiac electrophysiology, mechanics and fluid dynamics: application to the human left heart*. 2022.

BURCH, C. Logisim: A graphical system for logic circuit design and simulation. *ACM Journal of Educational Resources in Computing*, v. 2, p. 5–16, 03 2002.

ELLIS, T. O.; HEAFNER, J. F.; SIBLEY, W. L. *The GRAIL Language and Operations*. Santa Monica, CA: RAND Corporation, 1969.

FORTMANN-ROE, S. Insight maker: A general-purpose tool for web-based modeling and simulation. *Simulation Modelling Practice and Theory*, v. 47, p. 28–45, 09 2014.

HEINER, M. et al. Snoopy – a unifying petri net tool. In: HADDAD, S.; POMELLO, L. (Ed.). *Application and Theory of Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 398–407. ISBN 978-3-642-31131-4.

HEINER, M.; RICHTER, R.; SCHWARICK, M. Snoopy: A tool to design and animate/simulate graph-based formalisms. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008. (Simutools '08), p. 15:1–15:10. ISBN 978-963-9799-20-2. Disponível em: <http://dl.acm.org/citation.cfm?id=1416222.1416242>.

HLADISH, T. et al. Epifire: An open source c++ library and application for contact network epidemiology. *BMC bioinformatics*, v. 13, p. 76, 05 2012.

LIU, F. *Colored Petri Nets for systems biology*. Tese (doctoralthesis) — BTU Cottbus - Senftenberg, 2012.

MALONEY, J. et al. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, Association for Computing Machinery, New York, NY, USA, v. 10, n. 4, nov 2010. Disponível em: <https://doi.org/10.1145/1868358.1868363>.

MORARU, I. et al. Virtual cell modelling and simulation software environment. *Systems Biology, IET*, v. 2, p. 352 – 362, 10 2008.

REIS, R. F. et al. A validated mathematical model of the cytokine release syndrome in severe covid-19. *Frontiers in Molecular Biosciences*, Frontiers, p. 680, 2021.

SCHAFF, J. et al. A general computational framework for modeling cellular structure and function. *Biophysical Journal*, v. 73, n. 3, p. 1135–1146, 1997. ISSN 0006-3495. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0006349597781463>.

SHANNON, P. et al. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, Cold Spring Harbor Lab, v. 13, n. 11, p. 2498–2504, 2003.

SPENCER, S. L. et al. An ordinary differential equation model for the multistep transformation to cancer. *Journal of Theoretical Biology*, Elsevier, v. 231, n. 4, p. 515–524, 2004.

TAKALA, T.; MäKäRäINEN, M.; HAMALAINEN, P. Immersive 3d modeling with blender and off-the-shelf hardware. In: . [S.l.: s.n.], 2013. p. 191–192. ISBN 978-1-4673-6097-5.

TALKINGTON, A.; DANTOIN, C.; DURRETT, R. Ordinary differential equation models for adoptive immunotherapy. *Bulletin of mathematical biology*, Springer, v. 80, p. 1059–1083, 2018.

VIGMOND, E. et al. Solvers for the cardiac bidomain equations. *Progress in Biophysics and Molecular Biology*, v. 96, n. 1, p. 3–18, 2008. ISSN 0079-6107. Cardiovascular Physiome. Disponível em: <https://www.sciencedirect.com/science/article/pii-/S0079610707000740>.

VIRTANEN, P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, v. 17, p. 261–272, 2020.

.1 Código de Python gerado para o modelo Predador-Presa

Código completo

```
import scipy
import numpy as np

def initial_values() -> np.ndarray:
    H_0 = 100.0
    P_0 = 5.0
    return np.array((
        H_0,
        P_0,
    ))


def constants() -> list:
    a = 0.015
    b = 0.0125
    m = 0.8
    r = 0.2
    return [
```

```
        a,
        b,
        m,
        r,
    ]


def system(t: np.float64, y: np.ndarray, *constants) -> np.ndarray:
    H,P, = y

    a,b,m,r, = constants

    dH_dt = (r * H ) - (a * (H * P ) )
    dP_dt = ((H * P ) * b ) - (P * m )

    return np.array([dH_dt,dP_dt])


def simulate(st=0, tf=10, dt=0.1):
    sim_steps = np.arange(st, tf + dt, dt)

    simulation_output = scipy.integrate.solve_ivp(
        fun=system,
        t_span=(0, tf + dt),
        y0=initial_values(),
        args=constants(),
        t_eval=sim_steps,
    )

    # ...
```