Aceleração de Simulações da Eletrofisiologia Cardíaca com JAX

Jonatas Dias Machado Costa

JUIZ DE FORA

JULHO, 2026

Aceleração de Simulações da Eletrofisiologia Cardíaca com JAX

Jonatas Dias Machado Costa

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Joventino de Oliveira Campos

Coorientador: Bernardo Martins Rocha

JUIZ DE FORA

JULHO, 2026

Aceleração de Simulações da Eletrofisiologia Cardíaca com JAX

Jonatas Dias Machado Costa

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Joventino de Oliveira Campos
Doutor em Modelagem Computacional

Bernardo Martins Rocha
Doutor em Modelagem Computacional

José Jerônimo Camata
Doutor em Engenharia Civil

Rodrigo Weber dos Santos
Doutor em Matemática

JUIZ DE FORA

15 DE JULHO, 2026

Resumo

A simulação computacional da eletrofisiologia cardíaca permite estudar a propagação elétrica no tecido cardíaco e os cerários de formação de arritmias, mas seu custo computacional cresce rapidamente com o refinamento da malha, o número de passos temporais e a complexidade dos modelos utilizados. Este trabalho avalia o uso de JAX como alternativa para acelerar simulações baseadas no modelo do monodomínio e em diferenças finitas. Inicialmente, uma implementação em NumPy em CPU foi usada como referência para verificar a consistência numérica da implementação em JAX em casos controlados. Em seguida, os experimentos passaram a concentrar-se no desempenho da execução em JAX, especialmente em GPU, no estudo de convergência de malha e na extensão do simulador para um caso tridimensional com máscara anatômica. Foram avaliados os modelos biestável, FitzHugh-Nagumo, Mitchell-Schaeffer e Bueno-Orovio. Os resultados indicaram equivalência prática entre NumPy e JAX nas verificações iniciais em dupla precisão, ganhos expressivos de desempenho com JAX nas malhas maiores e viabilidade da execução do monodomínio 3D mascarado em GPU.

Palavras-chave: Eletrofisiologia cardíaca, JAX, GPU, monodomínio.

Abstract

Computational simulation of cardiac electrophysiology makes it possible to study electrical propagation in cardiac tissue and the scenarios leading to arrhythmia formation, but its computational cost increases rapidly with mesh refinement, the number of time steps, and the complexity of the models used. This work evaluates JAX as an alternative for accelerating simulations based on the monodomain model and finite differences. First, a NumPy implementation on CPU was used as a reference to verify the numerical consistency of the JAX implementation in controlled cases. The experiments then focused on the performance of JAX execution, especially on GPU, on grid convergence analysis, and on the extension of the simulator to a three-dimensional case with an anatomical mask. The bistable, FitzHugh-Nagumo, Mitchell-Schaeffer, and Bueno-Orovio models were evaluated. The results indicated practical equivalence between NumPy and JAX in the initial double-precision verification tests, substantial performance gains with JAX on larger meshes, and feasibility of running the masked 3D monodomain simulation on GPU.

Keywords: Cardiac electrophysiology, JAX, GPU, monodomain.

Agradecimentos

Ao meu pai, Fabiano, e à minha mãe, Neri, por terem acreditado em mim e me apoiado em cada etapa da minha vida. Se tenho a oportunidade de apresentar esse trabalho, é por conta do amor e do cuidado de vocês.

Aos meus avós, Altari e Marilza, por terem sido referências de carinho na minha vida desde sempre. Agradeço por terem me influenciado positivamente, me fazendo acreditar que posso alcançar meus sonhos.

Ao meu orientador, Joventino, e ao meu coorientador, Bernardo, pela paciência e auxílio não só no desenvolvimento desse projeto, mas também na minha formação como pesquisador. Sou imensamente grato por ter tido a oportunidade de trabalhar em projetos acadêmicos junto a eles na FISIOCOMP.

Aos meus amigos, que tornaram meus dias enquanto acadêmico mais leves e divertidos.

À UFJF e à FAPEMIG (APQ-02752-24) pelo fomento do trabalho desenvolvido.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

BO	Bueno-Orovio, Cherry e Fenton
CFL	Courant-Friedrichs-Lewy
CPU	Unidade Central de Processamento
FHN	FitzHugh-Nagumo
GPU	Unidade de Processamento Gráfico
JIT	Just-in-Time
L2	Norma euclidiana
MS	Mitchell-Schaeffer
RAM	Memória de Acesso Aleatório
S1-S2	Protocolo de dois estímulos
STL	*Standard Triangle Language*
VTK	*Visualization Toolkit*
WSL2	*Windows Subsystem for Linux 2*
XLA	*Accelerated Linear Algebra*

1 Introdução

1.1 Contextualização e Motivação

A simulação computacional da eletrofisiologia cardíaca é um componente relevante no processo de criação de gêmeos digitais humanos, com potencial de catalisar inovações no âmbito de doenças cardiovasculares (SEL et al., 2024). Um gêmeo digital do coração consiste em uma réplica virtual e personalizada da atividade elétrica cardíaca de um paciente, construída a partir de seus dados, com a qual é possível investigar mecanismos de arritmia, planejar e avaliar estratégias terapêuticas de forma não invasiva (SEL et al., 2024; BERG et al., 2026). Em particular, fenômenos complexos como a reentrada elétrica e as ondas espirais no miocárdio, difíceis de observar diretamente, podem ser reproduzidos e analisados em detalhe por meio dessas simulações (SUNDNES et al., 2006; COMTOIS; KNELLER; NATTEL, 2005; FENTON et al., 2002).

Para que esse potencial se concretize na prática clínica, essas simulações precisam ser ao mesmo tempo numericamente confiáveis e computacionalmente *eficientes*: acelerar a execução só traz benefício se o resultado permanecer fiel ao modelo. A construção e a calibração de gêmeos digitais, a simulação de geometrias realistas e a execução de simulações para muitos pacientes demandam um volume elevado de cálculo, o que torna o tempo de solução um fator crítico (BERG et al., 2026). Esforços recentes têm buscado simuladores capazes de explorar de forma escalável o poder de processamento de unidades de processamento gráfico (GPUs), reduzindo o tempo necessário para simulações de larga escala (ROCHA et al., 2011; MENA; FERRERO; MATAS, 2015; CAMPOS et al., 2016; OLIVEIRA et al., 2018; BERG et al., 2026). É na busca por simuladores eficientes da eletrofisiologia cardíaca que este trabalho se insere.

Esse custo computacional decorre dos processos de discretização espacial e temporal, aumentando com o crescimento do domínio discretizado e com a utilização de passos de tempo menores (SUNDNES et al., 2006). Na implementação dessas simulações, a escolha das ferramentas se torna decisiva. NumPy é uma biblioteca tradicional para

computação numérica baseada em arrays, enquanto JAX oferece recursos voltados ao alto desempenho, como compilação Just-in-Time, execução em GPU e CPU e vetorização de operações sobre arrays, mantendo ao mesmo tempo um código de alto nível em Python (HARRIS et al., 2020; BRADBURY et al., 2018; JAX, 2026b). Essas características tornam o JAX um candidato natural para acelerar simulações de eletrofisiologia cardíaca sem um alto custo de implementação.

Trabalhos anteriores já exploraram o uso de GPUs para acelerar simulações de eletrofisiologia cardíaca (MENA; FERRERO; MATAS, 2015; OLIVEIRA et al., 2018; VASCONCELLOS et al., 2020; KABOUDIAN; CHERRY; FENTON, 2021). É relevante então avaliar em que medida JAX pode ser usado para acelerar simulações desse tipo, preservando a consistência numérica em relação a uma implementação de referência. Neste trabalho, a implementação em JAX foi a principal e esteve presente em todos os casos. A implementação em NumPy serviu como referência de comparação nos casos iniciais, em que a execução em CPU ainda era viável. Nas malhas mais refinadas e na extensão tridimensional, em que o custo em CPU se torna proibitivo, apenas a implementação em JAX em GPU foi utilizada.

O trabalho avaliou modelos iônicos com diferentes níveis de complexidade, incluindo os modelos biestável, FitzHugh-Nagumo, Mitchell-Schaeffer e de Bueno-Orovio (MITCHELL; SCHAEFFER, 2003; FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962; BUENO-OROVIO; CHERRY; FENTON, 2008). Os testes foram organizados em domínios unidimensionais e bidimensionais para análise de consistência, desempenho e convergência de malha, e posteriormente estendidos para o monodomínio tridimensional com máscara anatômica. A abordagem de JAX em CPU foi mantida para observar o impacto das características do JAX além do processamento em GPU, mas o foco principal das simulações mais custosas foi a execução em JAX GPU.

1.2 Objetivos

1.2.1 Objetivo Geral

Avaliar o uso de JAX na simulação numérica da eletrofisiologia cardíaca, considerando desempenho computacional, convergência de malha e extensão para domínios tridimensionais.

1.2.2 Objetivos Específicos

1. Analisar o comportamento computacional das configurações avaliadas em simulações de propagação elétrica cardíaca com diferentes modelos iônicos.

2. Avaliar a influência do backend computacional, do tamanho da malha, do número de passos e do modelo iônico no tempo de execução das simulações.

3. Investigar a consistência numérica dos resultados obtidos pelas implementações em NumPy e JAX em casos representativos.

4. Identificar os cenários em que a execução em JAX GPU apresenta vantagem em relação à implementação de referência em NumPy, considerando desempenho, escalabilidade com o refinamento da malha e o esforço de adaptação da implementação.

1.3 Organização do Trabalho

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os conceitos teóricos necessários para o desenvolvimento do simulador, incluindo a propagação elétrica cardíaca, o modelo do monodomínio, os modelos iônicos avaliados e os recursos computacionais utilizados. O Capítulo 3 descreve a metodologia, incluindo a implementação em NumPy e JAX, os cenários avaliados, os critérios de comparação entre backends, o estudo de convergência e a extensão para o monodomínio 3D com máscara anatômica. O Capítulo 4 apresenta os resultados obtidos nas verificações numéricas, nos benchmarks, nos estudos de convergência e na simulação tridimensional. Por fim, o Capítulo 5 reúne as conclusões, limitações e possibilidades de trabalhos futuros.

2 Fundamentação Teórica

2.1 Condução Ativa

A propagação elétrica em tecidos excitáveis ocorre por condução ativa. Nesse processo, um estímulo local altera o potencial de membrana de uma célula e pode desencadear a ativação de células vizinhas. Em vez de o sinal apenas se dissipar passivamente ao longo do tecido, a própria dinâmica celular regenera o impulso durante a propagação.

Esse mecanismo aparece tanto em neurônios quanto em células cardíacas, pois ambos são sistemas excitáveis governados por variações do potencial de membrana e pela abertura e fechamento de canais iônicos. Por isso, modelos simplificados originalmente associados à excitabilidade neural, como FitzHugh-Nagumo e Nagumo, também são úteis como aproximações matemáticas para estudar propagação elétrica em meios excitáveis (FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962).

No contexto cardíaco, essa ativação local precisa ser combinada com o acoplamento elétrico entre regiões vizinhas do tecido. Assim, a descrição matemática da propagação envolve dois componentes: um termo local, associado à dinâmica iônica de cada ponto, e um termo espacial, associado à difusão do potencial pelo tecido.

2.2 Formulação de Reação-Difusão

A combinação entre dinâmica local e acoplamento espacial pode ser representada por uma formulação de reação-difusão. Nessa estrutura, a reação descreve a variação local do potencial de membrana, determinada pelo modelo iônico adotado, enquanto a difusão descreve o espalhamento do potencial entre pontos vizinhos do domínio. De forma geral, a equação pode ser escrita como

$$\frac{\partial v}{\partial t} = k\nabla^2 v + f(v, \mathbf{q}), \tag{2.1}$$

onde v é o potencial elétrico, $\mathbf{q}$ representa as demais variáveis internas do modelo iônico, $k\nabla^2 v$ é o termo difusivo e $f(v, \mathbf{q})$ é o termo de reação local.

Essa forma aparece nos casos unidimensionais, bidimensionais e tridimensionais. A diferença entre eles está no domínio espacial e no cálculo do Laplaciano. Em uma dimensão, a difusão depende apenas da variação ao longo de x; em duas dimensões, depende das direções x e y; e, em três dimensões, depende das direções x, y e z. O papel dos modelos iônicos é fornecer o termo local $f(v, \mathbf{q})$ e as equações das variáveis internas.

2.3 Formulação do Monodomínio

O modelo do monodomínio é uma formulação clássica para representar a propagação elétrica em tecido cardíaco (SUNDNES et al., 2006). Ele também possui estrutura de reação-difusão, pois combina um termo espacial, associado à condução elétrica pelo tecido, com um termo local, associado às correntes iônicas da membrana celular.

Uma forma usual da equação do monodomínio é

$$\frac{\partial v}{\partial t} = \nabla \cdot (k \nabla v) - I_{ion}(v, \mathbf{q}) + I_{stim}, \tag{2.2}$$

onde k representa a condutividade ou difusividade efetiva do tecido, I_{ion} representa as correntes iônicas locais e I_{stim} representa um estímulo externo.

Neste trabalho, foi usada uma forma simplificada e normalizada dessa estrutura, compatível com os modelos iônicos considerados: tomando a difusividade k como constante no espaço e reunindo em um único termo de reação $f(v, \mathbf{q})$ as correntes iônicas e o estímulo, a equação recai exatamente na Eq. (2.1), já apresentada na formulação geral de reação-difusão. Assim, o monodomínio não substitui os modelos iônicos. Ele define como o potencial se propaga no espaço, enquanto os modelos iônicos definem como cada ponto do tecido se ativa, se recupera e retorna ao repouso. Na extensão tridimensional, essa mesma ideia foi aplicada em uma malha cartesiana mascarada, usando o modelo Mitchell-Schaeffer como termo de reação local.

2.4 Modelos Iônicos Considerados

Os modelos iônicos descrevem a dinâmica local do potencial de membrana e das variáveis internas associadas à recuperação ou às correntes celulares. Princípios de excitabilidade elétrica que governam a propagação em axônios neuronais são semelhantes aos observados em células cardíacas, permitindo o uso de modelos simplificados de excitabilidade para estudar a dinâmica elétrica do coração (SUNDNES et al., 2006; FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962).

Neste trabalho, os modelos iônicos foram separados do código responsável pela simulação espacial e temporal, facilitando a simulação de diferentes modelos. Para isso, foi definida uma interface modular composta por um termo de reação e, quando aplicável, por um termo de porta. Foram considerados nessa interface os modelos biestável, FitzHugh-Nagumo e Mitchell-Schaeffer. Além disso, o modelo minimal de Bueno-Orovio foi implementado separadamente, por possuir quatro variáveis de estado (FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962; MITCHELL; SCHAEFFER, 2003; BUENO-OROVIO; CHERRY; FENTON, 2008).

Os quatro modelos formam uma progressão de complexidade crescente, ilustrada na Figura 2.1, que mostra o potencial ao longo do tempo em uma célula isolada, sem o termo de difusão. O modelo biestável apenas chaveia entre dois estados de repouso e ativação, sem repolarização. O FitzHugh-Nagumo já reproduz um pulso de excitação completo, com retorno ao repouso. O Mitchell-Schaeffer acrescenta um platô antes da repolarização, e o Bueno-Orovio reproduz um potencial de ação com forma mais próxima da observada no tecido ventricular humano. Cada modelo é detalhado nas seções seguintes.

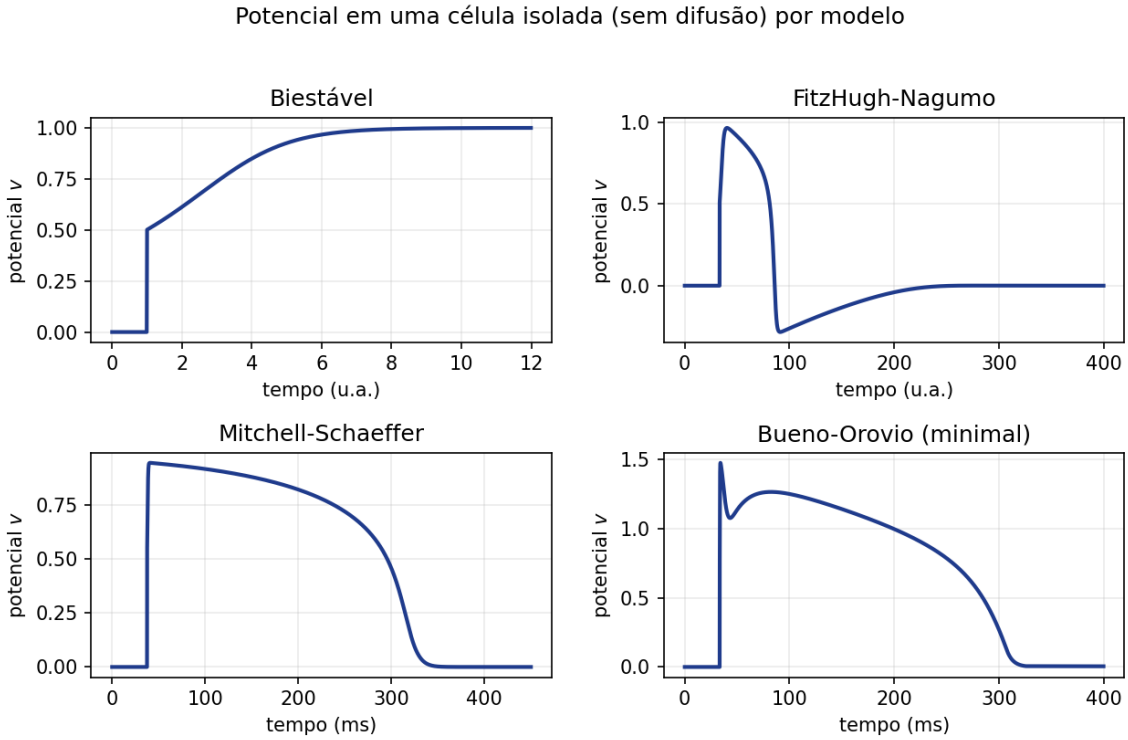

Figura 2.1: Potencial ao longo do tempo em uma célula isolada (sem difusão), para os quatro modelos iônicos considerados. A sequência ilustra a progressão de complexidade, do chaveamento biestável ao potencial de ação realista do modelo de Bueno-Orovio. As escalas de tempo seguem as unidades de cada modelo (adimensionais nos fenomenológicos, milissegundos nos demais).

2.5 Modelo Biestável

O modelo de condução ativa em um neurônio pode ser representado por uma função de reação cúbica, como no modelo de Nagumo (NAGUMO; ARIMOTO; YOSHIZAWA, 1962):

$$f(v) = Av(1 - v)(v - \alpha), \tag{2.3}$$

onde v é o potencial de membrana normalizado, α é o limiar de disparo e A é o fator de escala da reação.

A equação do modelo biestável completa, combinando o termo de reação cúbico

com a difusão espacial é dada por:

$$\frac{\partial v}{\partial t} = k\nabla^2 v + Av(1-v)(v-\alpha), \tag{2.4}$$

onde $\partial v/\partial t$ representa a taxa de variação do potencial ao longo do tempo, $\nabla^2 v$ representa a curvatura espacial do potencial (termo de difusão), e k é o coeficiente de difusão.

2.6 Modelo de FitzHugh-Nagumo (FHN)

O principal problema da equação biestável apresentada anteriormente é sua não fidelidade ao comportamento das células após sua ativação. Após a propagação de um impulso nervoso, as células têm seu potencial reduzido gradualmente até o retorno ao repouso. No modelo biestável, as células atingem $v = 1$ e permanecem nesse estado indefinidamente.

Para resolver esse problema, é introduzida uma variável de recuperação w, que cresce lentamente após o disparo, reduzindo o valor de v até sua eventual inativação. Por crescer muito mais lentamente que v, essa variável introduz um período refratário, no qual não podem ocorrer novos disparos da célula.

As equações representativas do processo de condução ativa seguindo o modelo FitzHugh-Nagumo são (FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962):

$$\frac{\partial v}{\partial t} = k\nabla^2 v + Av(1-v)(v-\alpha) - w, \tag{2.5}$$

$$\frac{\partial w}{\partial t} = \epsilon(v - \gamma w), \tag{2.6}$$

onde w é a variável de recuperação, ϵ controla a velocidade de recuperação e γ controla o equilíbrio de w.

2.7 Modelo de Mitchell-Schaeffer (MS)

O modelo Mitchell-Schaeffer foi utilizado como uma progressão natural em relação ao modelo FitzHugh-Nagumo. Enquanto o modelo FHN permite representar a excitação e a recuperação de um meio excitável de forma simplificada, o modelo Mitchell-Schaeffer

foi proposto especificamente para descrever a dinâmica do potencial de ação cardíaco por meio de um modelo de duas correntes (MITCHELL; SCHAEFFER, 2003).

Essa escolha permite trabalhar com um modelo computacionalmente simples e, ao mesmo tempo, mais próximo do comportamento cardíaco do que o FHN. Mitchell e Schaeffer (2003) apresentaram o modelo como uma alternativa útil para simulações numéricas em duas e três dimensões, nas quais a eficiência computacional é importante. Além disso, suas constantes de tempo permitem representar aspectos associados à iniciação, ao platô, ao decaimento e à recuperação do potencial de ação. Dessa forma, ele é adequado para testar a propagação elétrica em casos mais complexos, como espirais em duas dimensões e a extensão tridimensional com máscara anatômica, sem exigir o custo computacional de modelos iônicos mais detalhados.

No modelo Mitchell-Schaeffer, a variável principal v representa o potencial de ação normalizado e a variável h representa uma porta de recuperação. A equação usada na simulação pode ser escrita como

$$\frac{\partial v}{\partial t} = k\nabla^2 v + J_{in}(v,h) + J_{out}(v), \tag{2.7}$$

$$\frac{\partial h}{\partial t} = \begin{cases} \dfrac{1-h}{\tau_{open}}, & v < v_{gate}, \\ -\dfrac{h}{\tau_{close}}, & v \geq v_{gate}. \end{cases} \tag{2.8}$$

As correntes de entrada e saída são dadas por

$$J_{in}(v,h) = \frac{hv^2(1-v)}{\tau_{in}}, \tag{2.9}$$

$$J_{out}(v) = -\frac{v}{\tau_{out}}. \tag{2.10}$$

O termo J_{in} é responsável pela ativação rápida do potencial, enquanto J_{out} representa a tendência de retorno do potencial ao repouso. A variável h controla a disponibilidade da corrente de entrada: quando v está abaixo do limiar v_{gate}, h tende a se recuperar; quando v está acima desse limiar, h decai. Com isso, o modelo representa a perda temporária de excitabilidade após a ativação, associada ao período refratário.

Os parâmetros do modelo controlam as escalas de tempo das correntes e da

variável de recuperação. Neste trabalho, foram usados os mesmos valores em todos os casos Mitchell-Schaeffer, tanto nas simulações bidimensionais quanto na simulação tridimensional, que são apresentados na Tabela 2.1.

Tabela 2.1: Descrição e valores dos parâmetros usados no modelo Mitchell-Schaeffer. O conjunto de valores segue a formulação original do modelo (MITCHELL; SCHAEFFER, 2003).

Parâmetro	**Valor**	**Interpretação**
τ_{in}	0,3 ms	Constante de tempo da corrente de entrada, associada à ativação rápida do potencial.
τ_{out}	6,0 ms	Constante de tempo da corrente de saída, associada ao retorno do potencial ao repouso.
τ_{open}	120,0 ms	Constante de tempo da recuperação de h quando o potencial está abaixo do limiar.
τ_{close}	150,0 ms	Constante de tempo do decaimento de h quando o potencial está acima do limiar.
v_{gate}	0,13 (adim.)	Limiar que define a mudança de regime da variável h.

Na condição inicial em repouso, foi adotado $v = 0$ e $h = 1$. Assim, a corrente de entrada está inicialmente disponível, permitindo a ativação do tecido quando um estímulo eleva o potencial acima do limiar. Após a ativação, h diminui, reduzindo temporariamente a excitabilidade local.

2.8 Modelo de Bueno-Orovio

Assim como o Mitchell-Schaeffer, o modelo mínimo de Bueno-Orovio (BUENO-OROVIO; CHERRY; FENTON, 2008) também busca equilibrar a fidelidade ao comportamento cardíaco e o custo computacional, e representa o próximo nível dessa progressão de complexidade. Enquanto o modelo Mitchell-Schaeffer descreve o potencial de ação por meio de apenas duas variáveis e duas correntes principais, o modelo mínimo utiliza quatro variáveis

de estado e representa a corrente transmembrana total por três componentes agregadas: uma corrente rápida de entrada, uma corrente lenta de entrada e uma corrente de saída.

Essa maior complexidade permite representar uma dinâmica mais rica do potencial de ação ventricular humano, incluindo propriedades relevantes em tecido. Ao mesmo tempo, o modelo foi proposto com a preocupação de manter um custo computacional menor que o de modelos iônicos mais detalhados. Por isso, sua inclusão neste trabalho permite avaliar o simulador em um caso mais exigente do ponto de vista numérico e computacional, mas ainda compatível com simulações em malhas maiores.

No modelo mínimo de Bueno-Orovio, a variável u representa o potencial transmembrânico normalizado, enquanto v, w e s representam variáveis de porta associadas à recuperação e às correntes iônicas. A notação deste modelo exige um alerta: por fidelidade à formulação original (BUENO-OROVIO; CHERRY; FENTON, 2008), este modelo preserva os símbolos adotados por seus autores, que não coincidem com os usados no restante do texto. Aqui o potencial transmembrânico é denotado por u, enquanto nos demais modelos e no Capítulo 3 esse papel cabe a v; da mesma forma, v e w designam variáveis de porta, com significado distinto do que possuem no modelo de FitzHugh-Nagumo, em que w é a variável de recuperação. Em resumo, no modelo de Bueno-Orovio o papel desempenhado por v nos demais modelos cabe a u. A dinâmica local usada neste trabalho segue a formulação epicárdica do modelo (BUENO-OROVIO; CHERRY; FENTON, 2008). No caso espacial, o termo difusivo é somado apenas à equação de u; as equações abaixo descrevem o termo local do modelo:

$$\frac{\partial u}{\partial t} = k\nabla^2 u - \left(J_{fi} + J_{so} + J_{si}\right), \tag{2.11}$$

$$\frac{\partial v}{\partial t} = (1 - H_v)\frac{v_\infty - v}{\tau_v^-} - H_v\frac{v}{\tau_v^+}, \tag{2.12}$$

$$\frac{\partial w}{\partial t} = (1 - H_w)\frac{w_\infty - w}{\tau_w^-} - H_w\frac{w}{\tau_w^+}, \tag{2.13}$$

$$\frac{\partial s}{\partial t} = \frac{\frac{1 + \tanh(k_s(u - u_s))}{2} - s}{\tau_s}. \tag{2.14}$$

As três correntes agregadas são dadas por

$$J_{fi} = -\frac{vH_v(u-\theta_v)(u_u-u)}{\tau_{fi}}, \tag{2.15}$$

$$J_{so} = \frac{(u-u_o)(1-H_w)}{\tau_o} + \frac{H_w}{\tau_{so}}, \tag{2.16}$$

$$J_{si} = -\frac{H_w ws}{\tau_{si}}. \tag{2.17}$$

As funções de chaveamento foram escritas com a função de Heaviside,

$$H(z) = \begin{cases} 0, & z < 0, \\ 1, & z \geq 0, \end{cases} \tag{2.18}$$

com $H_v = H(u-\theta_v)$, $H_w = H(u-\theta_w)$, $H_{v^-} = H(u-\theta_v^-)$ e $H_o = H(u-\theta_o)$. As escalas de tempo auxiliares e os valores de equilíbrio usados nas equações são

$$\tau_v^- = (1-H_{v^-})\tau_{v1}^- + H_{v^-}\tau_{v2}^-, \tag{2.19}$$

$$\tau_w^- = \tau_{w1}^- + (\tau_{w2}^- - \tau_{w1}^-)\frac{1+\tanh(k_w^-(u-u_w^-))}{2}, \tag{2.20}$$

$$\tau_{so} = \tau_{so1} + (\tau_{so2} - \tau_{so1})\frac{1+\tanh(k_{so}(u-u_{so}))}{2}, \tag{2.21}$$

$$\tau_s = (1-H_w)\tau_{s1} + H_w\tau_{s2}, \tag{2.22}$$

$$\tau_o = (1-H_o)\tau_{o1} + H_o\tau_{o2}, \tag{2.23}$$

$$v_\infty = 1 - H_{v^-}, \tag{2.24}$$

$$w_\infty = (1-H_o)\left(1 - \frac{u}{\tau_{w,\infty}}\right) + H_o w_\infty^*. \tag{2.25}$$

A Tabela 2.2 apresenta todos os parâmetros do modelo mínimo e os valores de referência usados neste trabalho, que correspondem ao conjunto epicárdico proposto na formulação original do modelo (BUENO-OROVIO; CHERRY; FENTON, 2008).

Tabela 2.2: Parâmetros epicárdicos do modelo minimal de Bueno-Orovio.

Parâmetro	**Valor**	**Parâmetro**	**Valor**	**Parâmetro**	**Valor**	**Parâmetro**	**Valor**
u_o	0,0	θ_v^-	0,006	τ_{w2}^-	15,0	τ_{o2}	6,0
u_u	1,55	θ_o	0,006	k_w^-	65,0	τ_{so1}	30,0181
θ_v	0,3	τ_{v1}^-	60,0	u_w^-	0,03	τ_{so2}	0,9957
θ_w	0,13	τ_{v2}^-	1150,0	τ_w^+	200,0	k_{so}	2,0458
τ_v^+	1,4506	τ_{w1}^-	60,0	τ_{fi}	0,11	u_{so}	0,65
τ_{s1}	2,7342	τ_{s2}	16,0	k_s	2,0994	u_s	0,9087
τ_{si}	1,8875	$\tau_{w,\infty}$	0,07	w_∞^*	0,94	τ_{o1}	400,0

3 Metodologia

Para alcançar os objetivos propostos, a metodologia foi organizada em etapas que envolvem a implementação do simulador, a verificação da consistência numérica entre backends, o estudo de convergência de malha, a avaliação de desempenho computacional e a extensão para o monodomínio tridimensional com geometria realística. O foco principal da implementação foi avaliar a viabilidade do uso de JAX em simulações de eletrofisiologia cardíaca, especialmente em problemas nos quais o custo computacional cresce com o refinamento da malha. A implementação em NumPy foi usada como referência nos casos em que a execução em CPU ainda era viável, permitindo verificar se a formulação discreta em JAX produzia resultados consistentes.

3.1 Método das Diferenças Finitas

O método das diferenças finitas é uma técnica numérica utilizada para resolver equações diferenciais por meio da aproximação dos operadores diferenciais (HOLMES, 2007). Isso se dá pela discretização do domínio contínuo em um conjunto finito de pontos, substituindo as derivadas por diferenças entre valores vizinhos na malha (LANGTANGEN; LINGE, 2017).

3.1.1 Aproximação dos Operadores

Inicialmente os conceitos serão apresentados para o caso unidimensional para simplificar o tratamento. Sendo assim, o domínio espacial $\Omega = [0, L]$ é dividido em N subintervalos iguais de comprimento $\Delta x = L/N$, o que resulta em $N + 1$ pontos de malha igualmente espaçados, e o tempo é avançado em passos discretos de tamanho Δt, isto é

$$x_i = i\Delta x, \qquad i = 0, 1, \ldots, N, \tag{3.1}$$

$$t_n = n\Delta t, \qquad n = 0, 1, \ldots. \tag{3.2}$$

A derivada temporal é aproximada pela diferença progressiva, em que se parte da condição inicial e se avança no tempo, e que leva a uma atualização explícita do estado:

$$\frac{\partial v}{\partial t} = \frac{v_i^{n+1} - v_i^n}{\Delta t} + O(\Delta t). \tag{3.3}$$

A derivada espacial de segunda ordem é aproximada pela diferença central:

$$\frac{\partial^2 v}{\partial x^2} = \frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta x^2} + O(\Delta x^2). \tag{3.4}$$

Nas duas expressões acima, o termo $O(\Delta t)$ e o termo $O(\Delta x^2)$ representam o erro de truncamento da respectiva aproximação: a parcela desprezada ao substituir a derivada exata pela diferença finita, obtida pela expansão em série de Taylor da função em torno do ponto considerado. A notação indica a ordem com que esse erro diminui à medida que o espaçamento correspondente (Δt ou Δx) é reduzido. As fórmulas centrais no espaço valem nos pontos internos $i = 1, \ldots, N-1$; nos contornos $i = 0$ e $i = N$ aplicam-se as condições de contorno adotadas, apresentadas mais adiante.

Em domínios com mais de uma dimensão espacial, o termo difusivo da equação do monodomínio envolve o operador Laplaciano, que reúne as derivadas de segunda ordem em cada direção. A aproximação por diferença central é aplicada separadamente a cada eixo, somando as contribuições. Cada diferença central mantém a segunda ordem, de modo que o Laplaciano discreto tem erro de truncamento $O(\Delta x^2)$ em cada direção.

No caso bidimensional, a malha passa a ser indexada por (i, j), com espaçamentos Δx e Δy. O Laplaciano discreto é dado por:

$$\nabla^2 v_{i,j}^n \approx \frac{v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n}{\Delta x^2} + \frac{v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n}{\Delta y^2}. \tag{3.5}$$

Quando o espaçamento é uniforme, $\Delta x = \Delta y = h$, a expressão se reduz ao estêncil de cinco pontos:

$$\nabla^2 v_{i,j}^n \approx \frac{v_{i+1,j}^n + v_{i-1,j}^n + v_{i,j+1}^n + v_{i,j-1}^n - 4v_{i,j}^n}{h^2}. \tag{3.6}$$

No caso tridimensional, a malha é indexada por (i, j, k), com espaçamentos Δx, Δy e Δz,

e o Laplaciano discreto acrescenta a contribuição da direção z:

$$\nabla^2 v^n_{i,j,k} \approx \frac{v^n_{i+1,j,k} - 2v^n_{i,j,k} + v^n_{i-1,j,k}}{\Delta x^2} + \frac{v^n_{i,j+1,k} - 2v^n_{i,j,k} + v^n_{i,j-1,k}}{\Delta y^2} + \frac{v^n_{i,j,k+1} - 2v^n_{i,j,k} + v^n_{i,j,k-1}}{\Delta z^2}. \tag{3.7}$$

Para espaçamento uniforme nas três direções, $\Delta x = \Delta y = \Delta z = h$, obtém-se o estêncil de sete pontos:

$$\nabla^2 v^n_{i,j,k} \approx \frac{v^n_{i+1,j,k} + v^n_{i-1,j,k} + v^n_{i,j+1,k} + v^n_{i,j-1,k} + v^n_{i,j,k+1} + v^n_{i,j,k-1} - 6v^n_{i,j,k}}{h^2}. \tag{3.8}$$

Em todos os casos, a atualização explícita no tempo segue a mesma forma da versão unidimensional, substituindo a derivada espacial pelo Laplaciano discreto correspondente à dimensão do problema.

3.2 Esquema Explícito

Esquemas de atualização de modelos em passos de tempo podem utilizar o método explícito ou implícito das diferenças finitas (LANGTANGEN; LINGE, 2017).

No método explícito, o valor de cada ponto no próximo passo de tempo v_i^{n+1} é calculado diretamente a partir de valores já conhecidos no passo atual v^n. Isso torna a implementação simples e o custo computacional por passo baixo, porém impõe uma restrição de estabilidade: o passo de tempo Δt não pode ser grande demais em relação a Δx, caso contrário a simulação diverge (LANGTANGEN; LINGE, 2017).

O método implícito, por sua vez, calcula v_i^{n+1} a partir de outros valores do mesmo passo de tempo $n+1$, que também são desconhecidos. Isso gera um sistema de equações acopladas que deve ser resolvido a cada passo, tornando a implementação mais complexa e o custo computacional maior. Em contrapartida, o método implícito é incondicionalmente estável, permitindo o uso de passos de tempo maiores sem que a simulação divirja.

Neste trabalho, foi utilizado o esquema explícito, cuja fórmula é apresentada a seguir. Utilizando as aproximações apresentadas anteriormente, é possível discretizar a

equação biestável. Substituindo as derivadas pelas diferenças finitas correspondentes:

$$\frac{v_i^{n+1} - v_i^n}{\Delta t} = k\frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta x^2} + f(v_i^n). \tag{3.9}$$

Isolando v_i^{n+1}, obtém-se a fórmula de atualização explícita para o modelo de reação-difusão em uma dimensão para os pontos internos do domínio:

$$v_i^{n+1} = v_i^n + \Delta t \left[k\frac{v_{i+1}^n - 2v_i^n + v_{i-1}^n}{\Delta x^2} + f(v_i^n) \right], \qquad i = 1, \ldots, N-1. \tag{3.10}$$

Para o caso tridimensional, usado na extensão do simulador, aplica-se a mesma construção, substituindo a derivada espacial pelo Laplaciano discreto em três dimensões apresentado anteriormente. Indexando a malha por (i, j, k), com espaçamentos Δx, Δy e Δz, a discretização da equação de reação-difusão fica

$$\begin{aligned} \frac{v_{i,j,k}^{n+1} - v_{i,j,k}^n}{\Delta t} = k\Bigg[& \frac{v_{i+1,j,k}^n - 2v_{i,j,k}^n + v_{i-1,j,k}^n}{\Delta x^2} \\ & + \frac{v_{i,j+1,k}^n - 2v_{i,j,k}^n + v_{i,j-1,k}^n}{\Delta y^2} \\ & + \frac{v_{i,j,k+1}^n - 2v_{i,j,k}^n + v_{i,j,k-1}^n}{\Delta z^2} \Bigg] + f(v_{i,j,k}^n). \end{aligned} \tag{3.11}$$

Isolando $v_{i,j,k}^{n+1}$, obtém-se a fórmula de atualização explícita para os pontos internos do domínio tridimensional:

$$\begin{aligned} v_{i,j,k}^{n+1} = v_{i,j,k}^n + \Delta t \Bigg[k\Bigg(& \frac{v_{i+1,j,k}^n - 2v_{i,j,k}^n + v_{i-1,j,k}^n}{\Delta x^2} \\ & + \frac{v_{i,j+1,k}^n - 2v_{i,j,k}^n + v_{i,j-1,k}^n}{\Delta y^2} \\ & + \frac{v_{i,j,k+1}^n - 2v_{i,j,k}^n + v_{i,j,k-1}^n}{\Delta z^2} \Bigg) + f(v_{i,j,k}^n) \Bigg]. \end{aligned} \tag{3.12}$$

3.2.1 Condições de Contorno

Condição de fluxo nulo

A condição $v_x = 0$ nas extremidades do domínio unidimensional significa que não há fluxo de corrente pelas pontas. Aplicando essa condição na malha discreta, obtêm-se fórmulas

de atualização modificadas para os pontos $i = 0$ e $i = N$:

$$v_0^{n+1} = v_0^n + \Delta t \left[k \frac{2(v_1^n - v_0^n)}{\Delta x^2} + f(v_0^n) \right], \tag{3.13}$$

$$v_N^{n+1} = v_N^n + \Delta t \left[k \frac{2(v_{N-1}^n - v_N^n)}{\Delta x^2} + f(v_N^n) \right]. \tag{3.14}$$

Em duas e três dimensões, a mesma ideia é aplicada em cada direção separadamente e será discutida em mais detalhes na Seção 3.9.2.

Condição periódica

Ao impor a condição $v_0^n = v_N^n$ para todos os passos de tempo, a fibra passa a se comportar como um anel, com o ponto inicial conectado diretamente ao ponto final. Isso faz com que a onda elétrica nunca encontre uma extremidade e continue se propagando indefinidamente. Esta condição é útil para representar uma situação de reentrada no domínio unidimensional.

3.2.2 Condição de Estabilidade

O esquema explícito é apenas condicionalmente estável: o passo de tempo precisa respeitar um limite associado à discretização do termo difusivo, caso contrário a solução diverge. A análise de estabilidade de von Neumann aplicada ao termo de difusão discretizado por diferença central fornece a condição geral, dada por (LANGTANGEN; LINGE, 2017):

$$F_\Sigma = \Delta t \sum_i \frac{k}{\Delta x_i^2} \leq \frac{1}{2}, \tag{3.15}$$

em que F_Σ é o número de Fourier somado sobre as direções espaciais, k o coeficiente de difusão e Δx_i o espaçamento na direção i.

Para uma malha isotrópica, em que $\Delta x_i = h$ é igual nas d direções, a soma se reduz a $d\,k\,\Delta t/h^2$, ou seja $F_\Sigma = d\,F$, e a condição (3.15) assume a forma do número de Fourier por eixo

$$F = \frac{k\,\Delta t}{h^2} \leq \frac{1}{2d}, \tag{3.16}$$

cujo limite vale $1/2$ em uma dimensão, $1/4$ em duas dimensões e $1/6$ em três dimensões.

Isolando o passo de tempo no caso isotrópico, $\Delta t \leq h^2/(2dk)$, de modo que o passo máximo estável decai com o quadrado do espaçamento à medida que a malha é refinada.

Na implementação, o passo de tempo é calculado diretamente a partir da forma geral (3.15), fixando F_Σ em um valor de segurança abaixo de 1/2:

$$\Delta t = \frac{F_\Sigma}{\dfrac{k}{\Delta x^2} + \dfrac{k}{\Delta y^2} + \dfrac{k}{\Delta z^2}}, \qquad F_\Sigma = 0{,}45, \tag{3.17}$$

o que mantém $F_\Sigma = 0{,}45 \leq 1/2$. Para uma malha cúbica uniforme isso corresponde a $F_\Sigma = 3F$, com número de Fourier por eixo $F = 0{,}15$, abaixo do limite de $1/6 \approx 0{,}1667$ do caso isotrópico tridimensional; em malha anisotrópica, contudo, o critério que de fato governa a estabilidade é a soma F_Σ, e não o valor por eixo. Nos estudos bidimensional e unidimensional, em que a malha é isotrópica, o passo é fixado de forma equivalente pelo número de Fourier F, informado na descrição de cada estudo.

Para que o número de Fourier permaneça constante ao longo de um estudo de refinamento, o passo de tempo não é fixado de forma independente, mas recalculado a cada malha a partir do espaçamento. Ao reduzir Δx, o passo diminui proporcionalmente a Δx^2, e o número de passos para cobrir um mesmo tempo físico cresce na mesma proporção. Essa relação $\Delta t \propto \Delta x^2$ é também o motivo pelo qual a ordem de convergência observada se aproxima de dois mesmo com avanço temporal de primeira ordem, pois o erro temporal $O(\Delta t)$ passa a decair como $O(\Delta x^2)$, na mesma taxa do erro espacial.

Além da restrição difusiva, foi imposto um teto independente $\Delta t \leq \Delta t_{\max}$, com $\Delta t_{\max} = 0{,}05$ ms, associado à estabilidade do termo de reação dos modelos iônicos mais rígidos, cuja dinâmica local pode exigir um passo menor que o ditado apenas pela difusão. Nas malhas mais grossas, em que o passo difusivo excederia esse teto, o passo de tempo é limitado por $\Delta t_{\max}$; nesses casos o número de Fourier efetivo fica abaixo do valor alvo e a relação $\Delta t \propto \Delta x^2$ deixa de valer enquanto o teto estiver ativo.

3.2.3 Ordem de Convergência

A solução numérica de uma equação diferencial aproximada por diferenças finitas contém um erro de discretização, que depende do espaçamento da malha. A ordem de con-

vergência mede a taxa com que esse erro diminui à medida que a malha é refinada.

Em geral, para um método de ordem p, o erro E se relaciona com o espaçamento h por:

$$E \approx C\, h^p, \tag{3.18}$$

em que C é uma constante que não depende de h. Quanto maior o valor de p, mais rápido o erro decresce conforme a malha é refinada (HOLMES, 2007; ROY, 2005).

Na prática, o valor de p pode ser estimado a partir dos erros obtidos em malhas com espaçamentos diferentes. Considerando duas malhas com espaçamentos h_1 (mais grossa) e h_2 (mais fina), relacionadas por uma razão de refinamento $r = h_1/h_2$, e seus respectivos erros E_1 e E_2, a ordem de convergência observada é dada por:

$$p = \frac{\ln(E_1/E_2)}{\ln r}. \tag{3.19}$$

A validade dessa estimativa depende de a malha estar suficientemente refinada para que o termo de menor ordem do erro domine, condição conhecida como faixa assintótica de convergência. Em malhas grossas, fora dessa faixa, a ordem observada pode se afastar da ordem teórica do esquema. Para a aproximação espacial por diferença central de segunda ordem, como a aplicada neste trabalho, espera-se $p \approx 2$ dentro da faixa assintótica (HOLMES, 2007).

Neste trabalho, na ausência de uma solução exata, o erro E foi calculado em relação à solução obtida na malha mais refinada, o que fornece uma estimativa a posteriori da ordem de convergência, na linha da extrapolação de Richardson (ROY, 2005; NASA Glenn Research Center, 2021). A concordância entre a ordem observada e a ordem esperada é tomada como indício de consistência da implementação com o comportamento previsto pelo esquema numérico.

3.3 Ferramentas Computacionais

Definido o esquema numérico de atualização, é necessário escolher a ferramenta computacional usada para implementá-lo. Essa escolha é particularmente relevante neste trabalho,

já que um dos objetivos centrais é avaliar o ganho de desempenho obtido ao migrar a simulação de um ambiente tradicional baseado em CPU para um ambiente com execução acelerada em GPU. Por isso, antes de descrever a implementação do simulador propriamente dita, esta seção apresenta as duas bibliotecas Python utilizadas como backends de cálculo: NumPy, usada como referência em CPU, e JAX, usada como backend principal voltado à aceleração das simulações.

NumPy é uma biblioteca tradicional para computação numérica baseada em arrays, usada em Python para operações vetorizadas sobre dados multidimensionais executadas em CPU (HARRIS et al., 2020). Neste trabalho, a implementação em NumPy serviu como referência de comparação numérica e baseline de desempenho. Essa implementação executa em uma única thread de CPU. As operações de NumPy atuam sobre arrays inteiros, mas o laço temporal é sequencial e as rotinas empregadas não distribuem o cálculo entre múltiplos núcleos. Assim, o baseline representa um uso comum de NumPy, abaixo do desempenho que uma implementação otimizada em CPU poderia alcançar. Ao longo deste texto, o termo vetorizado se refere à expressão do cálculo como operações sobre arrays inteiros, delegadas a rotinas compiladas da biblioteca. O uso explícito de instruções SIMD do processador está fora desse escopo. O aproveitamento dessas unidades, quando ocorre, é decidido internamente pelas rotinas da biblioteca e pelo compilador, sem controle direto no código.

O JAX é uma biblioteca para computação numérica de alto desempenho voltada à computação sobre arrays e à transformação de programas, com execução em aceleradores, criada com foco em desempenho e em aprendizado de máquina em larga escala (BRADBURY et al., 2018). Ele combina uma interface de programação semelhante à de NumPy com transformações de função componíveis, como diferenciação automática, compilação Just-in-Time e vetorização automática. Essas funções são compiladas para a representação intermediária do XLA, que gera código otimizado para execução em CPU, GPU e TPU. Neste trabalho, não foram utilizadas a diferenciação automática nem a vetorização automática; o uso do JAX se concentrou na compilação Just-in-Time com XLA e na interface de arrays semelhante à de NumPy executada em GPU, recursos diretamente voltados à aceleração das simulações de diferenças finitas. Essa combinação de

interface familiar e execução acelerada motivou a escolha do JAX como backend principal do simulador.

O simulador foi desenvolvido a partir de um esquema explícito de diferenças finitas, com cálculo do termo difusivo por aproximações do Laplaciano, aplicação das condições de contorno e avaliação dos termos locais dos modelos iônicos. A implementação principal foi construída em JAX, cuja proposta e motivação foram apresentadas na Seção 3.3, usando uma estrutura de código semelhante à de NumPy adaptada ao seu modelo de execução (BRADBURY et al., 2018; JAX, 2026g).

3.3.1 Modelo de Programação em JAX

O modelo de programação em JAX se organiza em torno de um mecanismo central: a compilação por rastreamento, conhecida como *tracing*. Ao aplicar `jax.jit` a uma função, o JAX a executa uma vez com argumentos abstratos, que carregam apenas a forma e o tipo dos arrays, sem os valores concretos, e registra as operações aplicadas sobre esses argumentos para gerar uma versão compilada (JAX, 2026e). Para que esse rastreamento funcione, a função precisa ser pura, dependendo apenas de suas entradas e sem efeitos colaterais (JAX, 2026e). As três características de programação descritas a seguir decorrem diretamente desse modelo.

Funções. Em NumPy, uma função executa imediatamente sobre os valores fornecidos. Em JAX, `jax.jit` compila a função antes da execução repetida: a primeira chamada inclui o custo de rastreamento e compilação, e as chamadas seguintes, com formas e tipos compatíveis, reutilizam a versão compilada (JAX, 2026e). Quando a forma ou o tipo dos argumentos muda, a função é compilada novamente. Neste trabalho, `jax.jit` foi aplicado a duas rotinas com papéis distintos. A primeira é a função de passo, que recebe o estado atual do sistema e devolve o estado no instante seguinte. Ela monta o estêncil espacial e calcula o Laplaciano discreto, avalia o termo de reação e o termo de porta do modelo iônico no estado corrente, aplica a máscara e as condições de contorno, e retorna as variáveis atualizadas. A segunda é a rotina de avanço, que encadeia um número fixo de execuções dessa função de passo, propagando o estado de uma iteração para a seguinte sem retornar ao interpretador Python e sem materializar os estados intermediários. Com

isso, o cálculo do Laplaciano, a avaliação dos termos iônicos e a atualização das variáveis ocorrem inteiramente dentro da parte compilada, e o laço temporal completo é entregue ao compilador como uma única unidade.

Vetores. Os arrays de `jax.numpy` têm interface próxima à de NumPy, porém são imutáveis e não permitem atribuição direta por índice (JAX, 2026g). A atualização no estilo `v[i] = x`, válida em NumPy, foi substituída por `v = v.at[i].set(x)`, que retorna um novo array sem modificar o original. Dentro de código compilado com `jax.jit`, quando o array original não é reutilizado, o compilador realiza a atualização no próprio espaço de memória, evitando custo adicional. As operações sobre a malha foram escritas de forma vetorizada, usando recursos como `jnp.pad` e `jnp.roll` para montar o estêncil espacial.

Condicionais. Como os valores não estão disponíveis durante o rastreamento, uma decisão do tipo `if v[i] > a` sobre os valores de um array não pode ser usada dentro da função compilada, pois os valores rastreados só afetam o fluxo de controle por meio de atributos estáticos, como forma e tipo (JAX, 2026e). A escolha condicional sobre a malha foi escrita com `jnp.where(cond, a, b)`, que avalia as duas alternativas e seleciona ponto a ponto.

Loops. Um laço `for` em Python sobre milhares de passos seria desenrolado durante o rastreamento, gerando um grafo grande e uma compilação lenta. Por isso o avanço no tempo foi estruturado com `jax.lax.scan`, que representa uma repetição com estado acumulado: a cada passo recebe o estado atual, calcula o novo estado e o repassa para a iteração seguinte (JAX, 2026d). Assim, o potencial e as variáveis de recuperação são carregados ao longo do tempo dentro de uma repetição compatível com a compilação, sem executar o laço principal como um loop Python.

Em resumo, a atualização funcional com `.at[...].set(...)`, a seleção com `jnp.where` e a repetição com `jax.lax.scan` são as formas correspondentes a atribuição direta, condicional e laço quando o código precisa ser puro e compilável. A Listagem 3.1 apresenta um exemplo de código com essas formas para o simulador do modelo de FitzHugh-Nagumo em duas dimensões.

```python
import jax
import jax.numpy as jnp

v = jnp.zeros((N + 1, N + 1)).at[:N // 10, :].set(0.3)
w = jnp.zeros((N + 1, N + 1)) # condições iniciais

@jax.jit  # função compilada por rastreamento (tracing)
def passo(v, w):
    vp = jnp.pad(v, 1, mode="reflect")  # cond. Neumann
    lap = (vp[2:, 1:-1] - 2*vp[1:-1, 1:-1] + vp[:-2, 1:-1]
        + vp[1:-1, 2:] - 2*vp[1:-1, 1:-1] + vp[1:-1, :-2]) / dx**2
    # incrementos avaliados no estado do tempo n (v^n, w^n)
    reac = A * v * (1 - v) * (v - alpha) - w
    dw   = eps * (v - gamma * w)
    # atualizacao simultanea para o tempo n+1
    v_novo = v + dt * (k * lap + reac)
    w_novo = w + dt * dw
    v_novo = jnp.where(ativo, v_novo, 0.0)  # condicional
       vetorizada (mask)
    return v_novo, w_novo

# Avanço no tempo com lax.scan, sem laço Python explícito
def avanco(v, w, npassos):
    corpo = lambda estado, _: (passo(*estado), None)
    (v, w), _ = jax.lax.scan(corpo, (v, w), None, length=npassos)
    return v, w
```

Listagem 3.1: Passo de tempo e avanço temporal do simulador em JAX, reunindo a função compilada, o estêncil vetorizado, a seleção condicional e o laço com `lax.scan`.

A listagem torna explícito um detalhe do esquema numérico: os incrementos das duas variáveis são avaliados no mesmo estado, correspondente ao tempo n, e só depois aplicados. A variável de recuperação toma como entrada o potencial no tempo n; o valor atualizado v^{n+1} só passa a ser usado no passo seguinte. Trata-se, portanto, de uma atualização simultânea das variáveis de estado. Essa ordem é preservada de forma idêntica nas implementações em NumPy e em JAX, condição necessária para que os dois backends produzam a mesma sequência de estados e para que a comparação numérica entre eles seja significativa.

3.3.2 Backends e Precisão Numérica

A seleção entre CPU e GPU foi feita pelo dispositivo disponível no ambiente de execução, mantendo a mesma formulação numérica. A implementação em JAX em GPU usou a mesma formulação da versão em JAX CPU, alterando apenas o dispositivo de execução.

Nos casos unidimensionais, as três configurações, NumPy em CPU, JAX em CPU e JAX em GPU, foram comparadas, o que permite separar os ganhos associados aos mecanismos de otimização do JAX dos ganhos associados ao paralelismo massivo da GPU. O backend de CPU do JAX, ao contrário da referência em NumPy, distribui a execução entre os núcleos disponíveis por padrão, de modo que a comparação entre NumPy CPU e JAX CPU contrapõe uma execução sequencial a uma execução paralela em CPU. Nos casos bidimensionais, a comparação de desempenho foi feita entre NumPy em CPU e JAX em GPU, com foco no ganho total da execução em GPU (BRADBURY et al., 2018; JAX, 2026b; JAX, 2026g). A execução em precisão dupla foi ativada com `jax_enable_x64`, configuração que habilita o uso de arrays em 64 bits no JAX. A precisão simples foi mantida nos benchmarks em que o objetivo era avaliar desempenho, enquanto a dupla precisão foi usada nos testes de consistência e nos estudos de convergência em que a comparação numérica era mais sensível.

A implementação em NumPy foi mantida como uma versão de referência em CPU. Seu papel principal foi auxiliar na verificação da consistência numérica da implementação em JAX nos casos 1D e 2D em que o custo computacional ainda permitia a execução em ambos os backends. Essa comparação não foi repetida no caso 3D com máscara anatômica, pois a execução em NumPy se tornaria muito custosa e a equivalência entre as formulações já havia sido avaliada nos casos de menor custo.

3.3.3 Adaptações da Implementação em JAX

Nos casos unidimensionais, a formulação em JAX manteve o mesmo esquema explícito usado na versão NumPy, mas as atualizações da malha foram escritas em estilo funcional. Para isso, foram usadas operações do tipo `.at[...].set(...)`, que retornam um novo array a cada passo, tanto para aplicar o estímulo inicial quanto para atualizar pontos específicos, como as extremidades no caso de Neumann. Nos casos periódicos, os índices dos vizinhos foram construídos com arrays de índices, permitindo representar a ligação entre as extremidades dentro da função compilada.

Nos casos bidimensionais, a principal adaptação foi expressar o cálculo do Laplaciano e das condições de contorno por operações vetorizadas compatíveis com JIT. Para

fronteiras de Neumann, foi usado preenchimento refletido da malha com `jnp.pad`; para fronteiras periódicas, foram usados deslocamentos circulares com `jnp.roll`. No protocolo S1-S2, descrito em detalhe na Seção 3.5, em que um primeiro estímulo (S1) gera uma frente de onda e um segundo estímulo (S2) é aplicado enquanto parte do tecido ainda se encontra refratária, de modo a induzir reentrada, o segundo estímulo foi incorporado ao avanço temporal com um vetor de gatilho que indica o instante de aplicação do S2. A operação `jnp.where` foi usada para aplicar esse estímulo apenas no passo correspondente, mantendo o estado inalterado nos demais passos.

Nos modelos com diferentes números de variáveis, o padrão de implementação foi mantido alterando apenas o estado carregado pelo avanço temporal. Nos modelos biestável, FitzHugh-Nagumo e Mitchell-Schaeffer, o estado possui duas variáveis, como (v, w) ou (v, h). No modelo Bueno-Orovio, o estado passa a carregar quatro variáveis, (u, v, w, s). Em todos os casos, o avanço temporal foi escrito como uma repetição sobre esse estado: a função de passo calcula as novas variáveis, e o estado atualizado é usado como entrada do passo seguinte.

No caso tridimensional com máscara anatômica, a adaptação principal foi representar o domínio mascarado por arrays tridimensionais. A máscara ativa foi convertida para um array booleano de JAX com `jnp.asarray`, e as atualizações foram restritas aos pontos ativos por meio de `jnp.where`. No cálculo do Laplaciano mascarado, vizinhos fora da máscara foram substituídos pelo valor do ponto central, mantendo a condição sem fluxo na fronteira irregular da geometria. Assim, a mesma ideia de condição de Neumann foi estendida da borda regular da caixa para a fronteira interna definida pela máscara.

Nas medições de desempenho, a implementação também precisou considerar o modo de execução assíncrono do JAX. Antes da coleta dos tempos, foi realizada uma execução de aquecimento para acionar a compilação JIT. Depois, as execuções temporizadas foram sincronizadas com `block_until_ready`, garantindo que o tempo registrado correspondesse ao cálculo efetivamente concluído, e não apenas ao envio das operações para o dispositivo (JAX, 2026b; JAX, 2026a). O custo de cada uma dessas etapas, incluindo as transferências de dados entre CPU e GPU, é quantificado na Tabela 4.15.

3.4 Cenários Avaliados

Os experimentos consideraram os modelos biestável, FitzHugh-Nagumo, Mitchell-Schaeffer e Bueno-Orovio. Os cenários avaliados incluíram propagação unidimensional, propagação bidimensional, condições periódicas, protocolos S1-S2 para geração de espirais e uma extensão tridimensional do monodomínio, avaliada tanto em uma caixa cartesiana completa, no estudo de convergência de malha, quanto em uma geometria mascarada a partir de uma superfície STL.

Os casos 1D e 2D foram usados principalmente para avaliar consistência numérica, desempenho computacional e convergência de malha em cenários controlados. O caso 3D estendeu o simulador a três dimensões em duas etapas: uma caixa cartesiana completa, usada no estudo de convergência de malha no mesmo modelo e domínio do caso anatômico, e uma geometria mais complexa, definida por uma máscara ativa construída a partir de uma superfície STL. Ambas mantiveram a formulação numérica baseada em diferenças finitas.

3.5 Definição dos Problemas

Neste trabalho, os tempos são reportados em milissegundos (ms) e as variáveis de potencial são tratadas em forma normalizada em todos os modelos. As escalas de espaço e o coeficiente de difusão, por sua vez, dependem da natureza de cada modelo. Os modelos biestável e FitzHugh-Nagumo são fenomenológicos e não possuem escala espacial física intrínseca, de modo que seus domínios são expressos em unidades espaciais do modelo e o coeficiente de difusão k é adimensional. O modelo Mitchell-Schaeffer usa constantes de tempo em milissegundos, mas mantém o domínio espacial em unidades do modelo. O modelo de Bueno-Orovio é calibrado para reproduzir a eletrofisiologia do tecido ventricular humano (BUENO-OROVIO; CHERRY; FENTON, 2008) e, por esse motivo, é definido em unidades físicas, com domínio em centímetros e coeficiente de difusão em cm^2/ms.

Nos problemas de propagação unidimensional e em domínio bidimensional retangular, foi adotada condição de Neumann homogênea nas bordas, representando ausência de fluxo pela fronteira do tecido (SUNDNES et al., 2006). Em uma dimensão, isso cor-

responde a $\partial v/\partial x = 0$ nas extremidades do domínio; em duas dimensões, corresponde a $\nabla v \cdot \mathbf{n} = 0$ em $\partial\Omega$, onde $\mathbf{n}$ é o vetor normal externo. Para o modelo mínimo de Bueno-Orovio, a mesma condição é aplicada à variável de potencial u.

As condições periódicas foram usadas apenas nos casos explicitamente definidos como periódicos, nos quais as bordas opostas do domínio são conectadas para representar uma fibra ou superfície fechada. Dessa forma, as tabelas de parâmetros se concentram em discretização, modelo iônico e condição inicial, enquanto as condições de contorno são descritas nesta definição do problema.

Na implementação, as condições de Neumann foram representadas por reflexão dos valores na fronteira, equivalente ao uso de pontos fantasmas no cálculo por diferenças finitas. Nos casos periódicos, os vizinhos das bordas foram obtidos por índices ou deslocamentos circulares, conectando numericamente uma extremidade à outra.

O protocolo S1-S2 foi usado para a formação de espirais no modelo FitzHugh-Nagumo: um primeiro estímulo (S1) em uma faixa do domínio e, em um instante alvo, um segundo estímulo (S2) em parte do domínio, de modo a induzir a reentrada. A configuração específica desse experimento (domínio, instantes, parâmetros do modelo e discretização por malha) é apresentada junto do resultado correspondente, no Capítulo 4.

3.6 Verificação entre Backends

A comparação direta entre NumPy e JAX foi usada como uma etapa inicial de verificação da implementação. Seu objetivo foi confirmar que a transposição da formulação discreta para JAX preservava a mesma solução numérica obtida pela versão NumPy quando os dois backends eram executados com a mesma malha, os mesmos parâmetros e as mesmas condições iniciais.

Roy discute a verificação de código como um conjunto de procedimentos voltados a identificar erros de implementação que afetam a discretização numérica, com testes formais baseados, por exemplo, em soluções manufaturadas e ordem de acurácia (ROY, 2005). Neste trabalho, a comparação entre NumPy e JAX teve escopo mais delimitado: medir o desvio entre duas implementações da mesma formulação discreta em casos controlados. Os desvios foram reportados por erro máximo absoluto e erro relativo em norma

L_2, usando a implementação NumPy em dupla precisão como referência.

Essa verificação foi concentrada nos casos 2D representativos, usando a maior malha para a qual a execução NumPy CPU foi mantida como baseline. A partir dessa etapa, a implementação JAX GPU foi adotada como backend principal nas análises de convergência, nos benchmarks mais refinados e na extensão tridimensional, reduzindo a necessidade de repetir a mesma comparação direta contra NumPy em todos os cenários posteriores.

3.7 Convergência de Malha

Além da comparação entre backends, foi realizado um estudo de convergência de malha para estimar o efeito do refinamento sobre a solução numérica. Nesse estudo, o domínio físico foi mantido fixo e o número de pontos da malha foi aumentado, reduzindo Δx. Como a grandeza avaliada é o erro da própria solução, o passo de tempo respeitou o limite de estabilidade, com o número de Fourier de malha mantido em $F = k\Delta t/\Delta x^2 = 0{,}2$, abaixo do teto de $1/4$ em duas dimensões. Nos modelos com reação mais rígida, um teto adicional de Δt foi aplicado. No estudo de convergência tridimensional, em que os espaçamentos diferem entre as direções, a condição de estabilidade equivalente é a soma sobre as três direções $i \in \{x, y, z\}$, $\sum_i k\,\Delta t/\Delta x_i^2 \leq 1/2$; adotou-se o valor 0,45 para essa soma, abaixo do limite, com o mesmo teto adicional de $\Delta t \leq 0{,}05$ ms.

Como o objetivo do estudo era avaliar a variação da solução com o refinamento da malha, a solução mais refinada disponível foi adotada como referência numérica. Essa escolha permite comparar as malhas mais grossas com uma solução calculada no mesmo modelo e no mesmo domínio, mas com menor espaçamento espacial. Para isso, a malha mais refinada executada em JAX GPU com dupla precisão foi usada como referência, e as soluções obtidas em malhas mais grossas foram comparadas com essa referência nos mesmos pontos físicos do domínio.

O erro de cada malha foi quantificado por uma norma L_2 relativa, calculada sobre

todos os pontos da malha em um único instante, o tempo final da simulação:

$$E = \frac{\|u - u_{\text{ref}}\|_2}{\|u_{\text{ref}}\|_2}, \tag{3.20}$$

em que u é a solução avaliada na malha em teste, u_{ref} é a solução de referência amostrada nos mesmos pontos físicos e $\|\cdot\|_2$ denota a norma euclidiana espacial. Assim, o erro corresponde ao desvio do campo inteiro em relação à referência em um instante específico, e não a um acúmulo ao longo dos passos de tempo. Esse valor de E alimenta a estimativa da ordem de convergência apresentada na Seção 3.2.3.

3.8 Benchmarks de Desempenho

Os benchmarks compararam o tempo de execução entre os backends avaliados. Nos casos em que a execução NumPy estava disponível, também foi calculada a aceleração relativa da implementação JAX em relação à referência em CPU.

Nos benchmarks bidimensionais, o número de Fourier de malha foi mantido fixo em $F = k\Delta t/\Delta x^2 = 0{,}2$ entre as diferentes malhas, abaixo do limite de estabilidade difusiva em duas dimensões, $1/4$. Fixar F mantém comparável a carga de trabalho entre as resoluções, já que o passo de tempo acompanha Δx^2 e o tempo físico simulado permanece o mesmo em todas as malhas. O mesmo valor de F foi adotado no estudo de convergência, de modo que as duas análises usam discretizações temporais consistentes e dentro do limite de estabilidade.

A metodologia de medição seguiu as recomendações oficiais de benchmarking do JAX, que destacam três cuidados principais: separar o custo de compilação da execução repetida, sincronizar explicitamente o dispositivo antes de registrar o tempo e controlar a precisão numérica usada na comparação (JAX, 2026b). Por isso, antes da coleta dos tempos, foi realizada uma etapa de aquecimento para acionar a compilação JIT. Além disso, as execuções temporizadas foram sincronizadas com `block_until_ready`, evitando que o tempo medido representasse apenas o envio assíncrono das operações para o dispositivo. O tempo reportado em cada caso corresponde à média de cinco execuções cronometradas após o aquecimento, antecedidas de uma carga de trabalho auxiliar na GPU para

estabilizar a frequência de operação, de modo que a medida não dependa da ordem de execução nem do estado ocioso do dispositivo. Além da média, foram registrados os tempos mínimo e máximo observados entre as repetições, usados para indicar a dispersão das medidas. Nas execuções em NumPy de maior custo, em que uma única repetição já se estende por dezenas de minutos, o tempo reportado corresponde a uma única execução tomada como referência, e a dispersão não é informada; como essas execuções são longas e limitadas pela CPU, a variabilidade relativa esperada é pequena diante da diferença de ordem de grandeza observada entre os backends. A cronometragem foi feita com `time.perf_counter` em torno das chamadas já compiladas, com sincronização explícita do dispositivo, e a separação por etapas foi obtida instrumentando diretamente as fases de transferência, rastreamento, compilação e execução. Não foi empregado um perfilador dedicado, como o JAX Profiler, que permitiria atribuir custo a operações individuais dentro do laço compilado; essa análise mais fina é apontada como desdobramento futuro.

Vale observar que, nas malhas menores, o tempo total é dominado por custos fixos de execução, como o lançamento das operações na GPU, e não pelo cálculo numérico em si. Nesse regime, a diferença de tempo entre precisão simples e dupla é da ordem da própria variação entre execuções, de modo que a comparação entre as duas precisões só se torna significativa nas malhas maiores, em que o custo de cálculo domina o tempo total.

3.9 Extensão para Monodomínio 3D

Para a simulação do monodomínio em três dimensões, foi utilizada uma máscara anatômica definida sobre um domínio cartesiano maior, em formato de caixa. Nesse domínio, cada ponto da malha tridimensional representa uma posição discreta do tecido, separada dos pontos vizinhos pelos espaçamentos Δx, Δy e Δz. A utilização de uma máscara anatômica sobre uma malha cartesiana segue a ideia geral de representar geometrias cardíacas complexas em um domínio computacional discreto.

A geometria anatômica foi obtida a partir de um arquivo STL, que descreve a superfície do domínio por meio de uma malha de triângulos. Essa geometria corresponde a um modelo biventricular paciente-específico, construído a partir de imagens de ressonância magnética obtidas no Hospital Universitário da UFJF (HU-UFJF) e processadas pelo

pipeline MyoMesh, desenvolvido no laboratório FISIOCOMP da UFJF (NAMORATO et al., 2025). A partir dessa superfície, foi construída uma máscara booleana sobre a malha cartesiana. Assim, cada ponto da malha é classificado como ativo ou inativo, dependendo de estar dentro ou fora da geometria descrita pelo STL.

O arquivo STL descreve apenas a superfície do domínio por meio de faces triangulares, e não corresponde diretamente à malha numérica usada pelo simulador. Por isso, o STL foi tratado como uma descrição geométrica da forma anatômica, enquanto a simulação continuou sendo feita em uma grade cartesiana regular. Para isso, foi criado um conversor que utiliza a superfície descrita pelo STL para construir uma máscara booleana sobre uma malha cartesiana regular, com pontos distribuídos nas direções x, y e z.

Os dados da simulação são armazenados em matrizes tridimensionais definidas sobre toda a caixa (*bounding-box*), mas a evolução do potencial elétrico é calculada apenas nos pontos ativos da máscara. Os pontos fora da máscara não representam tecido cardíaco e, por isso, não participam da propagação. A Figura 3.1 resume as etapas desse tratamento no qual a superfície do domínio de interesse é descrita por um arquivo STL que é usado para definir uma máscara booleana sobre uma caixa cartesiana regular. A fronteira irregular resultante é tratada como uma borda de fluxo nulo, descrita mais adiante.

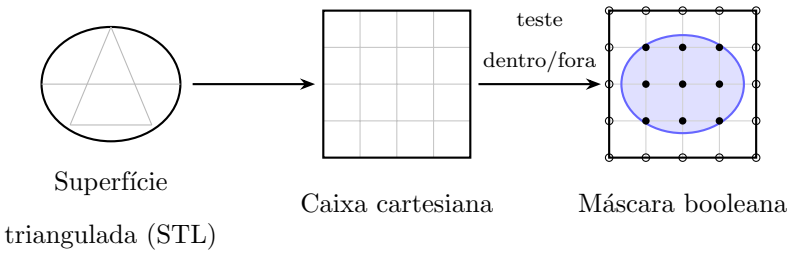

Figura 3.1: Etapas do tratamento da geometria anatômica em 3D.

3.9.1 Geração da Máscara Ativa

A partir dos limites espaciais da superfície STL, foi construída uma caixa tridimensional envolvendo toda a geometria. Essa caixa foi discretizada em pontos igualmente espaçados. Em seguida, cada ponto da grade foi classificado como interno ou externo à superfície usando a rotina `vtkSelectEnclosedPoints`, da biblioteca VTK. Essa função avalia um conjunto de pontos em relação a uma superfície fechada e produz uma máscara binária, na

qual os pontos externos recebem valor 0 e os pontos internos recebem valor 1. Com isso, a superfície STL passa a definir quais pontos da grade cartesiana pertencem ao domínio anatômico. Os pontos classificados como internos formaram a máscara ativa da simulação, enquanto os pontos externos permaneceram inativos.

Como essa classificação depende da definição de interior e exterior da superfície, assume-se que o STL utilizado descreve uma superfície fechada.

Na implementação, a solução numérica é calculada nos vértices da grade cartesiana, que são os pontos onde as variáveis do modelo iônico são efetivamente atualizadas ao longo do tempo. Além dessa malha de cálculo, foi construída uma segunda máscara, definida por célula, usada exclusivamente para a visualização do domínio e que não participa do cálculo da solução. Seu papel é apenas decidir quais blocos (células ou *voxels*) são desenhados como tecido ativo na representação gráfica da malha, e não armazenar ou mostrar valores do potencial. Para isso, cada célula é classificada como ativa ou inativa a partir da posição do seu centro em relação à superfície STL. Dessa forma, os vértices definem onde a simulação é calculada, enquanto os centros das células definem apenas quais blocos aparecem ao desenhar a geometria da malha ativa.

3.9.2 Tratamento da Fronteira para o Caso 3D

Na fronteira da máscara que define o tecido ativo, alguns vizinhos de um ponto ativo podem estar fora do tecido. Nesses casos, os pontos externos não são tratados como tecido ativo. O tratamento para impor uma condição de fluxo nulo na fronteira, é semelhante à condição de Neumann homogênea usada em problemas de reação-difusão. No contexto do monodomínio cardíaco, essa condição representa a hipótese de que o tecido está eletricamente isolado na fronteira do domínio (SUNDNES et al., 2006; NIEDERER et al., 2011).

Em termos contínuos, a condição de fluxo nulo pode ser escrita como

$$\nabla v \cdot \mathbf{n} = 0 \quad \text{em } \partial\Omega, \tag{3.21}$$

onde v representa o potencial transmembrânico, $\mathbf{n}$ é o vetor normal externo à fronteira e

$\partial\Omega$ representa a fronteira do domínio.

Na implementação numérica, essa condição foi tratada usando a ideia de pontos ou células fantasma, uma estratégia comum para impor condições de Neumann em métodos de diferenças finitas e volumes finitos (DANIEL, 2020; LANGTANGEN; LINGE, 2017). Em uma dimensão, por exemplo, se v_i é um ponto ativo e v_{i+1} estaria fora da máscara, a condição de fluxo nulo pode ser aproximada por

$$\frac{v_{i+1} - v_i}{\Delta x} = 0, \Rightarrow v_{i+1} = v_i. \tag{3.22}$$

Assim, o ponto externo não recebe um novo valor físico. Ele apenas assume o valor do ponto ativo adjacente para que não exista diferença de potencial na direção normal à fronteira. Sendo assim, quando o ponto $i + 1$ está fora da máscara, usa-se $v_{i+1} = v_i$, resultando em

$$\frac{v_i - 2v_i + v_{i-1}}{\Delta x^2} = \frac{v_{i-1} - v_i}{\Delta x^2}. \tag{3.23}$$

No caso tridimensional, o mesmo procedimento é aplicado separadamente nas direções x, y e z. Quando um vizinho pertence à máscara ativa, seu valor é usado normalmente no cálculo do Laplaciano. Quando o vizinho está fora da máscara, usa-se o valor do ponto central. Dessa forma, a difusão ocorre apenas entre pontos ativos da geometria, enquanto a fronteira irregular da máscara se comporta como uma fronteira sem fluxo.

A Figura 3.2 ilustra esse tratamento sobre a malha cartesiana. Os pontos internos à máscara formam o tecido ativo, e a fronteira irregular separa esses pontos dos pontos externos. Quando um ponto ativo tem um vizinho externo, esse vizinho funciona como ponto fantasma e recebe o valor do próprio ponto ativo, de modo que não exista fluxo na direção normal à fronteira.

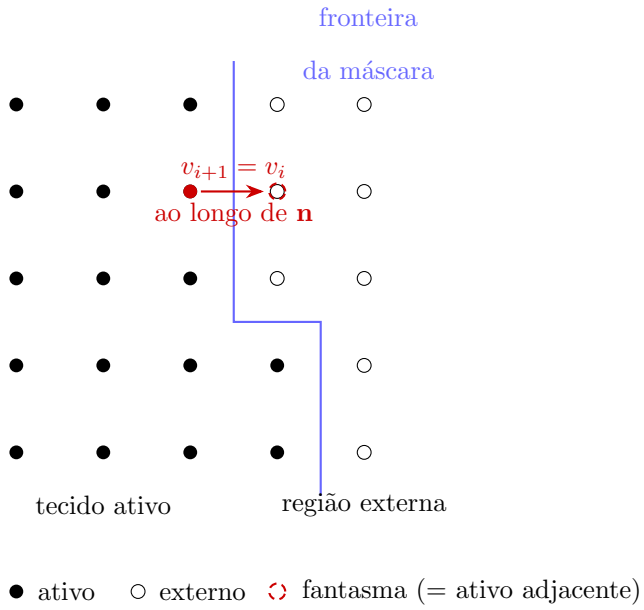

Figura 3.2: Tratamento da fronteira irregular da máscara por células fantasma, ilustrado em um corte bidimensional da malha cartesiana.

3.9.3 Visualização dos Resultados 3D

Após a simulação, os dados podem ser visualizados de duas formas. A primeira consiste em projetar o campo calculado sobre a superfície STL. Essa visualização produz uma imagem mais próxima da geometria anatômica original, mas exige um tratamento adicional, pois a superfície triangular do STL não coincide exatamente com os pontos da malha cartesiana.

A segunda forma consiste em visualizar diretamente a malha cartesiana ativa. Nesse caso, são exibidas as células que pertencem à máscara, coloridas de acordo com o potencial calculado. Essa visualização é menos suave, pois mostra a discretização usada pelo algoritmo, mas representa de forma mais direta o domínio onde a simulação foi executada.

Na visualização sobre a superfície STL, foi usado o filtro `Resample With Dataset` do ParaView para transferir os valores da grade cartesiana para a malha triangular da superfície. Inicialmente, essa projeção podia gerar artefatos, pois os pontos fora da máscara possuem valor nulo e podiam influenciar a interpolação perto da fronteira. Para reduzir esse efeito, foi exportado também um campo auxiliar, chamado `v_surface`. Esse campo não altera a simulação. Ele apenas estende, para a região externa próxima à máscara,

o valor do ponto ativo mais próximo. Com isso, a superfície STL é colorida a partir de valores representativos do tecido ativo, evitando que os zeros fora da máscara produzam regiões visualmente inativas que não correspondem ao comportamento calculado no interior do domínio.

3.10 Plataforma de Testes

A plataforma de testes computacionais é descrita em detalhes na Tabela 3.1. Um ponto dessa configuração merece registro, por afetar diretamente a leitura dos resultados em CPU. A execução ocorreu sob WSL2, que não expôs todos os recursos do processador. Das oito unidades físicas do Ryzen 7 5700G, apenas quatro ficaram visíveis ao ambiente, totalizando oito processadores lógicos. Como o backend de CPU do JAX distribui o cálculo entre os núcleos disponíveis, os tempos medidos em CPU refletem metade da capacidade do processador.

Tabela 3.1: Ambiente computacional usado nos benchmarks.

Item	**Configuração**
GPU	NVIDIA GeForce RTX 4060 Ti, 8 GB
CPU	AMD Ryzen 7 5700G (8 núcleos, 16 threads)
Sistema	WSL2, Ubuntu 24.04.4 LTS
No WSL	4 núcleos, 8 threads
Python	3.12.3
JAX	0.9.2, backend GPU
CUDA	Runtime 12.9.79 (via pacotes pip do JAX)
cuDNN	9.20.0.48
Driver NVIDIA	596.49
Memória	16 GB de RAM

4 Resultados

Este capítulo apresenta os resultados obtidos nas etapas de verificação entre backends, convergência de malha, desempenho computacional e extensão tridimensional do simulador, seguindo a mesma progressão incremental dos casos 1D, 2D e 3D adotada ao longo do trabalho. Os casos 1D e 2D foram usados para verificar a consistência numérica entre NumPy e JAX e para avaliar desempenho e convergência em cenários controlados; a partir desses resultados, a análise se concentra em execuções JAX GPU, abrangendo modelos iônicos mais complexos em 2D e a extensão para o monodomínio 3D com geometria anatômica obtida a partir de uma superfície STL.

4.1 Experimentos Computacionais

Os experimentos foram organizados por aumento gradual de dimensionalidade e complexidade. Os casos 1D foram usados como etapa inicial de verificação e desempenho, pois permitem avaliar o comportamento computacional do simulador em problemas menores e mais controlados. Os casos 2D ampliaram essa análise para domínios espaciais mais representativos, incluindo propagação em onda plana e formação de espirais. A comparação com malhas refinadas foi usada para avaliar o efeito do refinamento sobre a solução. A extensão 3D avaliou a aplicação do simulador em uma geometria mascarada construída a partir de uma superfície STL.

A Tabela 4.1 lista os modelos iônicos usados ao longo dos experimentos e os principais parâmetros adotados em cada caso. Os modelos biestável e FitzHugh-Nagumo foram usados nos testes iniciais de propagação e desempenho. Os modelos Mitchell-Schaeffer e Bueno-Orovio foram usados para avaliar casos mais complexos em 2D, e o modelo Mitchell-Schaeffer também foi adotado na execução 3D com máscara anatômica.

A escolha desses modelos permitiu avaliar o simulador em níveis crescentes de complexidade. O modelo biestável foi usado como caso mais simples de propagação, por conter apenas uma variável de potencial e uma dinâmica de excitação elementar. O mo-

delo FitzHugh-Nagumo acrescenta uma variável de recuperação, permitindo representar propagação com recuperação temporal do meio e protocolos de reentrada. O modelo Mitchell-Schaeffer também trabalha com potencial e variável de porta, mas apresenta uma dinâmica mais próxima de modelos cardíacos simplificados e foi usado nos casos bidimensionais mais complexos e na extensão 3D. Por fim, o modelo de Bueno-Orovio possui maior número de variáveis internas e foi usado como caso mais exigente entre os modelos avaliados.

Tabela 4.1: Constantes dos modelos iônicos usados nos experimentos.

Modelo	**Variáveis**	**Constantes**	**Observação**
Biestável	v	$A = 1,0$; $\alpha = 0,1$	$f(v) = Av(1-v)(v-\alpha)$; sem variável de recuperação.
FitzHugh-Nagumo	v, w	$A = 1,0$; $\alpha = 0,1$; $\epsilon = 0,005$; $\gamma = 2,0$	$f(v,w) = Av(1-v)(v-\alpha)-w$; $g(v,w) = \epsilon(v-\gamma w)$.
Mitchell-Schaeffer	v, h	$\tau_{in} = 0,3$; $\tau_{out} = 6,0$; $\tau_{open} = 120,0$; $\tau_{close} = 150,0$; $v_{gate} = 0,13$	h modula a corrente de entrada; a corrente de saída é proporcional a $-v/\tau_{out}$.
Bueno-Orovio	u, v, w, s	Conjunto epicárdico completo do modelo minimal.	$k = 0,001171$ (cm^2/ms) no termo difusivo do potencial u.

4.2 Verificação entre Backends

A consistência numérica entre backends foi avaliada como uma etapa inicial de apoio à implementação em JAX. A comparação foi feita entre NumPy em CPU e JAX em GPU, sempre com a mesma malha, os mesmos parâmetros e as mesmas condições iniciais. Assim, a análise mede a diferença entre duas implementações da mesma formulação discreta. O efeito do refinamento de malha é analisado na seção seguinte.

Como mostra a Tabela 4.2, os desvios entre NumPy CPU e JAX GPU em dupla precisão ficaram na ordem do erro de arredondamento numérico nos casos 2D avaliados. Como os erros comparam a variável principal no estado final, sempre na mesma malha e com os mesmos parâmetros, isso indica que, para esses problemas discretos, a implementação em JAX reproduz efetivamente a solução da versão NumPy quando ambas executam a mesma formulação. A mesma tabela traz a comparação em precisão simples. Nesse formato os erros ficam entre 10^{-7} e 10^{-6}, contra 10^{-15} ou menos em dupla precisão, que chega a zero no caso biestável. A separação de cerca de nove ordens de grandeza acompanha o épsilon de máquina de cada formato, da ordem de 10^{-7} em `float32` e de 10^{-16} em `float64`. Isso situa o desvio em precisão simples como efeito do arredondamento do próprio formato, sem indício de divergência entre as implementações.

Após essa verificação inicial, as etapas seguintes passaram a usar JAX GPU como backend principal. Os casos mais refinados e a extensão tridimensional foram avaliados em termos de convergência de malha, desempenho e comportamento da solução. A comparação com NumPy permaneceu restrita a esta etapa de consistência, pois seu papel era verificar a equivalência inicial entre as implementações antes das execuções mais custosas.

Tabela 4.2: Verificação entre NumPy CPU e JAX GPU nos casos 2D com baseline NumPy disponível, em precisão simples e dupla. Os erros comparam a variável principal no estado final, sempre na mesma malha e com os mesmos parâmetros, tomando a execução NumPy em dupla precisão como referência. Foram listados os maiores casos 2D para os quais a execução NumPy CPU foi mantida como referência.

Caso	**Malha**	**Precisão simples**		**Precisão dupla**	
		Máx. abs.	L_2 **rel.**	**Máx. abs.**	L_2 **rel.**
Biestável	400×400	$2{,}62 \times 10^{-6}$	$2{,}62 \times 10^{-6}$	0	0
FitzHugh-Nagumo	400×400	$8{,}08 \times 10^{-7}$	$4{,}72 \times 10^{-7}$	$3{,}33 \times 10^{-16}$	$1{,}81 \times 10^{-16}$
FHN periódico	400×400	$1{,}21 \times 10^{-7}$	$2{,}38 \times 10^{-7}$	$2{,}78 \times 10^{-17}$	$8{,}26 \times 10^{-17}$
Espiral S1-S2 FHN	400×400	$2{,}55 \times 10^{-6}$	$7{,}09 \times 10^{-7}$	$1{,}22 \times 10^{-15}$	$5{,}52 \times 10^{-16}$

4.3 Convergência de Malha

O estudo de convergência de malha foi feito com um cenário de onda plana para cada modelo iônico, sem repetir todos os casos usados nos benchmarks de desempenho. O objetivo é avaliar o efeito do refinamento espacial comparando cada malha com uma solução de referência obtida em uma malha mais fina.

Para reduzir o custo computacional, foi executado um cenário representativo de onda plana por modelo. A referência é a solução JAX GPU em dupla precisão na malha $N = 1600$ de cada modelo; os erros são calculados para a variável principal. A Tabela 4.3 reúne, para cada modelo, o erro relativo em norma L_2 em função do refinamento e a ordem observada correspondente. A ordem observada p, mostrada na última coluna, é calculada entre cada malha e a imediatamente mais grossa por $p = \log(E_N/E_{2N})/\log(\Delta x_N/\Delta x_{2N})$, que mede o quanto o erro L2 final cai quando Δx é reduzido pela metade, usando sempre a solução em $N = 1600$ como referência. Valores próximos de 2 indicam que o erro cai cerca de quatro vezes a cada refinamento.

Tabela 4.3: Resumo do estudo de convergência de malha e ordem observada do erro para um caso representativo de cada modelo iônico. A última coluna traz a ordem observada p. O procedimento de cálculo e a referência adotada estão descritos no texto.

Modelo	N	Δx	Δt	**Passos**	**Erro L2 final**	**Erro máx. final**	p **observado**
Biestavel 2D	100	1,00000	0,100000	1.400	$1{,}66 \times 10^{-10}$	$5{,}47 \times 10^{-10}$	–
Biestavel 2D	200	0,50000	0,025000	5.600	$4{,}37 \times 10^{-11}$	$1{,}46 \times 10^{-10}$	1,93
Biestavel 2D	400	0,25000	0,006250	22.400	$1{,}47 \times 10^{-11}$	$4{,}94 \times 10^{-11}$	1,58
Biestavel 2D	800	0,12500	0,001563	89.600	$4{,}36 \times 10^{-12}$	$1{,}47 \times 10^{-11}$	1,75
FitzHugh-Nagumo 2D	100	1,00000	0,100000	1.400	$9{,}28 \times 10^{-2}$	$1{,}68 \times 10^{-1}$	–
FitzHugh-Nagumo 2D	200	0,50000	0,025000	5.600	$3{,}11 \times 10^{-2}$	$5{,}60 \times 10^{-2}$	1,58
FitzHugh-Nagumo 2D	400	0,25000	0,006250	22.400	$1{,}07 \times 10^{-2}$	$1{,}92 \times 10^{-2}$	1,54
FitzHugh-Nagumo 2D	800	0,12500	0,001563	89.600	$3{,}11 \times 10^{-3}$	$5{,}58 \times 10^{-3}$	1,78
Mitchell-Schaeffer 2D	100	1,00000	0,050000	1.400	$1{,}31 \times 10^{-1}$	$6{,}60 \times 10^{-1}$	–
Mitchell-Schaeffer 2D	200	0,50000	0,050000	1.400	$7{,}80 \times 10^{-2}$	$4{,}26 \times 10^{-1}$	0,74
Mitchell-Schaeffer 2D	400	0,25000	0,012500	5.600	$2{,}13 \times 10^{-2}$	$1{,}21 \times 10^{-1}$	1,87
Mitchell-Schaeffer 2D	800	0,12500	0,003125	22.400	$4{,}90 \times 10^{-3}$	$2{,}80 \times 10^{-2}$	2,12
Bueno-Orovio minimal 2D	100	0,02400	0,010000	1.500	$3{,}24 \times 10^{-1}$	$1{,}40 \times 10^{0}$	–
Bueno-Orovio minimal 2D	200	0,01200	0,010000	1.500	$1{,}38 \times 10^{-1}$	$9{,}20 \times 10^{-1}$	1,24
Bueno-Orovio minimal 2D	400	0,00600	0,006148	2.440	$5{,}80 \times 10^{-2}$	$4{,}36 \times 10^{-1}$	1,25
Bueno-Orovio minimal 2D	800	0,00300	0,001537	9.758	$1{,}26 \times 10^{-2}$	$9{,}64 \times 10^{-2}$	2,20

A partir da tabela, observa-se que os erros mais altos aparecem entre malhas diferentes. Esses valores refletem a maior sensibilidade dos modelos Mitchell-Schaeffer e Bueno-Orovio ao refinamento espacial e temporal, principalmente nas regiões de frente de onda. No Mitchell-Schaeffer, a ordem observada passa de 0,74 no refinamento $100 \rightarrow 200$ para 2,12 em $400 \rightarrow 800$. No Bueno-Orovio, os dois primeiros refinamentos ficam próximos de 1,25 e o refinamento $400 \rightarrow 800$ chega a 2,20. Isso sugere que as malhas mais grossas representam a frente de onda com resolução insuficiente, ainda fora da faixa assintótica de convergência. Com o refinamento, a ordem observada se aproxima do valor esperado para o esquema ($p = 2$), indicando a entrada na faixa assintótica.

Para investigar esse comportamento nos modelos mais complexos, as soluções em $N = 400$ foram comparadas diretamente com a referência $N = 1600$ nos mesmos pontos físicos. Como as malhas são aninhadas, a solução refinada foi amostrada a cada quatro pontos em cada direção antes do cálculo do erro. Com os mapas de diferença, é

possível identificar se os erros ficam espalhados pelo domínio ou concentrados na região da frente de onda. A Figura 4.1 apresenta esses mapas para os modelos Mitchell-Schaeffer e Bueno-Orovio, e a Figura 4.2 mostra a evolução temporal do erro correspondente.

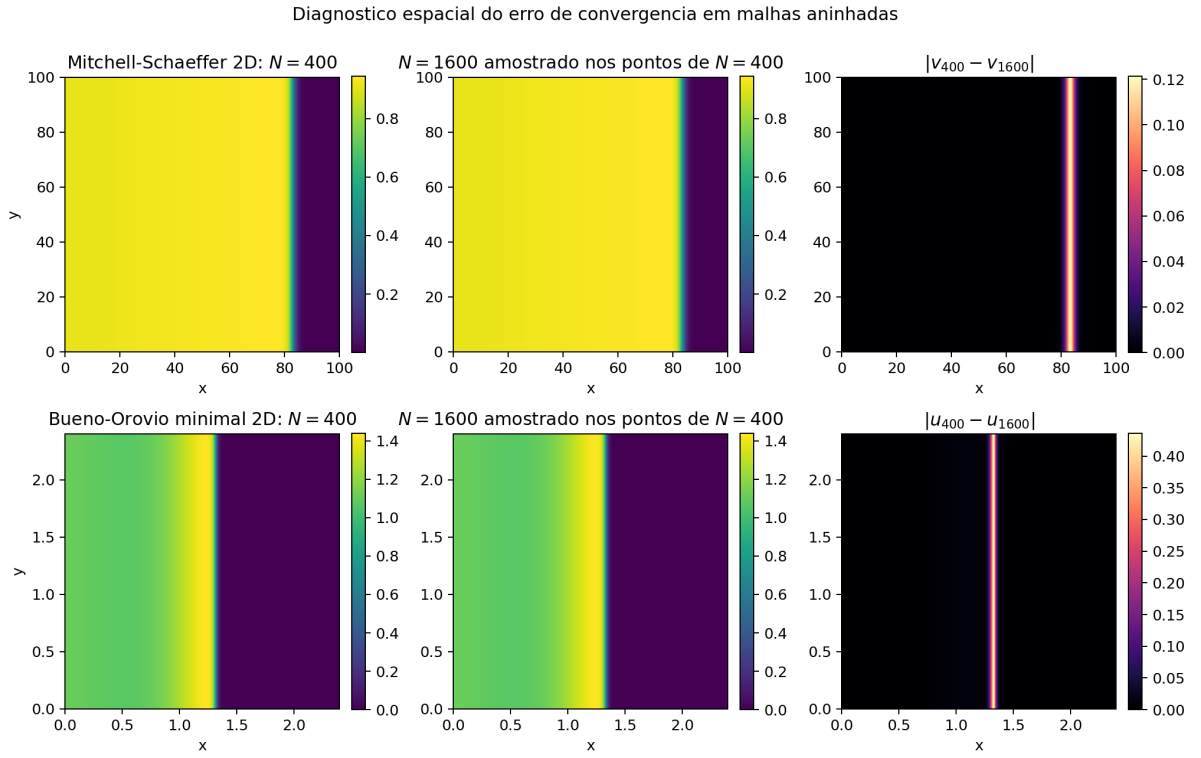

Figura 4.1: Diagnóstico espacial da diferença entre a malha $N = 400$ e a referência $N = 1600$ para os modelos Mitchell-Schaeffer e Bueno-Orovio. A referência refinada foi amostrada nos mesmos pontos físicos da malha grossa.

Os mapas da Figura 4.1 mostram que a diferença entre as malhas se concentra ao longo da frente de despolarização, e não de forma espalhada pelo domínio, o que é consistente com a sub-resolução da frente nas malhas mais grossas discutida anteriormente. A Figura 4.2 complementa esse diagnóstico ao mostrar que o desvio em relação à referência é maior durante a passagem da frente de onda pelo domínio, quando os gradientes espaciais são mais intensos, refletindo a maior sensibilidade dos modelos Mitchell-Schaeffer e Bueno-Orovio à discretização nessa fase.

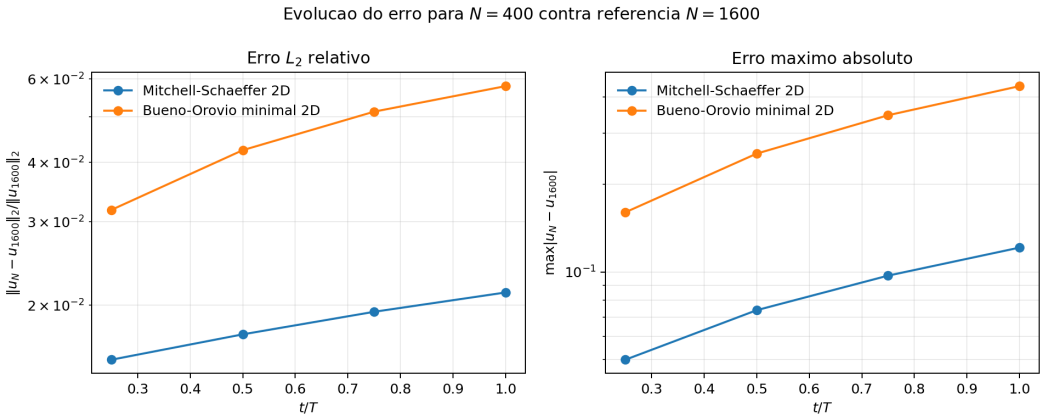

Figura 4.2: Evolução temporal do erro entre $N = 400$ e a referência $N = 1600$ nos modelos Mitchell-Schaeffer e Bueno-Orovio.

4.4 Desempenho Computacional

Depois da verificação entre backends e da análise de convergência de malha, o desempenho computacional foi avaliado pelo tempo médio de execução nos backends considerados. Quando a execução NumPy CPU estava disponível para a mesma malha e o mesmo caso, também foi calculado o speedup das implementações JAX em relação a essa referência.

As execuções em JAX foram cronometradas cinco vezes em todos os casos, após aquecimento. Para a referência em NumPy, esse número foi mantido nas malhas de menor custo, $N = 10^4$ em 1D e 100×100 e 200×200 em 2D. Nas demais, o tempo reportado vem de uma única execução, já que uma repetição isolada chega a cerca de sete horas na malha 1600×1600.

A análise foi organizada de forma incremental. Primeiro, foram avaliados casos unidimensionais, usados como testes controlados de menor custo para observar o efeito do tamanho da malha e da compilação JAX. Em seguida, foram avaliados casos bidimensionais, nos quais o número de pontos da malha e o custo de cada passo temporal aumentam de forma mais significativa. Os resultados tridimensionais são apresentados separadamente na seção de extensão para monodomínio 3D, pois envolvem uma máscara anatômica e uma etapa própria de visualização.

4.4.1 Benchmarks 1D

Os benchmarks unidimensionais foram usados como etapa inicial de avaliação de desempenho em um domínio simples e controlado. Nessa etapa, o objetivo foi observar o comportamento computacional das implementações quando o tamanho do vetor de solução aumenta. Para isso, foram considerados dois tamanhos de problema, com $N = 10.000$ e $N = 1.000.000$, mantendo $\Delta x = 1$, $\Delta t = 0,1$ e 5.000 passos de tempo. Diferentemente dos estudos de refinamento, em que a malha é refinada sobre um domínio físico fixo, aqui o espaçamento e o passo de tempo são mantidos constantes e o aumento de N corresponde a um domínio maior, com mais pontos na mesma resolução. Com Δx e Δt fixos, o número de Fourier permanece constante por construção em $F = k\Delta t/\Delta x^2 = 2{,}0 \cdot 0{,}1/1^2 = 0{,}2$, abaixo do limite de estabilidade em uma dimensão. Na implementação, N representa o índice final da malha, de modo que cada vetor de estado possui $N + 1$ pontos.

Foram avaliados três casos: modelo biestável, modelo FitzHugh-Nagumo e modelo FitzHugh-Nagumo com condição periódica. Em todos eles, a implementação NumPy em CPU foi usada como referência para o cálculo do speedup. A configuração completa desses casos está resumida na Tabela 4.4, e os tempos medidos são apresentados nas Tabelas 4.5, 4.6 e 4.7.

Tabela 4.4: Configuração completa dos casos 1D.

Caso	**Discretização**	**Modelo**	**Condição inicial**
Biestável 1D	$N = 10.000$ ou $1.000.000$; grade com $N + 1$ pontos; $\Delta x = 1$; $\Delta t = 0,1$; 5.000 passos; $T = 500$; $k = 2,0$.	$A = 1,0$; $\alpha = 0,1$.	$v_0[: N/10] = 0,3$.
FHN 1D	$N = 10.000$ ou $1.000.000$; grade com $N + 1$ pontos; $\Delta x = 1$; $\Delta t = 0,1$; 5.000 passos; $T = 500$; $k = 2,0$.	$A = 1,0$; $\alpha = 0,1$; $\epsilon = 0,005$; $\gamma = 2,0$.	$v_0[: N/10] = 0,3$; $w_0 = 0$.
FHN periódico 1D	$N = 10.000$ ou $1.000.000$; grade com $N + 1$ pontos; $\Delta x = 1$; $\Delta t = 0,1$; 5.000 passos; $T = 500$; $k = 2,0$.	$A = 1,0$; $\alpha = 0,1$; $\epsilon = 0,005$; $\gamma = 2,0$.	$v_0[: N/10] = 0,3$; $w_0 = 0$.

Tabela 4.5: Benchmarks 1D para o modelo Biestável. O speedup usa NumPy CPU como baseline no mesmo exercício.

Biestável 1D – 5.000 passos de tempo				
Malha	**Abordagem**	**Precisão**	**Média (s)**	**Speedup**
$N = 10.000$	NumPy CPU	float64	0,205	1,0×
$N = 10.000$	JAX GPU (lax.scan)	float32	0,045	4,6×
$N = 10.000$	JAX GPU (lax.scan)	float64	0,055	3,8×
$N = 10.000$	JAX CPU (lax.scan)	float64	0,072	2,9×
$N = 1.000.000$	NumPy CPU	float64	162,7	1,0×
$N = 1.000.000$	JAX GPU (lax.scan)	float32	0,180	904,7×
$N = 1.000.000$	JAX GPU (lax.scan)	float64	0,825	197,3×
$N = 1.000.000$	JAX CPU (lax.scan)	float64	72,70	2,2×

Tabela 4.6: Benchmarks 1D para o modelo FHN. O speedup usa NumPy CPU como baseline no mesmo exercício.

FHN 1D – 5.000 passos de tempo				
Malha	**Abordagem**	**Precisão**	**Média (s)**	**Speedup**
$N = 10.000$	NumPy CPU	float64	0,370	1,0×
$N = 10.000$	JAX GPU (lax.scan)	float32	0,054	6,9×
$N = 10.000$	JAX GPU (lax.scan)	float64	0,055	6,7×
$N = 10.000$	JAX CPU (lax.scan)	float64	0,086	4,3×
$N = 1.000.000$	NumPy CPU	float64	202,6	1,0×
$N = 1.000.000$	JAX GPU (lax.scan)	float32	0,303	667,9×
$N = 1.000.000$	JAX GPU (lax.scan)	float64	1,549	130,8×
$N = 1.000.000$	JAX CPU (lax.scan)	float64	110,1	1,8×

Tabela 4.7: Benchmarks 1D para o modelo FHN periódico. O speedup usa NumPy CPU como baseline no mesmo exercício.

FHN periódico 1D – 5.000 passos de tempo				
Malha	**Abordagem**	**Precisão**	**Média (s)**	**Speedup**
$N = 10.000$	NumPy CPU	float64	0,645	1,0×
$N = 10.000$	JAX GPU (lax.scan)	float32	0,134	4,8×
$N = 10.000$	JAX GPU (lax.scan)	float64	0,110	5,9×
$N = 10.000$	JAX CPU (lax.scan)	float64	0,157	4,1×
$N = 1.000.000$	NumPy CPU	float64	278,2	1,0×
$N = 1.000.000$	JAX GPU (lax.scan)	float32	0,305	913,6×
$N = 1.000.000$	JAX GPU (lax.scan)	float64	1,719	161,8×
$N = 1.000.000$	JAX CPU (lax.scan)	float64	85,14	3,3×

A Figura 4.3 resume a aceleração relativa dos benchmarks 1D em relação à referência NumPy CPU. Os resultados mostram que, para $N = 10.000$, as implementações em JAX apresentam ganhos moderados em relação ao NumPy. Nesse tamanho de problema, o custo fixo de lançamento das operações na GPU, que não cresce com o tamanho do problema, ainda tem peso relevante no tempo de execução. Para $N = 1.000.000$, o ganho em GPU se torna mais expressivo, chegando a centenas de vezes em relação à

execução NumPy em CPU.

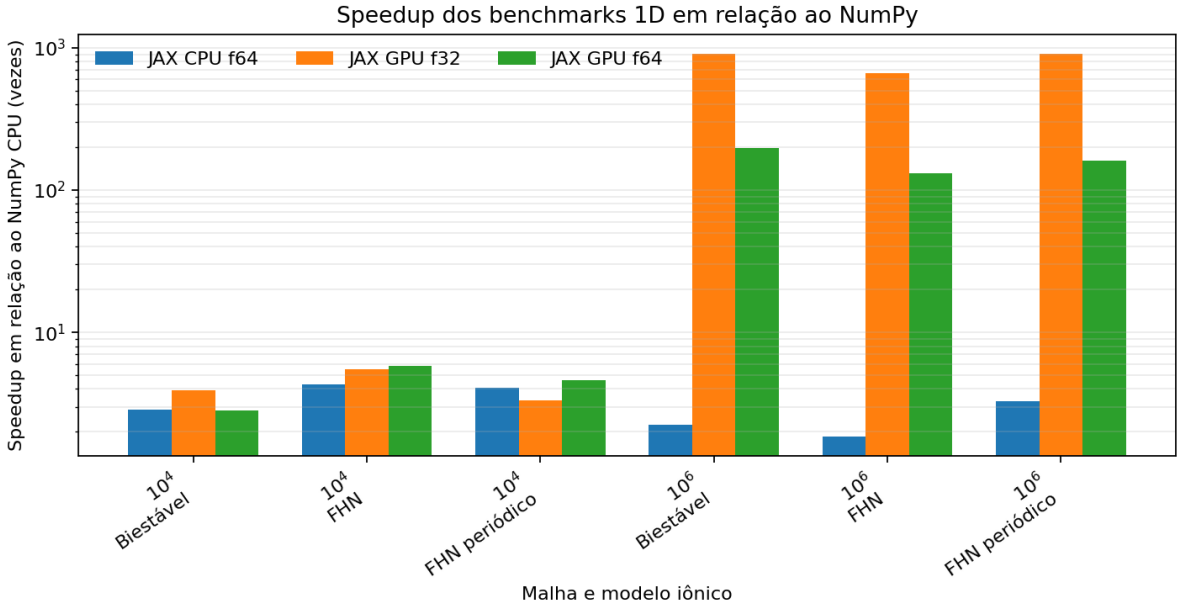

Figura 4.3: Aceleração relativa dos benchmarks 1D, usando a implementação NumPy em CPU como baseline.

Na comparação entre NumPy CPU e JAX CPU, a referência em NumPy executa em uma única thread, enquanto o backend de CPU do JAX distribui o cálculo entre os núcleos disponíveis, de modo que parte da diferença observada entre os dois já decorre do uso de múltiplos núcleos, somando-se ao efeito dos mecanismos de compilação. Em dupla precisão e $N = 1.000.000$, essa razão ficou entre aproximadamente 1,9× e 3,7× conforme o cenário. O ambiente WSL usado nos testes expõe oito processadores lógicos sobre quatro núcleos físicos; tomando como base os núcleos físicos efetivamente disponíveis, a eficiência paralela fica na faixa de 50% a 90%. Uma hipótese para esse retorno está na limitação por banda de memória: o estêncil de diferenças finitas realiza poucas operações por ponto em relação ao volume de dados que precisa ler e escrever a cada passo. Uma vez saturada a banda disponível, acrescentar núcleos deixa de reduzir proporcionalmente o tempo de execução. Soma-se a isso o fato de o avanço temporal ser sequencial, com um ponto de sincronização por passo, o que faz o custo relativo de coordenação entre threads crescer quando o trabalho por passo é pequeno.

Entre as cinco execuções em JAX, a diferença entre o menor e o maior tempo foi pequena. Nos benchmarks 1D com $N = 10^6$, ficou abaixo de 1% da média na maior

parte dos cenários em dupla precisão, com um único caso em precisão simples chegando a cerca de 8%. Os tempos reportados são, portanto, estáveis frente à repetição, e as diferenças discutidas nesta seção, de uma ou mais ordens de grandeza, estão muito acima dessa variação.

Outro ponto observado nos benchmarks 1D é a diferença entre as execuções em precisão simples e dupla. Em GPU, as execuções em `float32` apresentaram speedups maiores do que as execuções em `float64`. Isso ocorre porque, em dupla precisão, cada valor ocupa mais memória e as operações aritméticas tendem a ter menor vazão na GPU utilizada. Assim, parte do ganho obtido pelo paralelismo da GPU é reduzido pelo maior custo de armazenamento, movimentação de dados e cálculo em `float64`. Essa diferença ajuda a interpretar por que os maiores speedups aparecem nas execuções em precisão simples.

4.4.2 Benchmarks 2D

No benchmark 2D de onda plana, o domínio físico foi mantido fixo com $L = 100$ e o tempo final foi $T = 140$. Como o domínio é fixo, cada malha tem $\Delta x = L/N$, e o refinamento reduz o espaçamento à medida que N cresce. Para manter a razão difusiva constante, o passo de tempo é recalculado a cada malha por $\Delta t = F\,\Delta x^2/k$ com $F = k\Delta t/\Delta x^2 = 0,2$, abaixo do limite de estabilidade difusiva em duas dimensões; assim Δt diminui proporcionalmente a Δx^2 e o número de passos para cobrir o tempo final T cresce na mesma proporção. Nesse caso, nenhuma malha atingiu o teto de passo de tempo, de modo que F permaneceu igual a $0,2$ em todas elas. Os parâmetros do modelo biestável foram $k = 2,0$, $A = 1,0$ e $\alpha = 0,1$. Como esses valores são constantes entre malhas, as tabelas de refinamento mostram apenas os parâmetros que variam com N. O modelo FitzHugh-Nagumo em 2D aparece em dois contextos: a configuração de onda plana, com a mesma discretização do caso biestável, é usada no estudo de convergência de malha; já a avaliação de desempenho do FitzHugh-Nagumo em 2D é feita no caso com protocolo S1-S2 para formação de espiral, apresentado em seguida.

A Tabela 4.8 reúne os parâmetros de discretização comuns aos dois casos 2D de onda plana, biestável e FitzHugh-Nagumo, que variam com o refinamento da malha, e a

Tabela 4.9 apresenta a configuração completa desses dois casos.

Nas tabelas de desempenho, o speedup é calculado em relação à execução NumPy CPU da mesma malha e do mesmo caso. Para o modelo biestável, essa referência foi obtida em todas as malhas; na malha 1600×1600 a execução em CPU levou cerca de 7 horas, o que evidencia o custo do refinamento sem aceleração. Para a espiral FHN, a referência NumPy não foi executada nas malhas mais refinadas devido ao alto custo em CPU, de modo que o speedup correspondente fica indisponível nesses casos.

Tabela 4.8: Parâmetros que variam com o refinamento de malha nos casos 2D de onda plana.

Parâmetro	100×100	200×200	400×400	800×800	1600×1600
Grade	101×101	201×201	401×401	801×801	1601×1601
Δx	1,0000	0,5000	0,2500	0,1250	0,0625
Δt	0,100000	0,025000	0,006250	0,001563	0,000391
Passos de tempo	1.400	5.600	22.400	89.600	358.400

Tabela 4.9: Configuração completa dos casos 2D de onda plana.

Caso	**Discretização**	**Modelo**	**Condição inicial**
Biestável 2D	$L = 100$; $N \in \{100, 200, 400, 800, 1600\}$; grade $(N+1) \times (N+1)$; $\Delta x = L/N$; $\Delta t = 0,2\Delta x^2/k$; $k = 2,0$; $T = 140$.	$A = 1,0$; $\alpha = 0,1$.	$v_0[: N/10, :] = 0,3$.
FitzHugh-Nagumo 2D	$L = 100$; $N \in \{100, 200, 400, 800, 1600\}$; grade $(N+1) \times (N+1)$; $\Delta x = L/N$; $\Delta t = 0,2\Delta x^2/k$; $k = 2,0$; $T = 140$.	$A = 1,0$; $\alpha = 0,1$; $\epsilon = 0,005$; $\gamma = 2,0$.	$v_0[: N/10, :] = 0,3$; $w_0 = 0$.

A espiral FHN foi gerada com o protocolo S1-S2 em um domínio quadrado com $L = 300$ unidades espaciais e condição de Neumann nas bordas. O primeiro estímulo (S1)

foi aplicado em uma faixa à esquerda do domínio, com $v = 0{,}3$ e $w = 0$, em variáveis normalizadas do modelo; no instante alvo $t = 180$ ms, um segundo estímulo (S2) foi aplicado na metade direita do domínio, atualizando v para o maior valor entre o estado atual e 0,3. O tempo final alvo foi $T = 500$ ms e os parâmetros do modelo foram $k = 2{,}0$, $A = 1{,}0$, $\alpha = 0{,}1$, $\epsilon = 0{,}005$ e $\gamma = 2{,}0$. Para cada malha, usou-se grade $(N+1) \times (N+1)$, $\Delta x = L/N$, na mesma unidade espacial do domínio, e passo de tempo escolhido pela restrição difusiva com teto $\Delta t \leq 0{,}05$ ms. Diferentemente do benchmark de onda plana, nas malhas mais grossas o passo difusivo $F\Delta x^2/k$ excederia esse teto, de modo que o passo de tempo é limitado por $\Delta t_{\max} = 0{,}05$ ms e o número de Fourier efetivo fica abaixo do valor alvo; apenas nas malhas mais finas o passo volta a ser ditado pela restrição difusiva. Por isso, a Tabela 4.10 apresenta apenas os valores que mudam com o refinamento da malha ou com a contagem discreta dos estímulos. A Figura 4.4 ilustra esse protocolo, com instantâneos da evolução da espiral formada ao longo do tempo.

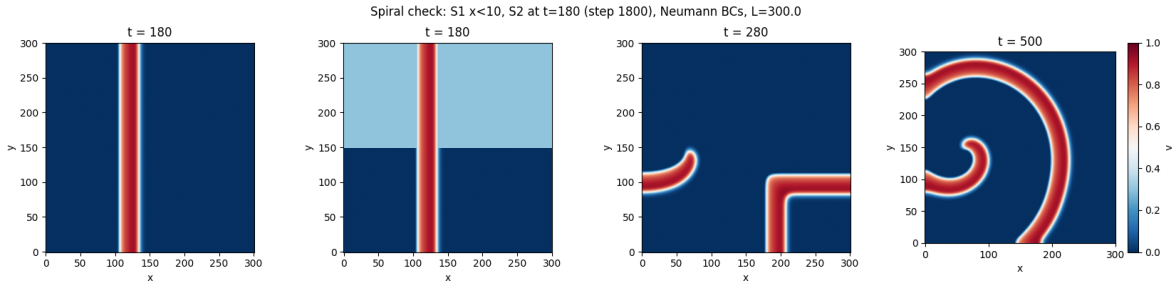

Figura 4.4: Simulação do protocolo S1-S2 para formação de espiral no modelo FitzHugh-Nagumo.

Tabela 4.10: Parâmetros variáveis da espiral FHN com protocolo S1-S2. Os parâmetros fixos do protocolo S1-S2 foram descritos na definição do problema. O valor de CFL reportado é o efetivo de cada malha.

Parâmetro	$N = 100$	$N = 200$	$N = 400$	$N = 800$	$N = 1.600$
Δx	3,0000	1,5000	0,7500	0,3750	0,1875
Δt	0,050000	0,050000	0,050000	0,014063	0,003516
CFL efetivo	0,011	0,044	0,178	0,200	0,200
Passos	10.000	10.000	10.000	35.556	142.222
Passo S2	3.600	3.600	3.600	12.800	51.200
$S1_W$ (pontos)	3	7	13	27	53

As Tabelas 4.11 e 4.12 apresentam os tempos medidos para o caso biestável e para a espiral FHN, respectivamente, e a Figura 4.5 resume a aceleração relativa correspondente. Nos benchmarks 2D, os ganhos em relação ao NumPy aumentam com o refinamento da malha. Nas malhas menores, o custo fixo de lançamento das operações na GPU, que não cresce com a malha, ainda tem impacto relevante no tempo de execução. À medida que o número de pontos cresce, o custo do cálculo passa a dominar, e a execução em JAX GPU apresenta aceleração mais expressiva. No caso biestável, a referência NumPy CPU foi medida em todas as malhas, e o speedup em GPU chega a mais de duas ordens de grandeza em dupla precisão na malha 1600×1600, superando três ordens de grandeza em precisão simples. Esses resultados motivam o uso de JAX GPU nos casos refinados apresentados a seguir.

Tabela 4.11: Benchmarks 2D para o caso Biestável. O domínio físico permanece fixo e o refinamento é feito pela redução de Δx. O speedup usa NumPy CPU como baseline no mesmo caso; quando esse baseline não foi executado, o speedup fica indisponível.

Biestável 2D – domínio físico fixo e refinamento por redução de Δx				
Malha	**Abordagem**	**Precisão**	**Média (s)**	**Speedup**
100×100	NumPy CPU	float64	0,141	1,0×
100×100	JAX GPU (lax.scan)	float32	0,017	8,4×
100×100	JAX GPU (lax.scan)	float64	0,016	8,6×
200×200	NumPy CPU	float64	1,448	1,0×
200×200	JAX GPU (lax.scan)	float32	0,060	24,1×
200×200	JAX GPU (lax.scan)	float64	0,062	23,4×
400×400	NumPy CPU	float64	19,88	1,0×
400×400	JAX GPU (lax.scan)	float32	0,497	40,0×
400×400	JAX GPU (lax.scan)	float64	0,700	28,4×
800×800	NumPy CPU	float64	1427,7	1,0×
800×800	JAX GPU (lax.scan)	float32	2,813	507,5×
800×800	JAX GPU (lax.scan)	float64	5,425	263,2×
1600×1600	NumPy CPU	float64	25992,7	1,0×
1600×1600	JAX GPU (lax.scan)	float32	16,90	1538,1×
1600×1600	JAX GPU (lax.scan)	float64	111,3	233,5×

Tabela 4.12: Benchmarks 2D para a espiral FHN S1-S2. O tempo de NumPy corresponde a uma execução única quando disponível. Nas malhas em que a referência NumPy foi omitida, o speedup fica indisponível.

FHN 2D com protocolo S1-S2				
Malha	**Abordagem**	**Precisão**	**Média (s)**	**Speedup**
100 × 100	NumPy CPU	float64	1,634	1,0×
100 × 100	JAX GPU (lax.scan)	float32	0,273	6,0×
100 × 100	JAX GPU (lax.scan)	float64	0,281	5,8×
200 × 200	NumPy CPU	float64	3,372	1,0×
200 × 200	JAX GPU (lax.scan)	float32	0,280	12,0×
200 × 200	JAX GPU (lax.scan)	float64	0,300	11,3×
400 × 400	NumPy CPU	float64	13,82	1,0×
400 × 400	JAX GPU (lax.scan)	float32	0,284	48,6×
400 × 400	JAX GPU (lax.scan)	float64	0,282	48,9×
800 × 800	JAX GPU (lax.scan)	float32	0,616	–
800 × 800	JAX GPU (lax.scan)	float64	3,189	–
1600 × 1600	JAX GPU (lax.scan)	float32	12,07	–
1600 × 1600	JAX GPU (lax.scan)	float64	67,54	–

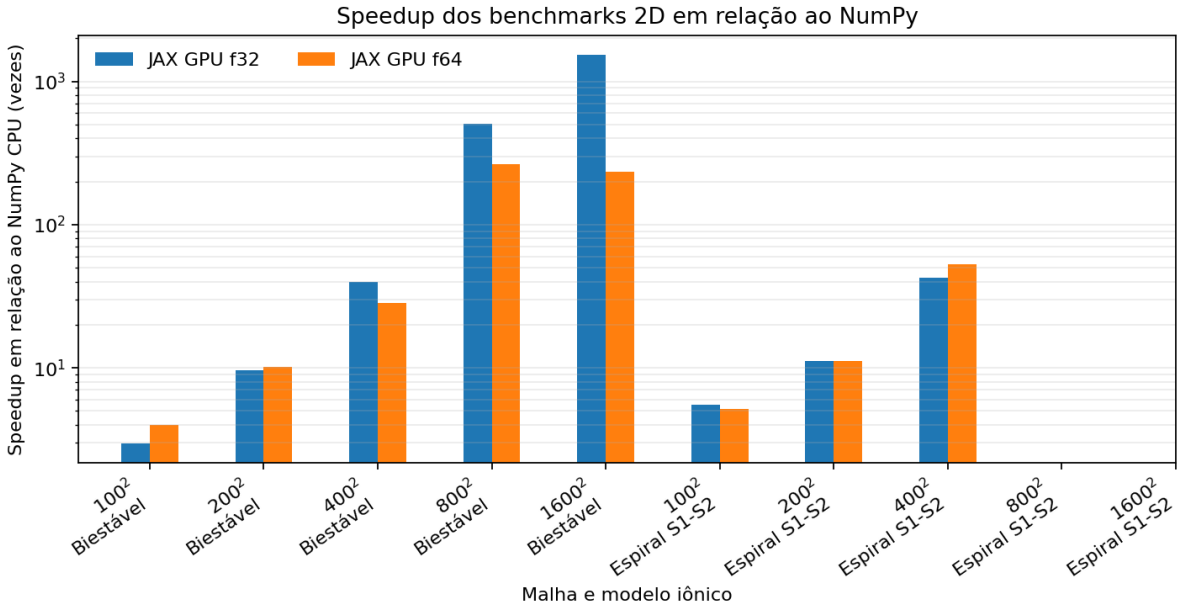

Figura 4.5: Aceleração relativa dos benchmarks 2D, usando a implementação NumPy em CPU como baseline.

A comparação entre precisão simples e dupla também evidencia o efeito do tipo de ponto flutuante sobre o desempenho da GPU. Nas malhas menores, os tempos em `float32` e `float64` ficam próximos, pois o tempo de execução nessas malhas ainda é dominado pelo custo fixo de lançamento das operações na GPU, que não depende da precisão. Conforme a malha é refinada, a razão entre o tempo em dupla e em simples precisão cresce de forma sistemática, passando de praticamente um na malha 100×100 para cerca de seis na malha 1600×1600. Esse comportamento é esperado em GPUs de uso geral por dois motivos que se somam. Primeiro, a taxa de operações em ponto flutuante de 64 bits é uma fração da taxa em 32 bits no hardware utilizado. Segundo, cada arranjo em dupla precisão ocupa o dobro de memória e exige o dobro de tráfego entre a memória global e as unidades de cálculo. Nas malhas pequenas esses custos ficam mascarados por esse custo fixo de lançamento; nas malhas refinadas a simulação passa a ser limitada pela capacidade de cálculo e pela largura de banda de memória, e a penalidade da dupla precisão deixa de ser amortizada. Por esse motivo, a precisão simples mantém um ganho expressivo na malha 1600×1600, enquanto a dupla precisão, embora ainda muito mais rápida que o NumPy, paga um custo relativo maior.

4.5 Casos Refinados em 2D

Depois dos benchmarks principais, foram executados casos adicionais com modelos mais complexos. Esses casos fazem a transição entre os benchmarks 2D e a extensão tridimensional, pois combinam dinâmica iônica mais exigente e malhas refinadas adequadas à escala e ao custo de cada modelo. As Tabelas 4.13 e 4.14 resumem essas execuções.

Tabela 4.13: Configuração completa dos modelos adicionais.

Caso	**Discretização**	**Modelo**	**Condição inicial**
Mitchell-Schaeffer onda plana	$N = 1.600$; grade 1601×1601; $L = 100,0$; $\Delta x = 0,06250$; $\Delta t = 0,000781$; $k = 1,0$; 89.600 passos; $T = 70,0$.	$\tau_{in} = 0,3$; $\tau_{out} = 6,0$; $\tau_{open} = 120,0$; $\tau_{close} = 150,0$; $v_{gate} = 0,13$.	$v_0[: N/10,:] = 0,3$; $h_0 = 1$.
Mitchell-Schaeffer espiral S1-S2	$N = 1.600$; grade 1601×1601; $L = 300,0$; $\Delta x = 0,18750$; $\Delta t = 0,007031$; $k = 1,0$; 128.000 passos; $T = 900,0$; $t_{S2} = 400,0$.	$\tau_{in} = 0,3$; $\tau_{out} = 6,0$; $\tau_{open} = 120,0$; $\tau_{close} = 150,0$; $v_{gate} = 0,13$.	$v_0[: 160,:] = 0,4$; $h_0 = 1$; S2 em $v[:,: 800]$ com amplitude $0,4$.
Bueno-Orovio onda plana	$N = 400$; grade 401×401; $L = 2,4$ cm; $\Delta x = 0,00600$; $\Delta t = 0,006149$; $k = 0,001171$; 48.792 passos; $T = 300,0$ ms.	Parâmetros epicárdicos do modelo minimal.	$u_0[: 25,:] = 1,0$; demais pontos em repouso: $u_0 = 0$, $v_0 = 1$, $w_0 = 1$, $s_0 = 0$.

Tabela 4.14: Desempenho dos modelos e domínios adicionais. A malha de cada caso é indicada no cabeçalho do respectivo bloco. Quando a referência NumPy CPU é omitida pelo custo computacional, o speedup fica indisponível.

Mitchell-Schaeffer – onda plana; malha 1600^2; 89.600 passos			
Abordagem	**Precisão**	**Média (s)**	**Speedup**
JAX GPU	float64	43,16	–
Mitchell-Schaeffer – espiral S1-S2; malha 1600^2; 128.000 passos			
JAX GPU	float64	83,51	–
Bueno-Orovio – onda plana; malha 400^2; 48.792 passos			
NumPy CPU	float64	711,3	1,0×
JAX GPU	float64	9,357	76,0×

4.5.1 Mitchell-Schaeffer

No modelo Mitchell-Schaeffer, foram executados dois casos em malha refinada 1600×1600: um de onda plana e outro com protocolo S1-S2 para formação de espiral. No caso de onda plana, um estímulo inicial aplicado à esquerda do domínio gera uma frente que se propaga até o tempo final $t = 70$ ms. A Figura 4.6 mostra o potencial normalizado ao final dessa simulação.

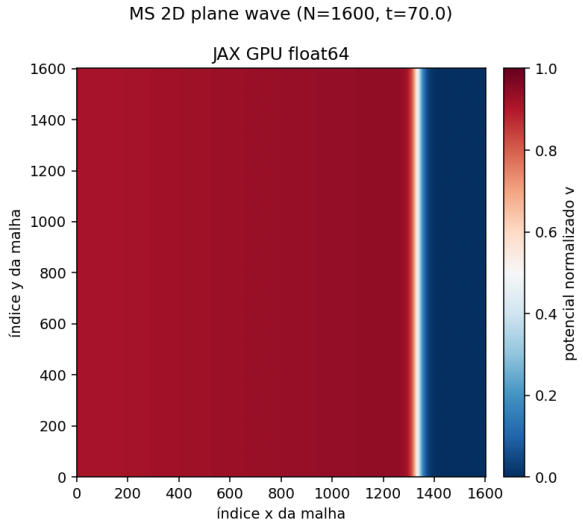

Figura 4.6: Onda plana propagada no modelo Mitchell-Schaeffer em malha refinada, ao final da simulação em $t = 70$ ms.

No caso da espiral, o protocolo S1-S2 aplica um segundo estímulo em $t = 400$ ms, e a frente de onda resultante se curva e forma uma estrutura espiral até o tempo final $t = 900$ ms. A Figura 4.7 apresenta snapshots dessa evolução.

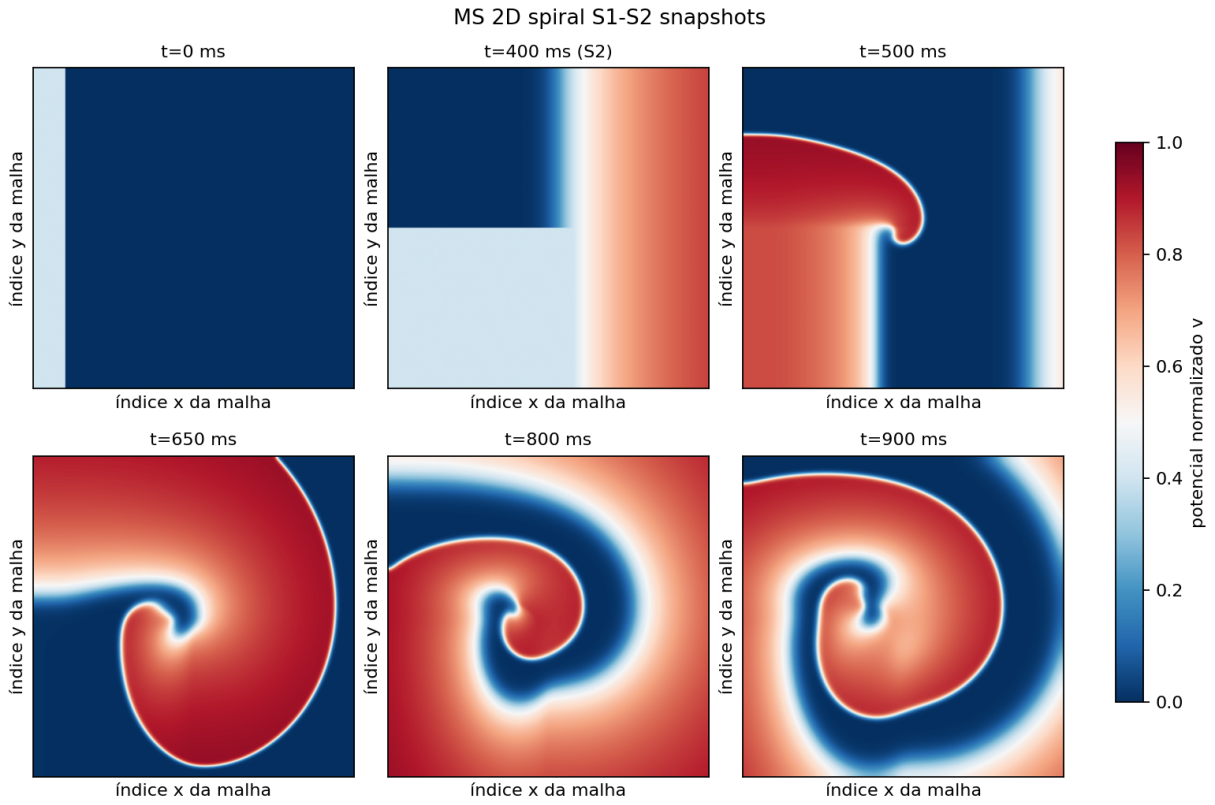

Figura 4.7: Snapshots da formação de espiral no modelo Mitchell-Schaeffer em malha refinada.

4.5.2 Bueno-Orovio

No modelo de Bueno-Orovio, foi executado um caso de onda plana até o tempo final $t = 300$ ms, suficiente para cobrir um potencial de ação completo, da despolarização à repolarização. Diferentemente dos demais modelos, que usam unidades normalizadas, este caso foi definido em unidades físicas, com domínio em centímetros e tempo em milissegundos; por isso o coeficiente de difusão k assume o valor $0,001171$ em $\mathrm{cm}^2/\mathrm{ms}$. A simulação usou malha 400×400, com $\Delta x = 0,006$ cm, resolução suficiente para representar a frente de onda do modelo. Por ser o modelo mais custoso entre os avaliados, com passo de tempo reduzido, uma malha tão refinada quanto a dos casos de Mitchell-Schaeffer elevaria demais o custo da referência em NumPy CPU; a malha 400×400 foi escolhida por

equilibrar resolução adequada e custo viável para a comparação. A Figura 4.8 mostra a evolução do potencial em seis instantes, cobrindo a propagação da frente de onda, o platô e a repolarização do tecido até o retorno ao repouso. A diferença entre as soluções NumPy e JAX permaneceu na ordem do arredondamento numérico. Por possuir quatro variáveis de estado, o modelo exige mais operações por ponto a cada passo de tempo, mas a execução em JAX GPU manteve o caso computacionalmente viável, com aceleração expressiva em relação à referência NumPy CPU.

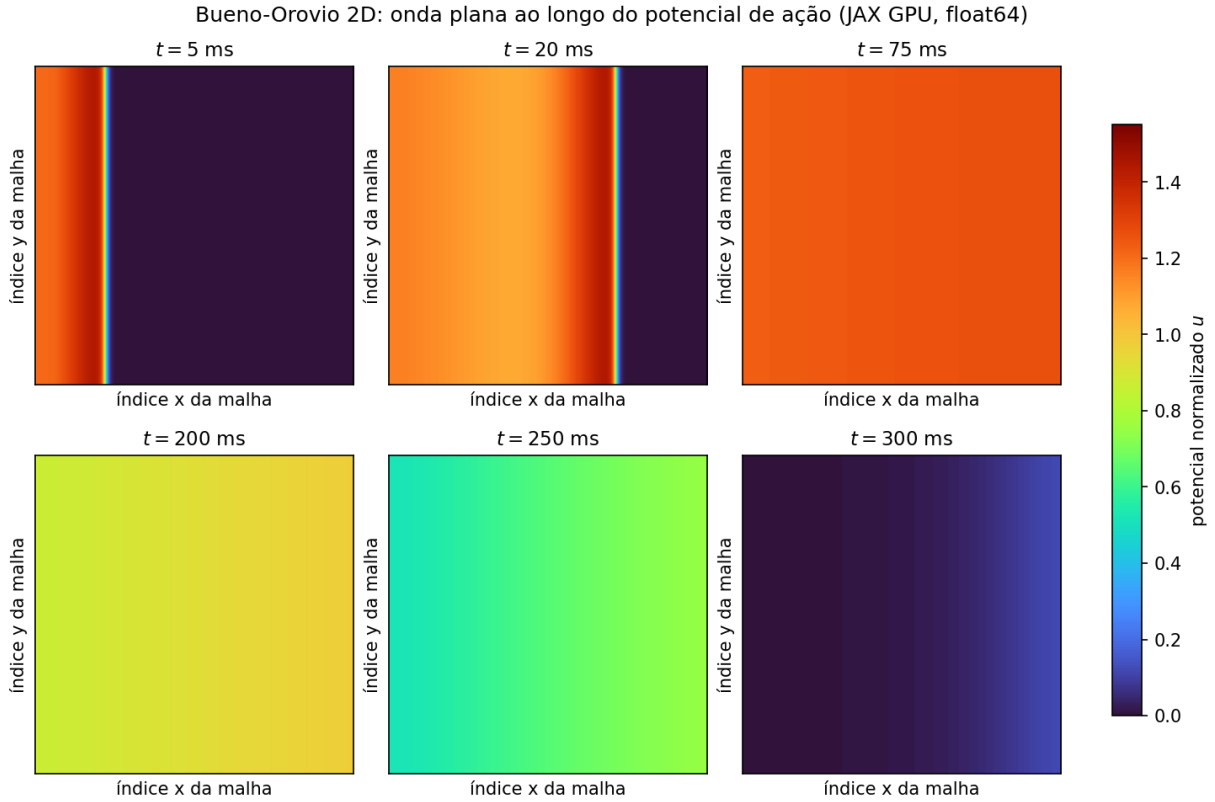

Figura 4.8: Onda plana no modelo minimal de Bueno-Orovio em malha 400×400, em seis instantes do potencial de ação. A frente de despolarização entra pela esquerda e percorre o domínio ($t = 5$ e 20 ms), o tecido atinge o platô ($t = 75$ ms) e em seguida repolariza, retornando ao repouso por volta de $t = 300$ ms. Solução em JAX GPU com dupla precisão; a diferença para a referência NumPy permanece na ordem do arredondamento numérico.

Em conjunto, os casos refinados em 2D mostram que a implementação em JAX GPU foi capaz de executar modelos mais exigentes, com dinâmica iônica mais complexa, em malhas refinadas. Esses resultados complementam os benchmarks de desem-

penho e servem como etapa intermediária antes da execução tridimensional com máscara anatômica.

4.5.3 Decomposição do Tempo de Execução

O tempo de uma execução JAX não corresponde apenas ao cálculo numérico. Ele reúne etapas distintas, entre elas a compilação *just-in-time* do laço de tempo, que ocorre uma única vez na primeira chamada. Para separar essas contribuições nas execuções em que o tempo foi anotado, cada etapa foi isolada por compilação *ahead-of-time*, com as chamadas `.lower()` e `.compile()` da função compilada. Esse procedimento mede, de forma independente, a transferência dos arranjos iniciais para a GPU, a construção do *jaxpr* (tracing), a geração do executável pela XLA (compilação), a execução do laço já compilado e a transferência do resultado de volta para a CPU.

Todas as etapas foram cronometradas com `time.perf_counter`, e cada medida é precedida de uma sincronização explícita do dispositivo com `block_until_ready`, para que o tempo registrado corresponda ao cálculo concluído. Não foi empregado um perfilador dedicado, como o JAX Profiler: a separação por etapas vem da instrumentação manual descrita acima. A etapa de execução foi medida após uma chamada de aquecimento, que paga o custo de compilação fora do cronômetro, e corresponde ao valor puro de cálculo. Diferentemente dos benchmarks da Seção 4.4, que reportam a média de cinco execuções, o valor aqui é o menor tempo entre três execuções, escolhido por representar a medida menos afetada por interferência do sistema. A Tabela 4.15 apresenta essa decomposição para as execuções cronometradas dos modelos Mitchell-Schaeffer em 2D e do caso 3D com máscara obtida de STL, apresentado adiante.

Tabela 4.15: Decomposição do tempo de parede das execuções JAX em etapas. A etapa de execução é o valor puro de cálculo: o laço `lax.scan` já compilado, medido após aquecimento (melhor de repetições). A compilação XLA é um custo único, isolado por compilação *ahead-of-time* (`.lower()`/`.compile()`). O *tracing* e as transferências entre CPU e GPU completam o tempo de parede.

Execução	Dim.	Passos	host→dev. (s)	Tracing (s)	Compilação (s)	Execução (s)	dev.→host (s)	Total (s)
Mitchell-Schaeffer 2D, onda plana	2D	89.600	0,91	0,05	0,20	43,16	0,17	44,49
Mitchell-Schaeffer 2D, espiral S1-S2	2D	128.000	0,73	0,05	0,23	83,51	0,17	84,68
Monodomínio 3D, máscara STL	3D	39.106	0,00	0,22	2,04	154,16	0,22	156,63

A decomposição mostra que a compilação é um custo fixo, que não depende do tamanho da malha, enquanto a execução cresce com o número de pontos e de passos de tempo. Nas malhas grandes usadas nesses casos, a compilação representa uma fração pequena do tempo total, e a execução domina o tempo. A etapa de execução é o valor puro de cálculo: o laço `lax.scan` já compilado, medido após aquecimento. A compilação XLA é um custo único, isolado por compilação *ahead-of-time* (`.lower()`/`.compile()`). O *tracing* e as transferências entre CPU e GPU completam o tempo. Essas transferências têm peso reduzido porque o estado permanece no dispositivo durante todo o laço. Como o avanço temporal é expresso com `lax.scan`, os arrays são enviados uma única vez no início e recuperados uma única vez ao final, em vez de a cada passo. No caso 3D com máscara, o envio para a GPU e o retorno para a CPU somaram cerca de 0,22 s, contra 154,16 s de cálculo, menos de 0,2% do tempo total. Por isso, os tempos comparados nos benchmarks correspondem à etapa de execução, medida após o aquecimento, de modo a excluir o custo único de compilação. Essa etapa de execução reúne um custo fixo de lançamento das operações, que não depende do tamanho da malha, e o custo do cálculo, que cresce com o número de pontos e de passos; nas malhas grandes o cálculo domina, e nas pequenas o custo fixo de lançamento ainda tem peso relevante.

4.6 Extensão para Monodomínio 3D

A etapa final dos resultados avalia a extensão do simulador para o monodomínio tridimensional. Primeiro, a formulação 3D foi testada em uma caixa completa, usando

malhas aninhadas para observar o comportamento do erro com o refinamento. Em seguida, a implementação foi aplicada a uma geometria mascarada obtida a partir de STL, que representa o caso final de aplicação do simulador em uma geometria irregular.

4.6.1 Convergência em Caixa 3D

Como etapa inicial da extensão para o monodomínio 3D, foi feito um estudo de convergência de malha na caixa cartesiana completa, sob as mesmas condições do caso anatômico apresentado a seguir: o modelo Mitchell-Schaeffer, o mesmo domínio físico da geometria obtida de STL e o mesmo protocolo de estímulo, em JAX GPU com dupla precisão. A única diferença é a ausência da máscara, o que isola o efeito do refinamento da malha sem a influência da geometria irregular. A malha mais refinada foi usada como referência, e as soluções das malhas mais grossas foram avaliadas nos mesmos pontos físicos.

Como o domínio corresponde à caixa que envolve a geometria obtida de STL, com extensões distintas nas três direções, os espaçamentos Δx, Δy e Δz não são iguais entre si, o que motiva o uso da forma somada e anisotrópica da condição de estabilidade apresentada na metodologia. O passo de tempo foi definido mantendo a soma dos números de Fourier por direção em 0,45, sujeita ao teto $\Delta t_{\max} = 0{,}05$ ms. Nas malhas mais grossas, $N = 32$, $N = 64$ e $N = 128$, o passo difusivo excederia esse teto, de modo que todas usaram $\Delta t = 0{,}05$ ms, com 6.000 passos para cobrir $t = 300$ ms; apenas a malha de referência $N = 256$ operou no limite difusivo, com passo de tempo menor. Como o passo é o mesmo nas três malhas mais grossas, a contribuição do erro temporal é comum a elas, e a diferença observada entre os níveis reflete principalmente o refinamento espacial

Os parâmetros dessa análise aparecem na Tabela 4.16 e os erros na Tabela 4.17. A solução de referência foi obtida na malha $N = 256$ em dupla precisão, e as malhas $N = 32$, $N = 64$ e $N = 128$ foram avaliadas nos mesmos pontos físicos para a variável de potencial v. A ordem foi avaliada em $t = 300$ ms, instante em que a frente de onda ainda está em plena propagação. Em tempos mais longos o tecido repolariza e o estado tende a um valor de repouso quase uniforme, o que reduz a norma da solução de referência e torna o erro relativo em norma L_2 pouco representativo. A coluna p obs. indica a ordem

observada entre a linha e a próxima malha duas vezes mais refinada.

Tabela 4.16: Parâmetros da análise de precisão do monodomínio 3D. A referência é a solução JAX GPU em dupla precisão na malha mais refinada.

Domínio	N_{ref}	N **comparados**	Δx_{ref}	Δt_{ref}	**Passos ref.**	**Pontos ref.**
Caixa 3D completa	256	32, 64, 128	0,48502	0,015343	19.553	16.974.593

Tabela 4.17: Erro do monodomínio 3D em relação à malha mais refinada. A solução refinada foi amostrada nos mesmos pontos físicos da malha grossa. Os erros foram calculados para a variável de potencial v. A coluna p obs. indica a ordem observada entre a linha e a próxima malha duas vezes mais refinada.

Domínio	N	Δx	Δt	**Pontos**	**Erro L2 final**	**Erro máx. final**	**Maior L2**	p **obs.**
Caixa 3D completa	32	3,88020	0,050000	35.937	$7{,}42 \times 10^{-2}$	$4{,}85 \times 10^{-2}$	$7{,}42 \times 10^{-2}$	2,06
Caixa 3D completa	64	1,94010	0,050000	274.625	$1{,}78 \times 10^{-2}$	$1{,}06 \times 10^{-2}$	$1{,}78 \times 10^{-2}$	1,17
Caixa 3D completa	128	0,97005	0,050000	2.146.689	$7{,}94 \times 10^{-3}$	$4{,}84 \times 10^{-3}$	$7{,}94 \times 10^{-3}$	–

No primeiro refinamento, o erro relativo final em norma L_2 foi reduzido de $7{,}42 \times 10^{-2}$ em $N = 32$ para $1{,}78 \times 10^{-2}$ em $N = 64$, o que corresponde a uma ordem observada de 2,06. Esse valor é compatível com a ordem dois esperada para o esquema de diferenças centradas usado na discretização espacial e indica que esses dois níveis estão dentro da faixa assintótica de convergência.

No refinamento seguinte, de $N = 64$ para $N = 128$, o erro continua diminuindo, de $1{,}78 \times 10^{-2}$ para $7{,}94 \times 10^{-3}$, mas a ordem observada cai para 1,17, abaixo do valor esperado. Essa redução não decorre de erro na implementação: a estimativa de ordem só é confiável quando as malhas estão na faixa assintótica e o erro da solução de referência é desprezível frente ao erro da malha avaliada (ROY, 2005; NASA Glenn Research Center, 2021). Como $N = 128$ está a apenas um nível de refinamento da referência $N = 256$, seu erro já se aproxima do erro de discretização da própria referência, e a diferença entre as duas malhas deixa de refletir apenas o erro de $N = 128$, o que reduz a ordem observada. O refinamento bem separado $N = 32 \rightarrow 64$, por outro lado, recupera a ordem esperada.

Em conjunto, o estudo em caixa 3D mostra que o refinamento reduz o erro de forma monotônica em relação à referência mais fina, com ordem próxima de dois no nível

bem separado ($N = 32 \rightarrow 64$), e que a ordem reduzida no nível mais próximo da referência é um efeito metodológico da razão de refino, e não uma limitação do simulador.

As execuções dessa análise têm custo modesto: o tempo de cálculo, medido após uma chamada de aquecimento, vai de 0,44 s na malha 32^3 a 80,2 s na malha de referência 256^3, passando por 1,04 s em 64^3 e 3,72 s em 128^3. Uma referência ainda mais fina, 512^3, não foi adotada por restrição de memória: em dupla precisão, essa malha ocupa praticamente toda a memória da GPU utilizada (8 GB), o que inviabiliza seu uso como referência no fluxo do estudo.

4.6.2 Execução em Geometria Mascarada por STL

Depois da análise de convergência na caixa completa, foi executado um caso em geometria anatômica obtida a partir de uma superfície STL. Nessa etapa, o STL foi convertido em uma máscara cartesiana refinada com $256 \times 256 \times 256$ células, correspondente a uma grade de $257 \times 257 \times 257$ pontos. A caixa que envolve a geometria anatômica não é cúbica: suas extensões físicas diferem entre os eixos. Como a grade usa o mesmo número de células nas três direções, o espaçamento resultante difere por eixo, com $\Delta x = 0{,}4850$, $\Delta y = 0{,}5430$ e $\Delta z = 0{,}3774$. Isso não compromete a isotropia da difusão, pois o operador laplaciano discreto divide a segunda diferença pelo espaçamento de cada direção, Δx_d^2, de modo que o coeficiente físico permanece igual nos três eixos. A simulação foi feita com o modelo Mitchell-Schaeffer até $t = 600$ ms, usando JAX GPU em dupla precisão. O passo de tempo foi $\Delta t = 0{,}015342914$ ms, totalizando 39.106 passos internos, com difusão isotrópica (coeficientes iguais nas três direções) e estímulo inicial aplicado em uma região lateral da máscara. A máscara final possui 2.588.453 pontos ativos, o que representa aproximadamente 15,25% da grade total. As métricas dessa execução aparecem na Tabela 4.18.

Tabela 4.18: Métricas da execução 3D usada nas visualizações do monodomínio em geometria anatômica obtida de STL. O tempo de execução é o valor puro de cálculo do laço já compilado, medido após aquecimento; a compilação XLA aparece em separado por ser um custo único. A decomposição completa em etapas está na Tabela 4.15. A máscara foi gerada a partir da superfície STL usando `vtkSelectEnclosedPoints`.

Métrica	**Valor**
Modelo iônico	Mitchell-Schaeffer
Backend	JAX GPU em dupla precisão
Malha	$257 \times 257 \times 257$ pontos
Células da grade	$256 \times 256 \times 256$
Pontos ativos na máscara	2.588.453 ($15,25\%$ da malha)
Δx, Δy, Δz	0,4850; 0,5430; 0,3774
Δt	0,015342914 ms
Tempo final	600,0 ms
Passos de tempo	39.106
Tempo de execução (cálculo)	154,16 s
Compilação (única)	2,04 s
Tempo de geração da máscara	23,01 s
Série exportada	61 instantes, de $t = 0$ a $t = 600$

4.6.3 Visualização da Propagação

A Figura 4.9 mostra a propagação na malha cartesiana ativa usada pela simulação. Essa visualização foi adotada para representar diretamente o domínio numérico calculado pelo algoritmo. A solução é apresentada nos blocos da própria máscara cartesiana, o que evita artefatos visuais que podiam aparecer quando o campo tridimensional era projetado sobre a superfície anatômica.

Para a coloração da superfície, foram exibidas apenas as células cujos oito vértices pertencem ao tecido ativo. Esse critério evita um artefato de visualização: as células de fronteira da máscara possuem alguns vértices fora do tecido, onde o potencial vale zero, e a interpolação entre esses vértices e os vértices ativos introduzia faixas artificiais de cor sobre a superfície, dando a regiões já despolarizadas uma aparência indevidamente azulada. Restringir a exibição às células totalmente internas remove essas faixas sem

alterar a solução calculada, que continua a usar todas as células ativas da máscara com a condição de fluxo nulo descrita na Seção 3.9.2.

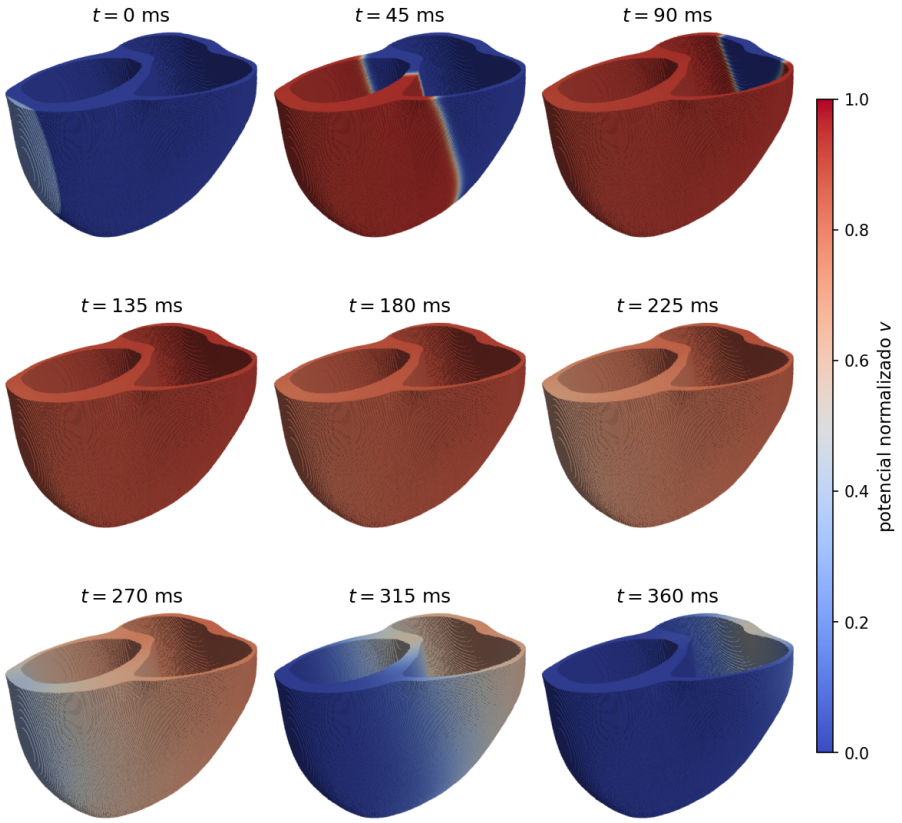

Figura 4.9: Visualização do caso 3D em geometria anatômica obtida de STL, com máscara cartesiana refinada em $256 \times 256 \times 256$ células. A sequência mostra a solução do modelo Mitchell-Schaeffer em nove instantes igualmente espaçados, de $t = 0$ a $t = 360$ ms, intervalo que abrange a atividade elétrica completa: a partir do repouso, a frente de despolarização percorre o tecido, atinge o platô e em seguida o tecido se repolariza, retornando ao repouso por volta de $t = 350$ ms. A simulação prosseguiu até $t = 600$ ms, permanecendo em repouso após esse intervalo. A renderização foi feita no ParaView diretamente sobre a malha cartesiana ativa.

O estudo em caixa 3D indica que o refinamento reduz o erro em relação à referência mais fina. A execução com máscara obtida de STL complementa essa análise ao mostrar que a mesma estrutura numérica também pode ser aplicada a uma geometria irregular, desde que o domínio anatômico seja representado por uma máscara cartesiana. Com isso, a etapa 3D inclui tanto uma verificação em geometria simples quanto uma aplicação mais próxima do uso pretendido do simulador.

5 Conclusão

Este trabalho teve como objetivo avaliar o uso de JAX, em CPU e em GPU, na simulação numérica de modelos simplificados de eletrofisiologia cardíaca, comparando-o a uma implementação de referência em NumPy. A avaliação foi conduzida em cenários crescentes de complexidade, abrangendo casos 1D, 2D e a extensão do monodomínio para 3D com geometria realística. Ao longo dos experimentos, buscou-se verificar a consistência numérica da implementação em JAX, medir o ganho computacional em relação à referência em NumPy e avaliar a viabilidade da abordagem em modelos iônicos com diferentes níveis de complexidade.

A comparação inicial entre backends mostrou que a implementação em JAX reproduziu a solução obtida em NumPy nos casos controlados avaliados. Nos casos 2D em dupla precisão, os erros entre NumPy CPU e JAX GPU ficaram próximos ao arredondamento numérico, indicando equivalência prática entre as duas implementações para a mesma formulação discreta. Essa etapa foi importante para dar suporte ao uso posterior de JAX GPU como backend principal nas simulações mais refinadas e na extensão tridimensional.

Os benchmarks indicaram ganho computacional expressivo com JAX, principalmente nas malhas maiores. Nos problemas pequenos, o custo fixo associado à compilação e ao uso da GPU ainda tem peso relevante no tempo total. Com o aumento do número de pontos e de passos temporais, a execução compilada passa a ser melhor aproveitada. Esse ganho está associado ao conjunto de adaptações feitas para JAX, incluindo o uso de `jax.jit`, operações vetorizadas, execução em GPU e estruturação do avanço temporal com `jax.lax.scan`. O `lax.scan` foi especialmente relevante por permitir que a repetição dos passos de tempo fosse expressa de forma compatível com compilação, reduzindo a dependência de laços Python no trecho mais custoso da simulação.

Os resultados também mostraram que a abordagem é viável para diferentes modelos iônicos. O simulador foi avaliado com os modelos biestável, FitzHugh-Nagumo, Mitchell-Schaeffer e de Bueno-Orovio, passando de casos simples de propagação para mo-

delos com maior custo numérico. Nos estudos de convergência, o refinamento da malha reduziu o erro em relação à solução mais refinada, e os maiores desvios ficaram associados aos modelos mais complexos e às regiões de frente de onda, onde a solução é mais sensível à discretização espacial e temporal.

Na etapa tridimensional, a eficiência da implementação em JAX também foi observada na execução do monodomínio 3D mascarado. A simulação final utilizou uma grade de $257 \times 257 \times 257$ pontos, com 2.588.453 pontos ativos na máscara, e foi executada com o modelo Mitchell-Schaeffer em JAX GPU com dupla precisão. Mesmo com esse tamanho de domínio e com 39.106 passos temporais, a simulação foi concluída em 154,16 s de tempo de cálculo, ou 156,63 s considerando também a transferência de dados, o traçado e a compilação, medidos como a melhor de três execuções após aquecimento. Esse resultado reforça a viabilidade do uso de JAX em problemas maiores, nos quais o custo computacional inviabilizaria a repetição frequente dos testes em uma implementação puramente baseada em CPU.

De forma geral, os resultados indicam que JAX é uma alternativa viável para acelerar simulações de eletrofisiologia cardíaca baseadas em diferenças finitas. A principal contribuição do trabalho foi mostrar que a mesma formulação discreta pode ser implementada de maneira compatível com compilação e execução em GPU, mantendo consistência com a versão NumPy em casos controlados e permitindo avançar para simulações maiores, como o monodomínio 3D mascarado.

5.1 Limitações e Trabalhos Futuros

Apesar dos resultados obtidos, este trabalho ainda apresenta limitações. A geometria 3D foi representada por uma máscara cartesiana, o que preserva a simplicidade da implementação e produz uma fronteira discreta em blocos. Além disso, o modelo utilizado não incorporou efeitos da anisotropia da propagação elétrica devido à organização do tecido cardíaco em fibras. Os modelos iônicos avaliados também são fenomenológicos, com poucas variáveis de estado, e não descrevem em detalhe as correntes iônicas individuais do miócito cardíaco. Os resultados em CPU, por sua vez, estão condicionados ao ambiente de execução. Sob WSL2, apenas quatro dos oito núcleos físicos do processador ficaram

disponíveis, de modo que o desempenho medido em CPU subestima o que a máquina completa alcançaria.

Como trabalhos futuros, podem ser exploradas comparações com simuladores consolidados de eletrofisiologia cardíaca, a inclusão de modelos anatômicos com propriedades mais realistas e a avaliação de estratégias numéricas mais adequadas para geometrias complexas. Outro caminho é avaliar o simulador com modelos iônicos biofísicos detalhados, como o modelo ToR-ORd de miócito ventricular humano (TOMEK et al., 2019), que descreve explicitamente as correntes iônicas e possui dezenas de variáveis de estado. Esses modelos são fisiologicamente mais realistas, porém mais custosos por ponto de malha, o que torna a aceleração em GPU oferecida por JAX especialmente relevante para viabilizar seu uso em simulações de larga escala. Além disso, seria relevante investigar a paralelização em múltiplas GPUs com JAX. A documentação da biblioteca descreve o uso de arrays distribuídos, particionamento automático da computação e programação manual por dispositivo com `shard_map`, recursos que podem permitir a divisão de malhas tridimensionais maiores entre diferentes dispositivos (JAX, 2026c; JAX, 2026f).

Outras direções dizem respeito à avaliação de desempenho. O baseline em CPU adotado aqui é uma implementação NumPy de thread única, o que limita o alcance da comparação. Uma referência mais exigente seria uma versão em Numba, que também compila em tempo de execução e preserva código Python de alto nível, mas permite paralelizar o laço de diferenças finitas entre múltiplos núcleos com `@njit(parallel=True)` e `prange`, com esforço de adaptação comparável ao exigido pelo JAX. A atribuição de custo às etapas da execução também poderia ser refinada com ferramentas de perfilamento dedicadas, como o JAX Profiler, em lugar da cronometragem manual empregada aqui. Os casos bidimensionais também poderiam ser executados no backend de CPU do JAX, inclusive em precisão simples, o que separaria o ganho devido à compilação do ganho devido à GPU também nesse regime. Essa avaliação ficou de fora pelo custo de execução nas malhas mais refinadas, em que uma única execução em CPU se estende por horas.

Do ponto de vista numérico, permanece em aberto avaliar se a precisão simples é suficiente para os cenários de interesse. Outra possibilidade é explorar precisão mista, combinando precisão simples no armazenamento e no transporte de dados com precisão

dupla nas operações mais sensíveis. Também seria relevante quantificar o desvio entre a solução obtida aqui e a de simuladores consolidados, estendendo a comparação dos tempos de execução para os próprios campos calculados.

O esquema explícito adotado impõe a restrição $\Delta t \propto \Delta x^2$, que se torna limitante nas malhas mais refinadas. Esquemas implícitos ou semi-implícitos removeriam essa restrição, ao custo de resolver sistemas lineares a cada passo, e sua avaliação em JAX é um desdobramento natural deste trabalho. Por fim, a diferenciação automática, um dos recursos centrais do JAX e que não foi explorado aqui, abre caminho para problemas inversos e estimação de parâmetros a partir de dados.

Bibliografia

BERG, L. A. et al. Toward cardiac electrophysiology digital twins with an efficient open source scalable solver on GPU clusters. *Scientific Reports*, 2026.

BRADBURY, J. et al. *JAX: composable transformations of Python+NumPy programs*. 2018. Disponível em: ⟨https://github.com/jax-ml/jax⟩.

BUENO-OROVIO, A.; CHERRY, E. M.; FENTON, F. H. Minimal model for human ventricular action potentials in tissue. *Journal of Theoretical Biology*, v. 253, n. 3, p. 544–560, 2008.

CAMPOS, J. O. et al. Lattice boltzmann method for parallel simulations of cardiac electrophysiology using GPUs. *Journal of Computational and Applied Mathematics*, Elsevier, v. 295, p. 70–82, 2016.

COMTOIS, P.; KNELLER, J.; NATTEL, S. Of circles and spirals: Bridging the gap between the leading circle and spiral wave concepts of cardiac reentry. *Europace*, v. 7, n. s2, p. S10–S20, 2005.

DANIEL, E. *Boundary conditions*. 2020. Finite Volume method: from 1D to 2D. Acesso em: 31 maio 2026. Disponível em: ⟨https://code-mphi.fr/documents/VF1D_2D/chapitre5.html⟩.

FENTON, F. H. et al. Multiple mechanisms of spiral wave breakup in a model of cardiac electrical activity. *Chaos*, v. 12, n. 3, p. 852–892, 2002.

FITZHUGH, R. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, v. 1, n. 6, p. 445–466, 1961.

HARRIS, C. R. et al. Array programming with numpy. *Nature*, v. 585, p. 357–362, 2020.

HOLMES, M. H. *Introduction to numerical methods in differential equations*. [S.l.]: Springer, 2007.

JAX. *Asynchronous dispatch*. 2026. Acesso em: 10 maio 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/async_dispatch.html⟩.

JAX. *Benchmarking JAX code*. 2026. Acesso em: 25 abr. 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/benchmarking.html⟩.

JAX. *Distributed arrays and automatic parallelization*. 2026. Acesso em: 05 jun. 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/parallel.html⟩.

JAX. *jax.lax.scan*. 2026. Acesso em: 10 maio 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/_autosummary/jax.lax.scan.html⟩.

JAX. *Just-in-time compilation*. 2026. Acesso em: 10 maio 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/jit-compilation.html⟩.

JAX. *Manual parallelism with shard_map*. 2026. Acesso em: 05 jun. 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/notebooks/shard_map.html⟩.

JAX. *Quickstart: How to think in JAX*. 2026. Acesso em: 25 abr. 2026. Disponível em: ⟨https://docs.jax.dev/en/latest/notebooks/thinking_in_jax.html⟩.

KABOUDIAN, A.; CHERRY, E. M.; FENTON, F. H. Real-time interactive simulations of complex ionic cardiac cell models in 2d and 3d heart structures with GPUs on personal computers. In: *Computing in Cardiology*. [S.l.: s.n.], 2021. p. 1–4.

LANGTANGEN, H. P.; LINGE, S. *Finite Difference Computing with PDEs: A Modern Software Approach*. Cham: Springer, 2017. Disponível em: ⟨https://link.springer.com/book/10.1007/978-3-319-55456-3⟩.

MENA, A.; FERRERO, J. M.; MATAS, J. F. R. GPU accelerated solver for nonlinear reaction-diffusion systems. application to the electrophysiology problem. *Computer Physics Communications*, v. 196, p. 280–289, 2015.

MITCHELL, C. C.; SCHAEFFER, D. G. A two-current model for the dynamics of cardiac membrane. *Bulletin of Mathematical Biology*, v. 65, n. 5, p. 767–793, 2003.

NAGUMO, J.; ARIMOTO, S.; YOSHIZAWA, S. An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, v. 50, n. 10, p. 2061–2070, 1962.

NAMORATO, F. L. et al. Development of a three-dimensional computational pipeline in python for personalized heart modeling. In: *Computing in Cardiology*. [S.l.: s.n.], 2025. v. 52, p. 1–4.

NASA Glenn Research Center. *Examining Spatial (Grid) Convergence*. 2021. Acesso em: 10 maio 2026. Disponível em: ⟨https://www.grc.nasa.gov/www/wind/valid/tutorial/spatconv.html⟩.

NIEDERER, S. A. et al. Simulating human cardiac electrophysiology on clinical timescales. *Frontiers in Physiology*, v. 2, p. 14, 2011. Disponível em: ⟨https://www.frontiersin.org/journals/physiology/articles/10.3389/fphys.2011.00014/full⟩.

OLIVEIRA, R. S. et al. Performance evaluation of GPU parallelization, space-time adaptive algorithms, and their combination for simulating cardiac electrophysiology. *International Journal for Numerical Methods in Biomedical Engineering*, v. 34, n. 2, p. e2913, 2018.

ROCHA, B. M. et al. Accelerating cardiac excitation spread simulations using graphics processing units. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 23, n. 7, p. 708–720, 2011.

ROY, C. J. Review of code and solution verification procedures for computational simulation. *Journal of Computational Physics*, v. 205, n. 1, p. 131–156, 2005.

SEL, K. et al. Building digital twins for cardiovascular health: From principles to clinical impact. *Journal of the American Heart Association*, v. 13, n. 19, p. e031981, 2024.

SUNDNES, J. et al. *Computing the Electrical Activity in the Heart*. [S.l.]: Springer, 2006.

TOMEK, J. et al. Development, calibration, and validation of a novel human ventricular myocyte model in health, disease, and drug block. *eLife*, v. 8, p. e48890, 2019.

VASCONCELLOS, E. C. et al. Accelerating simulations of cardiac electrical dynamics through a multi-GPU platform and an optimized data structure. *Concurrency and Computation: Practice and Experience*, v. 32, n. 5, p. e5528, 2020.