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

Dissertação apresentada como requisito para obtenção do título de Mestre em Ciência da Computação 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

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

São João del-Rei, 4 de março de 2024

Alexandre Bittencourt Pigozzo
Orientador

Prof. Rafael Sachetto Oliveira
Convidado 1

Prof. Marcelo Lobosco
Convidado 2

Profa. Bárbara de Melo Quintela
Convidado 3

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 nesta 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. Frequentemente, 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 por exemplo, e estudantes iniciando carreiras acadêmicas. Apesar de existirem softwares que auxiliam o desenvolvimento de modelos computacionais, estes frequentemente apresentam interfaces muito complexas na busca para se tornarem genéricos o suficiente e fornecerem muitos recursos. Neste trabalho, é apresentado o software que foi desenvolvido para construir e simular modelos de Equações Diferenciais Ordinárias. O objetivo foi desenvolver um software simples, fácil de usar e de estender com novas funcionalidades. Através da interface gráfica, o usuário pode construir um modelo utilizando um editor baseado em nós, realizar simulações e gerar o código que implementa o modelo. O software poderá ser utilizado na pesquisa e no processo ensino-aprendizagem de modelagem computacional.

Abstract

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

The areas of Mathematical and Computational Modeling have become increasingly important in today's world, in which scientific studies must bring increasingly faster results. Mathematical and computational models emerge as powerful tools for studying and understanding complex systems and can be used by researchers from several different areas. Often, models commonly require extensive mathematical knowledge to be created, which results in a high entry barrier for scientists without mathematical training, such as biologists, for example, and students starting academic careers. Although there is software that helps the development of computational models, they often present very complex interfaces in order to become generic enough and provide many resources. In this work, the software that was developed to build and simulate models of Ordinary Differential Equations is presented. The objective was to develop a simple software, easy to use and extend with new features. Through the graphical user interface, the user can build a model using a node-based editor, perform simulations and generate the code that implements the model. The software can be used in research and in the teaching-learning process of computational modeling.

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. Nesse contexto, foi desenvolvido um software para auxiliar as etapas de implementação e simulação de modelos de Equações Diferenciais Ordinárias (EDOs).

O principal objetivo deste trabalho foi 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 menos tempo seja investido na implementação do modelo. A partir do "desenho" de um modelo, 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, modela 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 de EDOs. 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", isto é, esse princípio nos diz que quanto maior é a concentração de duas substâncias maior é a chance delas interagirem, partindo da suposição de que as substâncias em questão possuem a capacidade e "afinidade"para interagir.

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 no modelo.

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

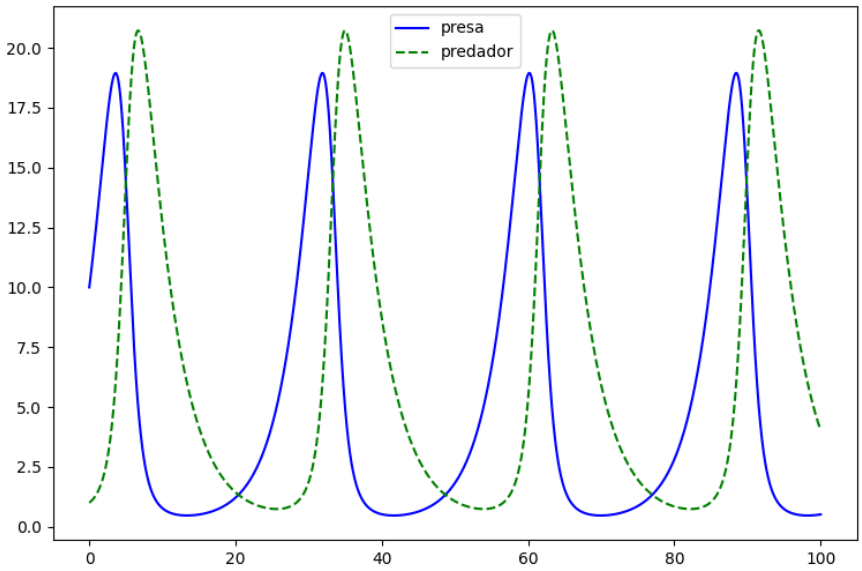

Figura 1 – Exemplo de resultado do modelo Predador-Presa clássico.

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

[1] Documentação da Unreal Engine

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.

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. As Árvores de Sintaxe Abstratas geralmente são utilizadas para realizar diversas verificações semânticas no código e elas podem ser a entrada para o algoritmo que gera uma outra RI mais próxima da linguagem alvo como, por exemplo, o Código de Três Endereços. O Código de Três Endereços é uma representação que está próxima da linguagem Assembly e pode ser usada como base para várias otimizações independentes de 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. Os Grafos Acíclicos Dirigidos são muito usados para realizar algumas otimizações no código como, por exemplo, verificação de variável morta, propagação de constantes e cópias, entre outras otimizações.

Como uma alternativa mais abstrata ao Assembly (que é dependente da arquitetura alvo), existe a linguagem intermediária da LLVM (LLVM *Intermediate Representation*), que pode ser transpilada para o Assembly da máquina alvo. A LLVM é uma infraestrutura para o desenvolvimento de compiladores com o objetivo de facilitar o desenvolvimento de compiladores abstraindo toda a parte do backend. O desenvolvedor não precisa se preocupar em conhecer e programar para as arquiteturas alvo com as quais quer trabalhar porque, considerando que a linguagem criada 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 construídos no software. A RI desenvolvida será apresentada e explicada na Seção 4.1.2.

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 facilitar a injeção de código nos *templates*.

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

```
from scipy.integrate import solve_ivp
import numpy as np
import pandas as pd

def odeSystem(t, P, {% for key, value in params %}{%if loop.is_last%} {{
    key}} {%else%} {{key}}, {%endif%}{%endfor%}):
## for key, value in vars
    {{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

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

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 do modelo, simulação, modificação do 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 computacional 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 2, 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.

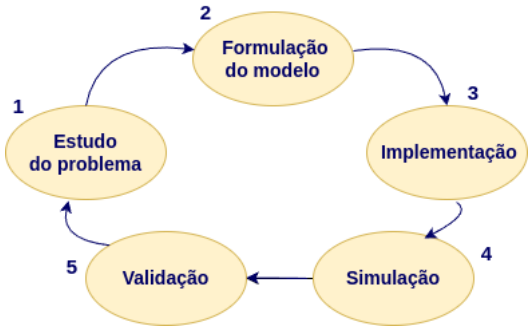

Figura 2 – Visão geral do ciclo da modelagem.

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 dessa abordagem 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 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 algum problema escolhido por ele para apresentar e discutir conceitos e/ou técnicas de modelagem de forma semelhante à metodologia *Problem Based Learning* (PBL) [refs]. O seguinte conjunto de passos é sugerido para o processo de estudo, modelagem e simulação computacional:

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. Avaliação dos resultados do modelo com a ajuda do professor: caso o comportamento desejado ou esperado não seja obtido, voltar para um dos passos anteriores como, por exemplo, o passo 2 ou 3 e repetir o processo.

A ideia dessa abordagem é colocar o aluno como protagonista no processo de construção do próprio conhecimento porque o aluno estará ativamente realizando todas as etapas de modelagem do problema. O software irá facilitar a construção e simulação do modelo de forma que os alunos poderão gastar mais tempo formulando as hipóteses e equações do modelo, planejando os cenários a serem simulados e avaliando os resultados obtidos.

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.

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 neste trabalho possui alguns diferenciais 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 que facilita a construção e simulação de modelos, 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.
- O software permite ao usuário acessar o código-fonte usado para simular o modelo, possibilitando que o usuário conheça o código usado na simulação e também possa utilizar e alterar o código como desejar.

Poucos *softwares* encontrados na literatura 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, 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; RICH-

TER; 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 visual utilizada neste trabalho. A partir do software Snoopy, foi possível observar que modelos gráficos, além do atrativo visual, facilitam a construção do modelo, e um melhor entendimento do que está sendo feito e 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 diversos recursos 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 membranas, 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 [3] (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.

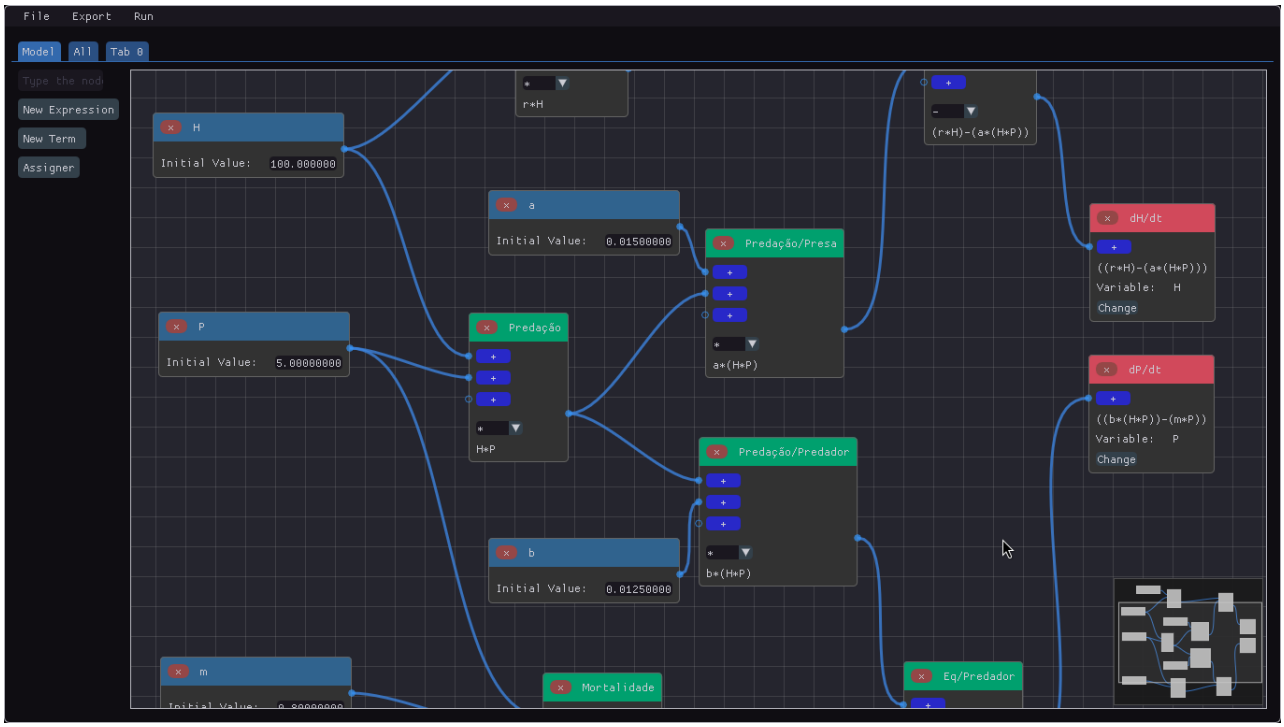

Figura 3 – Exemplo do modelo Predador-Presa (2.1) construído no *software*.

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 [4], 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 5.

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.

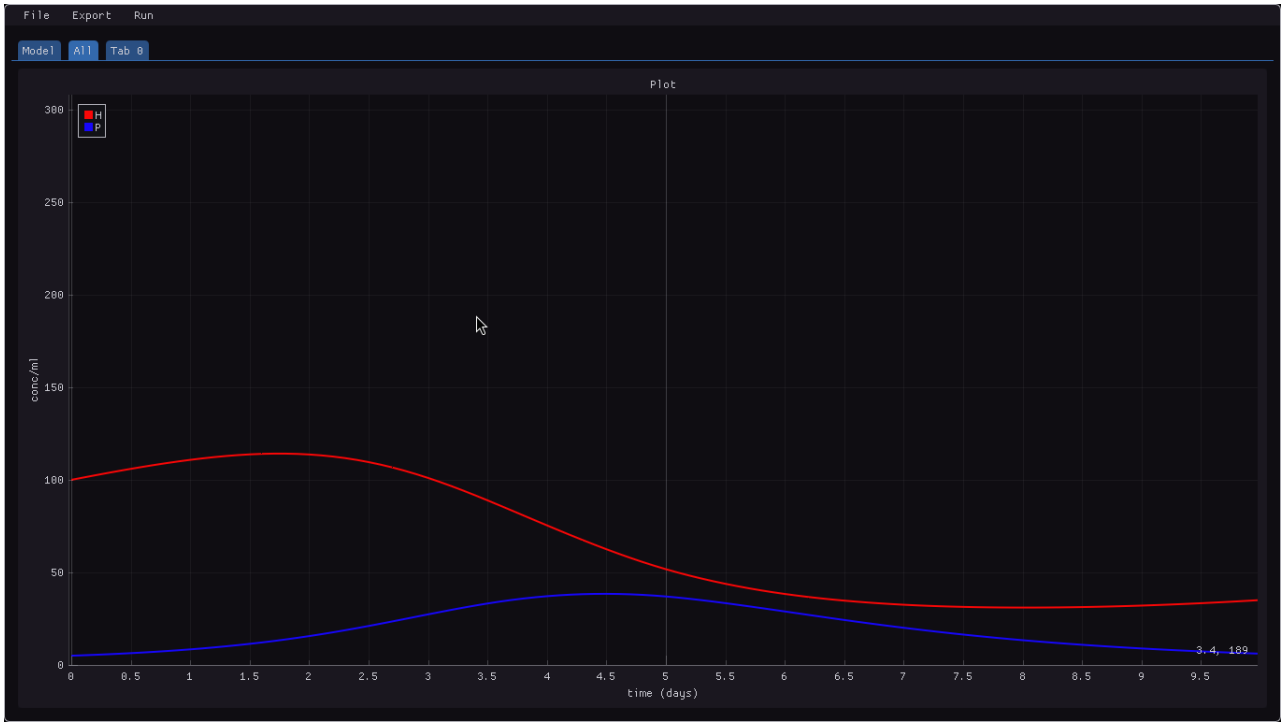

Figura 4 – Plotagem dos resultados da simulação do modelo Predador-Presa.

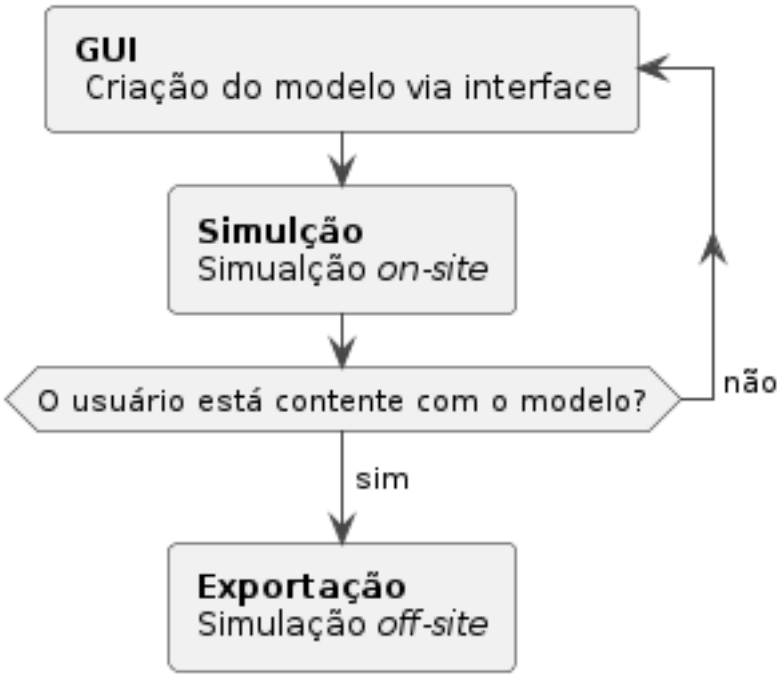

Figura 5 – Fluxograma da experiência de usuário esperada para o software.

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.

A construção de EDOs se baseia em alguns princípios gerais, como a Lei de Ação de Massas. 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 Representação de expressões na interface

Na GUI, os nós de expressão apresentam a expressão que estão construindo, para que o usuário não tenha que inspecionar cada nó individiualmente para determinar como uma expressão está sendo formada. Desta forma, não é necessário observar apenas a

equação final. Isto é feito por meio de árvores de expressões que cada um destes nós carrega consigo.

Na imagem 6 são apresentadas as árvores de expressões de todos os nós presentes em 3. Note que os nós 'Predação/Presa' e 'Predação/Predador' referenciam o mesmo nó base 'Predação'. Isto demonstra expressões podem ser usadas como entradas para outras expressões na forma de subexpressões. As subexpressões, além de possibilitar o usuário de evitar repetições na construção de modelos, permitem que os nós façam alterações relevantes para o contexto, como a adição de parênteses nas operações de multiplicação e divisão.

Esta representação foi desenvolvida de maneira agnóstica ao modelo, com o objetivo de permitir que o software seja extensível e, no futuro, oferte outros tipos de modelos. Um exemplo deste esforço se dá pela árvore de expressões não supôr que toda operação é infixada, também permitindo operações pós-fixadas. Exemplos da estensibilidade serão apresentados em 4.3, enquanto o futuro será discutido em **??**.

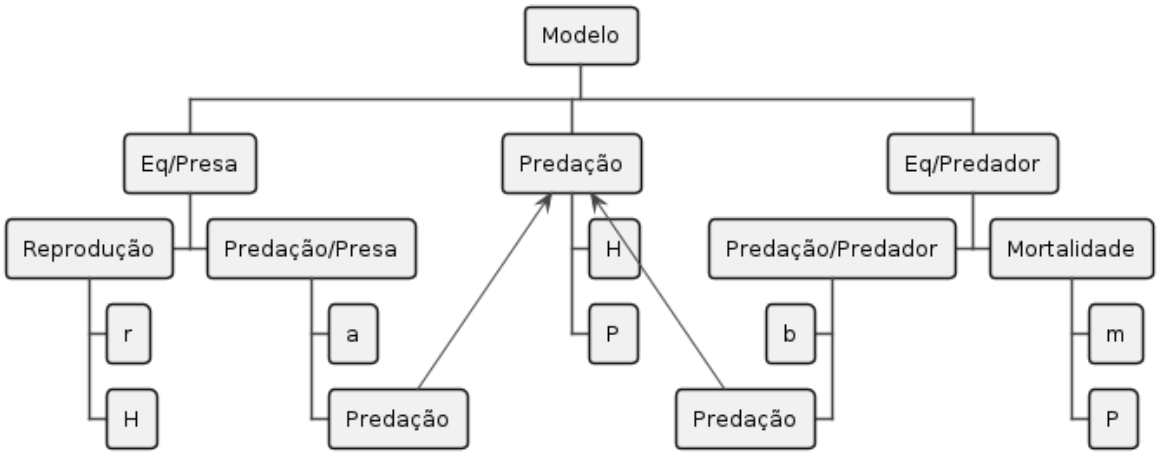

Figura 6 – Árvores de expressões para cada nó de expressão no modelo predador-presa [3]

4.1.2 Uma representação intermediária humanamente legível

Com a finalidade de possibilitar a escrita e leitura em disco dos modelos desenhados na GUI, uma RI foi definida. O formato JSON foi escolhido por sua popularização com a *web* e com sua clara representação de estruturas de dados como listas e tabelas *hash* que facilitam o desenvolvimento do software. A escolha de um formato pré-estabelecido também torna a RI mais resiliente a futuras mudanças, uma vez que não será necessário modificar a sintaxe do formato e nem as bibliotecas que o interpretam.

Apesar de ser usada como forma de armazenar informações do modelo, a RI possui um formato projetado para ser humanamente legível. Esta decisão foi tomada para possibilitar a recuperação do modelo construído sem a necessidade de ter o software instalado. Assim, caso o software caia em desuso, se torne impossível de instalar por algum motivo

(um sistema operacional diferente, por exemplo) ou simplesmente tenha mudanças que quebrem compatibilidade implementadas, não significa que o usuário tenha perdido seu modelo.

Em 4.1, pode-se observar o trecho da RI do modelo Predador-Presa [.2] que implementa a equação da presa. Note que é possível reconstruir a equação manualmente: cada entrada em `"arguments"` representa um termo (representado pelos campos nome e valor) ou uma combinação (representada pelos campos nome, operação e composição). Com este último, uma expressão pode ser construída a partir da repetida concatenação de todos os elementos de sua composição, inserindo a operação entre eles. Seguindo esta regra, tem-se as seguintes expressões:

- `"Predacao/Presa"`: $(+a) * (+H) * (+P)$, que simplificando se resume a $a * H * P$;
- `"Reprodução"`: $(+r) * (+H)$, que simplificando se resume a $r * H$;
- `"Eq/Presa"`: A subtração de `"Predacao/Presa"` e `"Reprodução"`, que resulta em $r * H - a * H * P$;

```
"arguments": [
    { "name": "Eq/Presa", "operation": "-",
        "composition": [
        { "name": "Reproducao", "contribution": "+" },
        { "name": "Predacao/Presa", "contribution": "+" }
    ]},
    { "name": "Reproducao", "operation": "*",
        "composition": [
        { "name": "r", "contribution": "+" },
        { "name": "H", "contribution": "+" }
    ]},
    { "name": "Predacao/Presa", "operation": "*",
        "composition": [
        { "name": "a", "contribution": "+" },
        { "name": "H", "contribution": "+" },
        { "name": "P", "contribution": "+" }
    ]},
    { "name": "H", "value": 100.0 },
    { "name": "P", "value": 5.0 },
    { "name": "r", "value": 0.2 },
    { "name": "a", "value": 0.015 }
]
```

Listing 4.1 – Trecho da RI que define termos e combinações usadas para definir a equação da presa.

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 ao usuário simular o modelo (plotando os resultados na interface), gerar o código equivalente em Python ou exportar um PDF com os resultados.

A geração de código é feita por meio da injeção de informações do modelo em marcações presentes no *template* e se baseia na substituição de *strings*. Os *templates* incluem o código base necessário para a resolução numérica, simulação e plotagem dos resultados com a inclusão de todas as bibliotecas necessárias. Os templates são escritos de maneira sucinta e legível com o objetivo de servir como um material de aprendizagem estensível.

Na listagem de código 4.2, observa-se parte do template que extrai o valor de variáveis e constantes para serem usados na simulação do próximo passo de tempo. Neste trecho, nota-se o uso de estruturas de controle como condicionais (`if`) e laços de repetição (`for`). Baseando a geração de código no sistema de *templates*, é possível adicionar novas formas de simulações sem alterar o código do programa. Por exemplo, é possível adicionar um *template* para gerar o código que simulará o modelo de forma estocástica usando o algoritmo de Gillespie.

```
23 def system(t: np.float64, y: np.ndarray, *constants) -> np.ndarray:
24     {% for arg in populations -%}
25     {{- arg.name }},
26     {%- endfor %} = y
27 
28     {%- if constants %}
29     {% for arg in constants -%}
30     {{- arg.name }},
31     {%- endfor %} = constants
32     {% endif -%}
33     # ...
```

Listing 4.2 – Trecho do *template* responsável por extrair o valor de variáveis e constantes

O 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 4.5, assim como será discutida a solução implementada.

4.3 Estensibilidade

É improvável que seja possível construir um software que atenda todas as necessidades de um público-alvo, e o software desenvolvido neste trabalho não é exceção; seria difícil determinar quais são as funcionalidades mínimas necessárias para que seja possível construir qualquer sistema de EDOs. Visando, então, maximizar os casos de uso nos quais este software pode ser utilizado, foi desenvolvido um sistema de extensões de usuário.

Além de prover os nós padrões [1] para a construção das EDOs exemplificadas, o software permite que o usuário desenvolva e utilize extensões para realizar operações que não estão prontamente disponíveis pela interface. Estas extensões são desenvolvidas em Python e incluem funções que serão interpretadas como nós na interface.

As funções decoradas por `@node` num arquivo de extensões serão interpretadas como nós. Seus nomes são usados em sua identificação na interface, e a quantidade de parâmetros que recebem determinam a quantidade de pinos de entradas. Por padrão, estes nós 'customizados' construirão expressões mais próximas da representação destas funções em linguagens de programação (como `funcao(param1, param2, ...)`), porém este comportamento pode ser sobrescrito com o parâmetro `format` no decorador.

Na listagem 4.3, é apresentado um exemplo de arquivo de extensões contendo um nó para calcular o seno de um parâmetro e um nó que eleva um parâmetro ao valor de outro. Este arquivo pode ser importado na interface, que possibilitará o uso dos nós como na figura 7.

```
34 import math
35
36 @node
37 def sin(x):
38     return math.sin(x)
39
40 @node(format='$1 ^ $2')
41 def pow(x, y):
42     return x ** y
```

Listing 4.3 – Exemplo de definição dos nós de extensão seno e potência com código Python válido.

Como estes nós são definidos utilizando Python, não há impacto na capacidade do software de simular o modelo e plotar os gráficos; as funções definidas nas extensões são diretamente incluídas no código final que será utilizado pela simulação. Ademais, os modelos salvos pela interface mantém uma referência aos arquivos de extensões utilizados

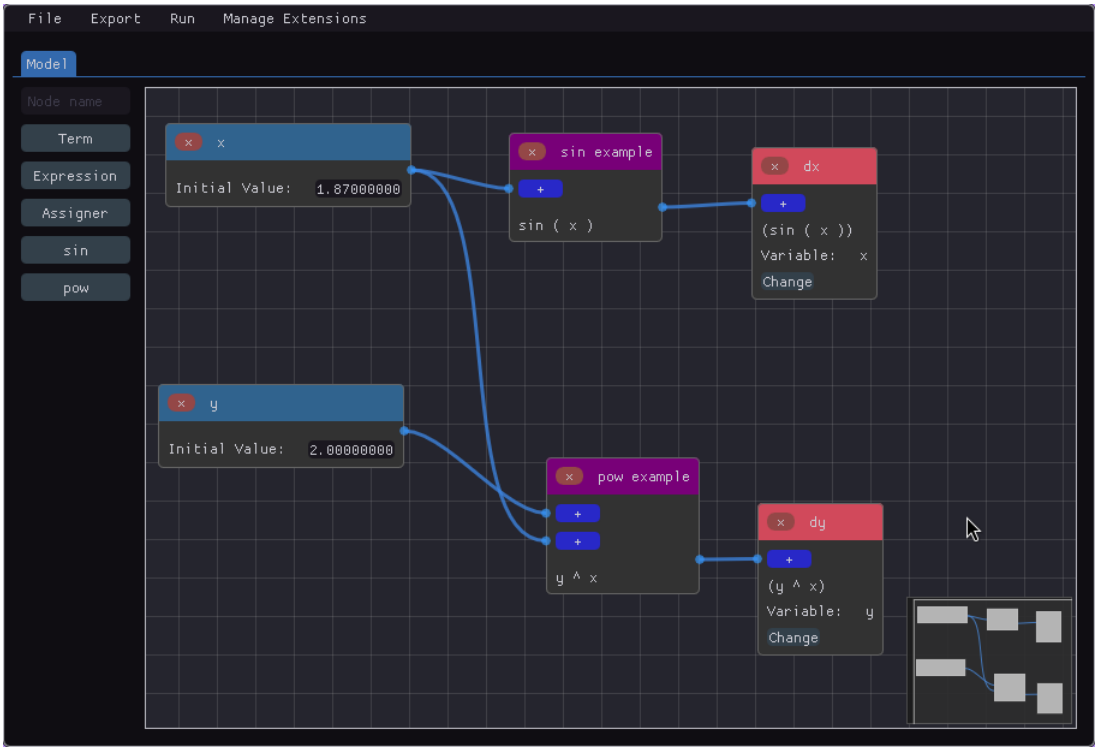

Figura 7 – Os nós definidos em 4.3 sendo utilizados na interface.

na construção do modelo, tornando possível que o software auto-importe as extensões necessárias quando carregar um modelo do disco.

4.3.1 Carregamento de extensões

Como mencionado anteriormente, os nós de extensões são definidos por funções que utilizam um decorador específico. Durante o carregemanto de um arquivo de extensões, o software injeta um trecho especial de código que define o decorador `@node` para inspecionar a função que está sendo decorada e emitir um JSON com as informações relevantes sobre a função como nome, quantidade e nome dos parâmetros e o formato da representação na interface. Essas informações servem como especificações para novos construtores de nós.

4.4 Implementação e Tecnologias utilizadas

O software desenvolvido foi dividido em dois módulos: (1) o responsável pela interface gráfica e (2) o responsável pela representação intermediária. O primeiro lida com a apresentação de gráficos, interações com o usuário e a plotagem dos resultados, enquanto o segundo define a representação em JSON dos modelos e realiza a transformação em código Python.

4.4.1 O módulo de GUI

TODO

4.4.2 A utilização de Rust para a GUI

O módulo de GUI [4.4.1] foi inicialmente desenvolvido em C++ e depois todo o código do software foi migrado para Rust (`https://www.rust-lang.org/pt-BR`). Um dos motivos foi o fato de que rapidamente se tornou desafiador manter uma grande base de código livre de *bugs* relacionados a ponteiros nulos e liberações duplas (*double frees*) porque manter o software livre desses erros exige experiência de programação e um profundo conhecimento da linguagem sendo usada.

A importância de evitar *bugs* relacionados à segurança de memória foi ressaltada pela agência de segurança nacional dos EUA em novembro de 2022[1], que, no documento citado, recomendou o uso das chamadas linguagens seguras no uso da memória (*memory safety languages*) como, por exemplo, Rust, C#, Go, Java, Python and Swift. A segurança de memória é uma propriedade de algumas linguagens de programação que evita que os programadores introduzam certos tipos de *bugs* relacionados ao uso da memória como ponteiros pendentes, ponteiros nulos, *buffer overflow*, entre outros. Outro motivo para a mudança de linguagem foi a disponibilidade de bibliotecas na linguagem Rust que facilitaram bastante o trabalho de salvar e carregar a RI, e também de gerar código.

Outra questão importante, o módulo de RI [4.4.3], escrito em Rust, demandava códigos *wrappers* para que fosse possível utilizar a biblioteca em C++, dado que as linguagens não compartilham das mesmas estruturas de dados. Por exemplo, não é possível usar uma *string* de C++ (`std::string`) em Rust, ou vice-versa (`&str`/`String`).

Algumas abordagens podem ser usadas para solucionar o problema, mas cada uma trouxe sua própria série de complicações:

- O projeto cbindgen provê um binário capaz de gerar cabeçalhos de C ou C++ a partir de arquivos fontes de Rust e um arquivo de configuração.

  Para que a geração dos cabeçalhos seja possível, macros especiais devem ser usados para declarar estruturas, enumeradores e funções com representações válidas na ABI (*Application Binary Interface*) de C/C++.

  Apesar do valor da geração automática dos cabeçalhos, alguns problemas se destascam:

[1] Software Memory Safety | U/OO/219936-22

Todas as funções que oferecem interoperabilidade entre as linguagens devem usar ponteiros para tipos abstratos. Isto significa abdicar da segurança provida pelo Rust, e re-introduz problemas relacionados ao gerenciamento de memória e ponteiros nulos;

Ponteiros só podem ser liberados pela mesma linguagem que os alocou, uma vez que os modelos de alocação são diferentes entre as linguagens. Isto torna o problema de gerenciamento de memória manual ainda mais complicado do que o usual;

Em versões modernas de C++, as regras de 3/5/0 causadas pelo RAII (*Resource Acquisition Is Initialization*) muitas vezes tornam estruturas e enumeradores com tipos opacos impossíveis de serem construídos na linguagem.

- O projeto bindgen Um projeto que almeija ser a contra-parte do cbindgen, gerando definições em Rust para estruturas de C/C++. Como a biblioteca de RI é escrita em Rust, isto significaria acoplá-la ao projeto da interface gráfica, conhecendo seus detalhes de implementação;

- cxx.rs: TODO

4.4.3 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**[2] e **GitHub Actions**[3].

O software utiliza a biblioteca **serde**[4] 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, **mi-**

[2] https://github.com/eqrion/cbindgen
[3] cbindgen-action
[4] https://serde.rs/

nijinja[5], 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 **??**.

4.4.4 Testes unitários

O software atualmente também implementa testes unitários em C++ utilizando o *framework* de testes **Catch2**[6]. 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**[7], que fornecem máquinas em nuvem sob demanda, assim impedindo que o desenvolvedor se descuide e esqueça de executá-los.

4.5 Distribuição do software

[5] https://github.com/mitsuhiko/minijinja
[6] https://github.com/catchorg/catch2
[7] 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,
    )

    # ...
```

.2 RI do modelo Predador-Presa

Arquivo JSON