
UNIVERSIDADE FEDERAL DO MARANHÃO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA - CCET

ENGENHARIA DA COMPUTAÇÃO

Nerval de Jesus Santos Junior

# Discrete Spatial Modeling: Uma proposta de arcabouço para construção de modelos dinâmicos espacialmente explícitos baseado no ecossistema Python

São Luís - MA

26 de julho de 2025

Nerval de Jesus Santos Junior

# Discrete Spatial Modeling: Uma proposta de arcabouço para construção de modelos dinâmicos espacialmente explícitos baseado no ecossistema Python

Monografia apresentada ao curso de Engenharia da Computação, como requisito parcial necessário para obtenção do grau de Bacharel em Engenharia da Computação. Centro de Ciência Exatas e Tecnológicas da Universidade Federal do Maranhão – CCET UFMA.

Engenharia da Computação

Universidade Federal do Maranhão

Orientador: Dr. Sérgio Souza Costa

São Luís - MA

26 de julho de 2025

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Diretoria Integrada de Bibliotecas/UFMA

Santos Junior, Nerval.
Discrete Spatial Modeling: Uma proposta de arcabouço para construção de modelos dinâmicos espacialmente explícitos baseado no ecossistema Python / Nerval Santos Junior. - 2025.
91 f.

Orientador(a): Sérgio Souza Costa.
Monografia (Graduação) - Curso de Engenharia da Computação, Universidade Federal do Maranhão, Centro de Ciências Exatas e Tecnologia, 2025.

1. Sistemas de Informação Geográfica. 2. Modelagem Dinâmica Espacial. 3. Dados Geoespaciais. 4. Python. 5. Ciência Aberta. I. Souza Costa, Sérgio. II. Título.

Nerval de Jesus Santos Junior

# Discrete Spatial Modeling: Uma proposta de arcabouço para construção de modelos dinâmicos espacialmente explícitos baseado no ecossistema Python

Monografia apresentada ao curso de Engenharia da Computação, como requisito parcial necessário para obtenção do grau de Bacharel em Engenharia da Computação. Centro de Ciência Exatas e Tecnológicas da Universidade Federal do Maranhão – CCET UFMA.

Trabalho de Monografia. São Luís - MA, __ de _____ de 2025.

---

**Dr. Sérgio Souza Costa**
Orientador
Universidade Federal do Maranhão

---

**Dr. Bruno Feres de Souza**
Examinador
Universidade Federal do Maranhão

---

**Dr. Luiz Henrique Neves Rodrigues**
Examinador
Universidade Federal do Maranhão

São Luís - MA
26 de julho de 2025

*Aos meus pais, irmãos e namorada.*

*Aos amigos, pelo apoio e companheirismo.*

# Agradecimentos

Agradeço antes de tudo a Deus, que me deu força, saúde e inteligência, para superar as dificuldades.

A minha família, minha avó querida, que sempre me deu todo o amor do mundo, amor esse que é essencial para eu ser quem eu sou.

Agradeço em seguida aos meus pais, que foram essenciais para a minha caminhada até aqui: Minha mãe, ao qual devo muito amor e sempre me apoiou durante toda a graduação, me deu suporte em muitos momentos, pela educação, amor, carinho e apoio incondicional, sem os quais seria impossível trilhar o caminho que sigo.

Aos familiares, se fazem presentes nos momentos de dificuldade e também nos de alegria.

A todos os meus amigos. Aos que estão longe e aos que estão perto. Sem eles, não daria para sorrir depois das dificuldades.

Ao meu orientador da pesquisa de graduação professor Sérgio Souza Costa, que dedicou tempo e acreditou que seria possível este trabalho, pela orientação, paciência e pelo apoio ao longo do tempo em que trabalhamos juntos.

Em especial agradeço a minha namorada e companheira, Nathalia, pela compreensão, incentivo, amor e pela sua paciência em ouvir minhas reclamações e em revisar este trabalho, que desde sempre me apoiou e me mostrou que sou capaz do que eu quiser, em todos as áreas da minha vida. Amo você, meu amor!

E a todos que de alguma forma contribuíram para essa conquista: Meus mais sinceros agradecimentos.

*"Estamos aqui para fazer alguma diferença no universo, se não, por que estar aqui?"*

(Steve Jobs)

# Resumo

Este trabalho apresenta o *DisSModel* (Discrete Spatial Modeling), um *framework* modular e extensível desenvolvido em Python para modelagem dinâmica espacialmente explícita, com foco em aplicações de mudanças no uso e cobertura da terra (LUCC). Inspirado no TerraME, o *DisSModel* oferece suporte à construção de autômatos celulares, simulação de sistemas dinâmicos e integração com dados geográficos. O suporte a modelos baseados em agentes está em desenvolvimento. Estruturado em quatro módulos principais (*Core*, *Geo*, *Models* e *Visualization*), o *framework* permite a manipulação de dados vetoriais e matriciais, definição de vizinhanças espaciais, simulação baseada em eventos discretos e visualizações interativas. A metodologia envolveu a análise das funcionalidades do TerraME, o desenho de uma arquitetura modular adaptada ao ecossistema Python e a validação do *framework* por meio de estudos de caso representativos, como simulações de incêndios florestais, modelos epidemiológicos e padrões espaciais emergentes. O *DisSModel* encontra-se disponível para testes no TestPyPI (<https://test.pypi.org/project/dissmodel/>) e no repositório GitHub do grupo LambdaGeo (<https://github.com/LambdaGeo/dissmodel/>). A proposta busca ampliar o acesso à modelagem espacial dinâmica, promovendo práticas de ciência aberta, reprodutibilidade e integração com fluxos de trabalho modernos em análise de dados geoespaciais.

**Palavras-chave**: Sistemas de Informação Geográfica; Modelagem dinâmica espacial; Dados geoespaciais; Python; Ciência aberta.

# Abstract

This work presents *DisSModel* (Discrete Spatial Modeling), a modular and extensible framework developed in Python for spatially explicit dynamic modeling, with a focus on land use and land cover change (LUCC) applications. Inspired by TerraME, *DisSModel* supports the development of cellular automata, simulation of dynamic systems, and integration with geographic data. Support for agent-based models is currently under development. Structured into four main modules (*Core*, *Geo*, *Models*, and *Visualization*), the framework enables manipulation of vector and raster data, definition of spatial neighborhoods, event-based simulation, and interactive visualizations. The methodology involved analyzing the functionalities of TerraME, designing a modular architecture adapted to the Python ecosystem, and validating the framework through representative case studies, such as forest fire simulations, epidemiological models, and emergent spatial patterns. *DisSModel* is available for testing on TestPyPI (<https://test.pypi.org/project/dissmodel/>) and in the LambdaGeo group repository on GitHub (<https://github.com/LambdaGeo/dissmodel/>). The proposal aims to broaden access to spatial dynamic modeling, fostering open science practices, reproducibility, and integration with modern geospatial data analysis workflows.

**Keywords**: Geographic Information Systems; Dynamic spatial modeling; Geospatial data; Python; Open science.

# Lista de ilustrações

# Lista de Códigos

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| UFMA | Universidade Federal do Maranhão |
| SIG | Sistemas de Informação Geográfica |
| RDF | Resource Description Framework |
| URI | Uniform Resource Identifier |
| API | Application Programming Interface |
| ESRI | Environmental Systems Research Institute |
| QGIS | Quantum GIS |
| GRASS GIS | Geographic Resources Analysis Support System GIS |
| PySAL | Python Spatial Analysis Library |
| GeoJSON | JavaScript Object Notation Geographics |
| GeoPandas | Geographic Pandas |
| TerraME | Terra Modeling Environment |
| GIS | Geographic Information System |
| DBCells | Database Cells |
| LUCC | Land Use and Cover Change |
| CLUMPY | Clustered Land Use and Management Pattern Analyzer |
| OpenLDM | Open Land-use Dynamics Model |
| eKDE | Empirical Kernel Density Estimation |
| ArcGIS | Geographic Information System by ESRI |
| Python | Linguagem de programação |
| Lua | Linguagem de programação usada no Framework TerraME |
| Scopus | Base de dados científica |
| RQ | Research Question (Questão de Pesquisa) |

| | |
|---|---|
| Bayes-eKDE | Método de calibração baseado em Kernel Density Estimation |
| EC | Critério de Exclusão |
| IC | Critério de Inclusão |
| IEEE | Institute of Electrical and Electronics Engineers |
| QPs | Questões de Pesquisa |
| IEEE Xplore | Digital Library of the IEEE |
| Shapely | Biblioteca para análise de geometria espacial |
| Fiona | Biblioteca para leitura e escrita de arquivos espaciais |
| NumPy | Biblioteca para computação científica |
| SciPy | Biblioteca para computação científica |
| Matplotlib | Biblioteca para visualização de dados |
| Pandas | Biblioteca para manipulação de dados |
| GeoSeries | Extension of Pandas Series for Geospatial Data |
| GeoDataFrame | Extension of Pandas DataFrame for Geospatial Data |
| PostGIS | PostgreSQL Extension for Geographic Information Systems |
| KML | Keyhole Markup Language |
| GPKG | GeoPackage |
| ACM | Association for Computing Machinery |
| PyCX | Biblioteca para simulação e visualização de autômatos celulares |

# Sumário

# 1 Introdução

As mudanças no uso e cobertura da terra (LUCC, do inglês *Land Use and Cover Change*) impactam diretamente o clima, a biodiversidade e a sustentabilidade em diferentes escalas. Essas transformações decorrem de interações complexas entre sistemas naturais, como florestas e rios, e sistemas sociais, como agricultura e urbanização (TURNER et al., 1995). Em 2024, a Amazônia Legal registrou uma taxa de desmatamento de 6.288 km², representando uma redução de 31% em relação a 2023, a menor desde 2012 (Instituto Nacional de Pesquisas Espaciais (INPE), 2020). Para subsidiar políticas públicas e apoiar estratégias de mitigação, ferramentas computacionais capazes de modelar dinâmicas espaciais e temporais tornaram-se indispensáveis (PONTIUS et al., 2018; SAJAN et al., 2022; ROMANELLO et al., 2024).

A modelagem de LUCC é um campo interdisciplinar que utiliza paradigmas como autômatos celulares, modelos baseados em agentes e sistemas dinâmicos para representar a complexidade dos sistemas socioambientais (SAJAN et al., 2022; CARNEIRO et al., 2013). Enquanto modelos dinâmicos descrevem a evolução temporal dos sistemas, abordagens espacialmente explícitas incorporam a dimensão geográfica, considerando heterogeneidades locais, barreiras naturais e variações contextuais (CARNEIRO et al., 2013).

Para o desenvolvimento desses modelos, existem atualmente algumas plataformas que facilitam esse processo, destacando-se nacionalmente o *Dinamica EGO*, desenvolvido e mantido pela Universidade Federal de Minas Gerais, e o *TerraME*, desenvolvido pelo Instituto Nacional de Pesquisas Espaciais (INPE). Este último destaca-se como uma ferramenta robusta para modelagem de sistemas ambientais e humanos acoplados, integrando múltiplos paradigmas e o banco geoespacial *TerraLib* (CARNEIRO et al., 2013; LIMA et al., 2013). Sua extensão, o *LuccME*, fornece um arcabouço modular para modelagem de uso e cobertura da terra, com aplicações de destaque em estudos na Amazônia (AGUIAR et al., 2012; AGUIAR et al., 2016).

Apesar de sua robustez, o *TerraME* enfrenta barreiras para uma adoção mais ampla na modelagem e análise de dados espaciais pela comunidade científica. Uma possível explicação é o uso de uma linguagem de programação menos popular do que o *Python*, o que pode limitar seu alcance e a integração com outras ferramentas amplamente utilizadas. Nos últimos anos, o ecossistema *Python* evoluiu significativamente para a análise de dados, destacando-se bibliotecas como *GeoPandas* e *PySAL*, voltadas especificamente para análises geoespaciais. No entanto, sua aplicação em modelos dinâmicos espacialmente explícitos ainda é incipiente.

Esse contexto motivou o desenvolvimento da biblioteca *DisSModel* (*Discrete Spatial*

*Modeling*), criada pelo grupo LambdaGeo da Universidade Federal do Maranhão (UFMA). O *DisSModel* busca combinar acessibilidade, flexibilidade e integração com o ecossistema Python, facilitando a modelagem de LUCC com maior reprodutibilidade e alinhamento com práticas de ciência aberta.

Guiada pelas questões: (QP1) Quais dados e fontes o TerraME utiliza? (QP2) Como é estruturado seu fluxo de trabalho? (QP3) Quais são suas aplicações e impactos?, a pesquisa teve como objetivo geral desenvolver e validar o *DisSModel*, comparando suas funcionalidades às do TerraME.

A metodologia adotada compreendeu uma revisão sistemática, identificação de requisitos, desenvolvimento e implementação da biblioteca e validação por meio de estudos de caso. O *DisSModel* é distribuído como software livre e está disponível no GitHub (<https://github.com/LambdaGeo/dissmodel/>) e para testes no TestPyPI (<https://test.pypi.org/project/dissmodel/>).

Assim, a proposta responde à necessidade de superar limitações de acessibilidade, modularidade e interoperabilidade, promovendo um arcabouço mais aderente ao ecossistema Python e ao fluxo de trabalho de pesquisadores que atuam na modelagem de LUCC.

## 1.1 Objetivos Gerais e Específicos

Desenvolver um arcabouço modular e acessível, chamado DisSModel, baseado no ecossistema Python, para modelagem dinâmica espacialmente explícita de mudanças no uso e cobertura da terra (LUCC), integrando dados geoespaciais e suportando múltiplos paradigmas de modelagem (autômatos celulares, modelos baseados em agentes e sistemas dinâmicos). Para alcançar esse objetivo, os seguintes objetivos específicos foram definidos:

- Analisar as funcionalidades do TerraME e mapear bibliotecas Python para manipulação de dados geoespaciais e simulações dinâmicas;
- Projetar e implementar uma arquitetura modular para o DisSModel, com módulos *Core*, *Geo*, *Models* e *Visualization*;
- Desenvolver mecanismos para suportar múltiplos paradigmas de modelagem;
- Integrar ferramentas de visualização interativa para facilitar a análise de resultados.

## 1.2 Estrutura do Trabalho

Este trabalho está organizado em 7 capítulos, cada um abordando uma etapa essencial no desenvolvimento, implementação e validação do *framework* DisSModel,

projetado para modelagem dinâmica espacialmente explícita no ecossistema Python. A seguir, apresenta-se uma visão geral do propósito e do conteúdo de cada capítulo, destacando sua contribuição para os objetivos do estudo.

O **Capítulo 2 – Fundamentação** estabelece a base teórica do trabalho, abordando os fundamentos da modelagem de mudanças no uso e cobertura da terra (LUCC). Este capítulo explora conceitos de dados espaciais (*geo-campos* e *geo-objetos*), paradigmas de modelagem (autômatos celulares, modelos baseados em agentes e sistemas dinâmicos) e o ecossistema Python para análises geoespaciais. Além disso, apresenta uma análise do estado da arte, comparando ferramentas como o TerraME e destacando as contribuições do DisSModel em termos de acessibilidade e integração com sistemas de informação geográfica (SIG).

O **Capítulo 3 – Metodologia** detalha a abordagem metodológica adotada, estruturada em três etapas principais: (1) projeto e arquitetura do *framework*, (2) desenvolvimento e implementação em Python, e (3) comparação e validação por meio de estudos de caso. Cada etapa é guiada por questões de pesquisa que orientam o desenvolvimento do DisSModel, desde a análise do TerraME até a implementação de uma arquitetura modular e sua validação em aplicações práticas, como simulações de propagação de incêndios e padrões espaciais.

O **Capítulo 4 – Projeto e Arquitetura** descreve a concepção da arquitetura do DisSModel, com base nos requisitos identificados a partir da análise do TerraME e do ecossistema Python. Este capítulo detalha os quatro módulos principais do *framework* (*Core*, *Geo*, *Models* e *Visualization*), explicando suas funcionalidades e interações. Também apresenta a análise do ecossistema Python, destacando bibliotecas como GeoPandas, Rasterio e Salabim, que suportam a manipulação de dados espaciais, simulações dinâmicas e visualização.

O **Capítulo 5 – Implementação dos Requisitos no DisSModel** descreve a implementação prática dos requisitos do *framework*, detalhando como os módulos suportam eventos discretos, manipulação de dados espaciais, estruturas de vizinhança, modelagem orientada a objetos, visualização gráfica e suporte multiparadigma. Exemplos de código, fluxogramas e diagramas ilustram a funcionalidade de cada módulo, com foco na integração com bibliotecas Python e na capacidade de atender às demandas da comunidade científica.

O **Capítulo 6 – Resultados** apresenta a implementação de modelos representativos no DisSModel, organizados em sistemas dinâmicos (*SIR*, *PredatorPrey*, *Coffee*), autômatos celulares (*Game of Life*, *FireModelProb*, *Snow*) e modelos híbridos (*BRMangue*). Cada modelo é descrito em termos de sua formulação, implementação e resultados, com visualizações que demonstram a capacidade do *framework* de integrar dados geoespaciais e simular fenômenos complexos.

Por fim, o **Capítulo 7 – Conclusão** sintetiza os resultados alcançados, discutindo as contribuições do *DisSModel* para a modelagem dinâmica espacialmente explícita. Este capítulo avalia o impacto do *framework* em termos de acessibilidade, flexibilidade e escalabilidade, além de propor direções para trabalhos futuros, como a expansão do suporte a novos paradigmas e a integração com outras ferramentas de análise geoespacial.

Essa estrutura reflete a progressão lógica do trabalho, desde a fundamentação teórica até a implementação prática e validação, culminando na análise das contribuições e perspectivas futuras do *DisSModel*.

# 2 Fundamentação

Neste capítulo, apresentam-se de forma sucinta os principais conceitos relacionados a dados espaciais, incluindo suas representações por meio de geo-campos e geo-objetos, que constituem a base para o desenvolvimento de modelos dinâmicos espaciais. Em seguida, discutem-se aspectos fundamentais desses modelos, com ênfase nos diferentes paradigmas de modelagem adotados. Por fim, expõe-se o estado da arte, com uma análise comparativa de algumas ferramentas existentes, considerando aspectos como paradigma de modelagem, nível de integração e disponibilidade em código aberto.

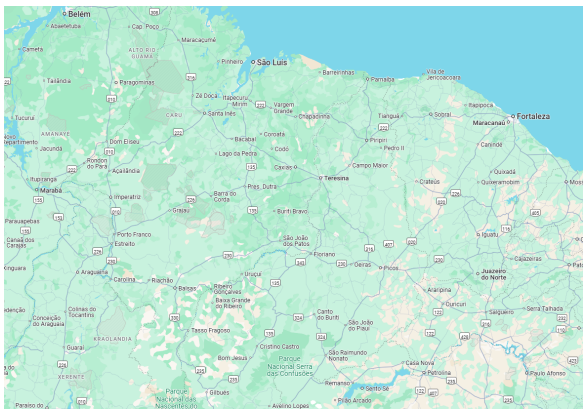
## 2.1 Dados Espaciais

Dados espaciais são fundamentais para a modelagem de mudanças no uso e cobertura da terra (LUCC), representando a superfície terrestre por meio de geo-campos e geo-objetos (GOODCHILD; LONGLEY, 2021). Geo-campos organizam o espaço em grades regulares (*rasters*), onde cada célula armazena valores categóricos ou numéricos, como em imagens de satélite (Figura 1a). Já os geo-objetos utilizam vetores para representar entidades geométricas (pontos, linhas, polígonos) com atributos associados, como em mapas geopolíticos (Figura 1b). Essas representações incorporam coordenadas para localização, permitindo o estabelecimento de relações topológicas entre unidades espaciais e a realização de diversas operações espaciais, além de possibilitar o armazenamento de múltiplos atributos por entidade.

Figura 1 – Representações de dados geográficos.


(a) Exemplo de Geo-campo (Imagem de Satélite).

Fonte: Google Earth, 2025.



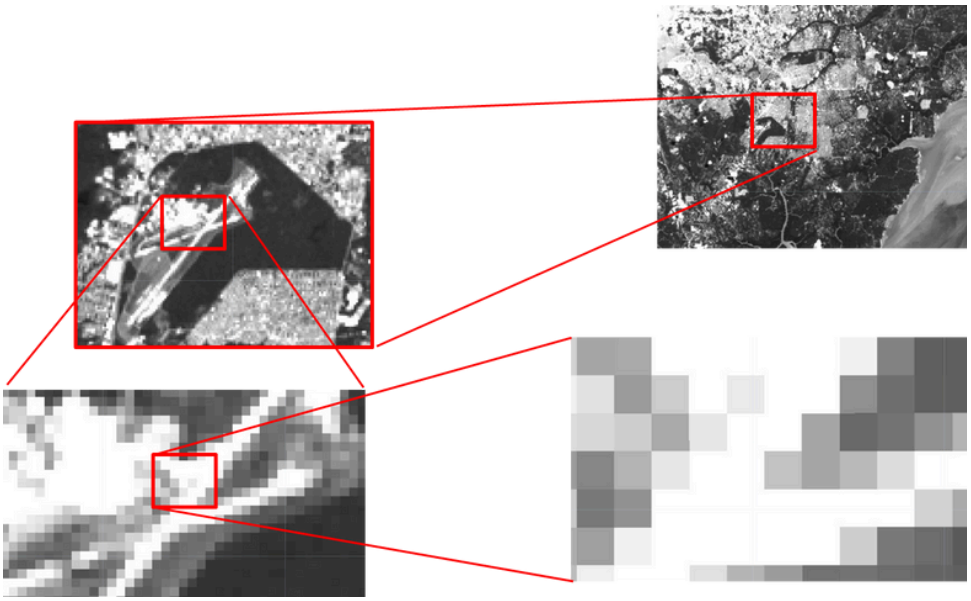

(b) Exemplo de Geo-objeto (Mapa Geopolítico).

Fonte: Google Maps, 2025.

Na Figura 2 é ilustrado a estrutura pixelizada dos geo-campos, evidenciando como

o espaço é discretizado em células regulares organizadas em matrizes, possibilitando análises em diferentes escalas até o nível do pixel. Em contrapartida, os geo-objetos são representados por tipos geométricos primitivos — pontos, linhas e polígonos — associados a dados vetoriais.

Figura 2 – Estrutura Pixelizada de Geo-campos.



Fonte: Autoria Própria.

Na Tabela 1 é resumido as principais diferenças entre os formatos *raster* e vetorial, destacando suas estruturas, características e formatos de arquivo mais comuns utilizados em aplicações de dados espaciais.

Tabela 1 – Comparação entre Formatos Raster e Vetorial.

| **Formato** | **Estrutura** | **Características** | **Formatos Comuns** |
|---|---|---|---|
| Vetorial | Pontos, linhas, polígonos | Representação de entidades geométricas discretas; alta precisão para limites; fácil atualização | .shp, .geojson, .kml, .svg |
| Raster | Pixels organizados em grades regulares | Ideal para fenômenos contínuos; depende da resolução espacial; exige maior armazenamento para alta resolução | .tif, .geotiff, .img, .asc |

Em aplicações de SIG, é fundamental que os formatos de imagem incluam informações de georreferenciamento, sendo preferíveis aqueles que preservam metadados espaciais,

como o GeoTIFF (LANG et al., 2021a).

## 2.2 Modelagem Espacial Dinâmica

A **modelagem espacial dinâmica** consolida-se como uma abordagem essencial para representar interações complexas entre sistemas naturais e sociais. Uma de suas aplicações centrais é a **modelagem de mudanças no uso e cobertura da terra** (LUCC — *Land Use and Cover Change*), cuja relevância cresce diante dos desafios contemporâneos relacionados à sustentabilidade, conservação da biodiversidade e mitigação das mudanças climáticas globais (VERBURG et al., 2006; PONTIUS et al., 2018). Essas mudanças são impulsionadas por fatores socioeconômicos e ambientais interdependentes, como expansão agrícola, urbanização, desenvolvimento de infraestrutura e políticas públicas, que geram padrões espaciais complexos, dinâmicos e frequentemente não lineares, mediados por processos de retroalimentação (LAMBIN; GEIST; LEPERS, 2003; VELDKAMP; LAMBIN, 2001). Nesse contexto, os **modelos espaciais dinâmicos** são ferramentas indispensáveis para explorar cenários futuros, avaliar impactos de políticas públicas e embasar a tomada de decisão em planejamento territorial.

De acordo com REN et al. (REN et al., 2019), os modelos podem ser classificados em um espectro conceitual que se estende desde *abordagens baseadas em padrões* (*pattern-based*) até *abordagens orientadas a processos* (*process-based*), incluindo configurações híbridas. Os modelos baseados em padrões identificam regularidades espaciais a partir de dados históricos, empregando métodos como regressão logística, redes neurais e SVM. Exemplos incluem o *Land Change Modeler (LCM)*, o *Dinamica EGO* e o *SLEUTH*. Por outro lado, os modelos orientados a processos priorizam a representação explícita de mecanismos decisórios humanos e dinâmicas biofísicas, adotando paradigmas como **Autômatos Celulares** (CA) e **Modelagem Baseada em Agentes** (ABM), como no *CLUE-S* (*Conversion of Land Use and its Effects*). Abordagens híbridas combinam esses métodos, buscando capturar múltiplas escalas de decisão e padrões emergentes.

Por sua vez, em CARNEIRO et al. é apresentado o *TerraME*, um **arcabouço multiparadigma** que integra Autômatos Celulares, ABM e **Dinâmica de Sistemas** (*System Dynamics*). Embora a Dinâmica de Sistemas não seja intrinsecamente espacial, ela pode ser incorporada como submodelo, representando estoques, fluxos e retroalimentações em processos acoplados a estruturas espaciais, como balanços hídricos e ciclos de carbono. A arquitetura modular do *TerraME*, com componentes como *Cell*, *Agent* e *Automaton*, permite a integração de paradigmas distintos em um único ambiente computacional, sendo essa flexibilidade explorada também no contexto educacional. Nesse sentido, em (ANDRADE et al., 2015) é descrito uma experiência didática que destaca a importância de articular esses paradigmas para públicos multidisciplinares, combinando *System Dynamics*,

CA e ABM de forma progressiva e auxiliando estudantes de diversas áreas a compreenderem a integração de diferentes perspectivas em modelos espaciais dinâmicos.

Portanto, a literatura evidencia que não há um paradigma único capaz de capturar integralmente a complexidade dos sistemas socioambientais, ressaltando a necessidade de **frameworks flexíveis** que combinem abordagens baseadas em padrões, processos e Dinâmica de Sistemas, com suporte explícito à representação espacial. Nesse contexto, o **framework proposto nesta pesquisa** configura-se como uma contribuição relevante, ao desenvolver uma plataforma em **Python**, linguagem reconhecida pela acessibilidade, ampla comunidade de desenvolvedores e robusto ecossistema de bibliotecas geoespaciais. Essa escolha busca facilitar a utilização por pesquisadores e estudantes, promovendo a aplicação prática e o ensino da modelagem socioambiental. A estrutura conceitual e operacional do framework será detalhada no Capítulo 4 desta monografia.

### 2.2.1 Estado da Arte

A literatura sobre modelagem espacial explícita revela avanços significativos, mas também aponta lacunas que ainda limitam a aplicação de ferramentas em contextos interdisciplinares. Ferramentas como o *Dinamica EGO*, desenvolvido pelo Centro de Sensoriamento Remoto da UFMG, destacam-se em simulações baseadas em Autômatos Celulares (CA), utilizando métodos como Pesos de Evidência para modelar processos como desmatamento e expansão urbana (VARNIER; WEBER, 2025; PINTO et al., 2025). Contudo, a ausência de suporte à Modelagem Baseada em Agentes (ABM) restringe a representação de decisões humanas mais complexas. De forma semelhante, ferramentas como *CLUE-S* e *CA_MARKOV* são eficazes na alocação espacial, mas carecem de integração robusta com ABM (MAS et al., 2014). O *TerraME*, embora versátil por integrar CA, ABM e Dinâmica de Sistemas, utiliza a linguagem *Lua*, cuja menor acessibilidade se deve a uma comunidade restrita e atualizações menos frequentes (CARNEIRO et al., 2013). No ecossistema Python, bibliotecas como *Mesa* e *PyCX* oferecem suporte a CA e ABM, mas não integram Sistemas de Informação Geográfica (SIG) de forma nativa, o que limita sua aplicação em modelagem LUCC (JORDAHL et al., 2021; HAEDRICH et al., 2023). Nesse contexto, destaca-se também o *Janus*, um pacote open-source em Python para simulações ABM de mudanças no uso e cobertura da terra, que inclui ferramentas de pré-processamento geoespacial e integração com dados como GeoTIFF e shapefiles (KAISER; FLORES; VERNON, 2020). Apesar de contribuir para ampliar o acesso a modelos ABM integrados a dados espaciais, o Janus permanece restrito a esse paradigma, sem suporte nativo a Autômatos Celulares ou Dinâmica de Sistemas, além de depender de bases de dados específicas, como o Cropland Data Layer. Assim, observa-se que, apesar dos avanços, não há ainda uma solução que una, de forma flexível, múltiplos paradigmas com suporte robusto a SIG em um ambiente de fácil acesso, evidenciando a necessidade

de frameworks mais integrados e abertos para a modelagem socioambiental.

Na Tabela 2, apresentam-se as principais ferramentas, destacando seus paradigmas de modelagem, integração com SIG e disponibilidade em código aberto, bem como suas limitações.

Tabela 2 – Comparação de Ferramentas para Modelagem LUCC.

| **Ferramenta** | **Linguagem** | **Paradigma** | **Integração GIS** | **Código Aberto** |
|---|---|---|---|---|
| LuccME | Lua | *Process-based* | Sim | Sim |
| TerraME | Lua | ABM, AC | Sim | Sim |
| Mesa | Python | ABM | Parcial | Sim |
| DynamicGrids.jl | Julia | AC | Não | Sim |
| PyCX | Python | AC, ABM | Não | Sim |
| CLUE | Não especificada | *Process-based* | Sim | Não |
| Dinamica EGO | Baseada em blocos | AC | Sim | Sim |
| Janus | Python | ABM | Sim | Sim |

As limitações identificadas reforçam a relevância do *TerraME* como uma importante referência, sobretudo por sua capacidade de integrar múltiplos paradigmas com dados geoespaciais. No entanto, a adoção do Python neste trabalho visa ampliar a acessibilidade e facilitar a integração com ferramentas modernas de análise e visualização, conforme detalhado no Capítulo 4.

# 3 Metodologia

Este trabalho propõe o desenvolvimento do *Discrete Spatial Modelling Framework* (DisSModel), voltado à modelagem dinâmica espacialmente explícita no ecossistema Python. O objetivo é oferecer uma alternativa flexível, extensível e de código aberto em relação a frameworks consolidados, como o *TerraME*, aproveitando as vantagens de acessibilidade e integração do Python com bibliotecas modernas de análise geoespacial. A metodologia adotada foi estruturada em três etapas principais: (1) projeto e definição da arquitetura do *framework*, (2) desenvolvimento e implementação em Python, e (3) comparação e validação por meio de aplicações práticas. Cada etapa foi orientada por questões de pesquisa que guiaram o delineamento do DisSModel, conforme descrito a seguir.

## 3.1 Projeto e Arquitetura do Framework

Definir os requisitos de um *framework* voltado à modelagem dinâmica espacialmente explícita exige uma pesquisa de longo prazo, envolvendo especialistas de diferentes áreas e sucessivos ciclos de refinamento e validação. O *TerraME* representa justamente o resultado desse processo colaborativo, consolidando-se como uma das principais ferramentas para o desenvolvimento de modelos espaciais dinâmicos no contexto brasileiro, especialmente entre pesquisadores do INPE. Por essa razão, este trabalho adota o *TerraME* como principal referência para a definição de requisitos e funcionalidades essenciais, buscando reinterpretar suas capacidades em uma solução flexível baseada no ecossistema Python.

- **Análise do TerraME:** Estudo detalhado da arquitetura, capacidades e fluxo de trabalho do TerraME, com foco em:
  - Suporte a paradigmas de modelagem, como eventos discretos, autômatos celulares (AC) e modelos baseados em agentes (ABM);
  - Integração com Sistemas de Informação Geográfica (SIG) via TerraLib;
  - Definição de vizinhanças e grades espaciais;
  - Visualização de resultados e exportação de mapas;
  - Modelagem orientada a objetos em Lua.

  Essa análise respondeu à questão de pesquisa QP1: *Quais dados, estruturas e métodos são adotados pelo TerraME no suporte à modelagem espacial discreta?* e forneceu a base para os requisitos do DisSModel, apresentados no Capítulo 4.

- **Mapeamento do ecossistema Python:** Identificação de bibliotecas relevantes para modelagem espacial e simulação, incluindo:
  - *GeoPandas* e *Shapely* para manipulação de dados vetoriais;
  - *Rasterio* e *Rasterstats* para processamento de dados raster;
  - *PySAL* para análises espaciais e definição de vizinhanças;
  - *Salabim* e *SimPy* para simulação de eventos discretos;
  - *Matplotlib*, *Folium* e *Streamlit* para visualização.

  Este levantamento respondeu à questão de pesquisa QP2: *Como o fluxo de trabalho do TerraME pode ser reinterpretado e adaptado em uma solução baseada exclusivamente no ecossistema Python?*, orientando a escolha de ferramentas para o DisSModel.

Os resultados desta etapa definiram a arquitetura modular do DisSModel, composta pelos módulos *Core, Geo, Models* e *Visualization*, detalhados no Capítulo 4.

## 3.2 Desenvolvimento do Framework em Python

A segunda etapa compreendeu a implementação do DisSModel, com ênfase em modularidade, flexibilidade e integração com o ecossistema Python. As principais atividades foram:

- Definição da arquitetura modular, separando camadas de dados espaciais (*Geo*), lógica de simulação (*Core* e *Models*) e visualização (*Visualization*);
- Implementação de estruturas de dados baseadas em *GeoDataFrames* para representar grades regulares e irregulares;
- Desenvolvimento de mecanismos de vizinhança, utilizando *PySAL* para estratégias de contiguidade (e.g., Moore, von Neumann);
- Criação de métodos para simulação temporal, suportando atualizações síncronas, assíncronas e baseadas em eventos discretos via *Salabim*;
- Desenvolvimento de interfaces para definição, parametrização e execução de modelos dinâmicos;
- Integração com ferramentas de visualização (*Matplotlib*, *Streamlit*) e formatos padrão de dados espaciais (e.g., *shapefiles*, GeoJSON, TIFF).

Esta etapa foi guiada pela QP2, garantindo que o DisSModel reinterpretasse o fluxo de trabalho do TerraME de forma eficiente no Python. A implementação é detalhada no Capítulo 5, com exemplos de código que ilustram a funcionalidade de cada módulo.

## 3.3 Comparação e Aplicações

A terceira etapa consistiu em demonstrar a aplicabilidade do DisSModel em relação ao TerraME, respondendo à questão de pesquisa QP3: *De que forma o arcabouço proposto contribui para democratizar e simplificar o desenvolvimento de modelos dinâmicos espacialmente explícitos?* Para isso, foram desenvolvidos diversos modelos ilustrativos no TerraME, que foram posteriormente reescritos no DisSModel. Essa estratégia não teve como objetivo realizar uma comparação formal baseada em métricas específicas, mas sim apresentar exemplos práticos que evidenciem como os principais recursos do TerraME foram reinterpretados no novo *framework*. Assim, fica a critério do leitor analisar aspectos como clareza do código e facilidade de uso, a partir dos estudos de caso disponibilizados.

As atividades realizadas nesta etapa foram:

- **Análise comparativa:** Demonstração prática do DisSModel em relação ao TerraME considerando:
  - Flexibilidade na configuração de modelos;
  - Integração com dados vetoriais e raster;
  - Suporte a múltiplos paradigmas (AC, ABM, sistemas dinâmicos);
  - Visualização e interoperabilidade com ferramentas externas;
  - Desempenho computacional em simulações.
- **Desenvolvimento de estudos de caso:** Implementação de modelos, como:
  - *FireModelProb*: Autômato celular probabilístico para propagação de incêndios florestais;
  - *Game of Life*: Simulação de padrões espaciais emergentes;
  - *SIR*: Modelo de sistemas dinâmicos para propagação de epidemias;
  - Outros exemplos, como *Predator-Prey*, *Anneal* e *Snow*.
- **Integração com bibliotecas Python:** Demonstração da interoperabilidade com *GeoPandas*, *PySAL*, *Salabim*, *Matplotlib* e *Streamlit*.

As três etapas metodológicas foram interdependentes, com os resultados de cada uma orientando a seguinte. A análise do TerraME e do ecossistema Python informou o projeto da arquitetura, que, por sua vez, guiou a implementação. A validação por meio de estudos de caso demonstrou a eficácia do DisSModel, atendendo aos objetivos de flexibilidade, escalabilidade e acessibilidade propostos.

# 4 Projeto e arquitetura

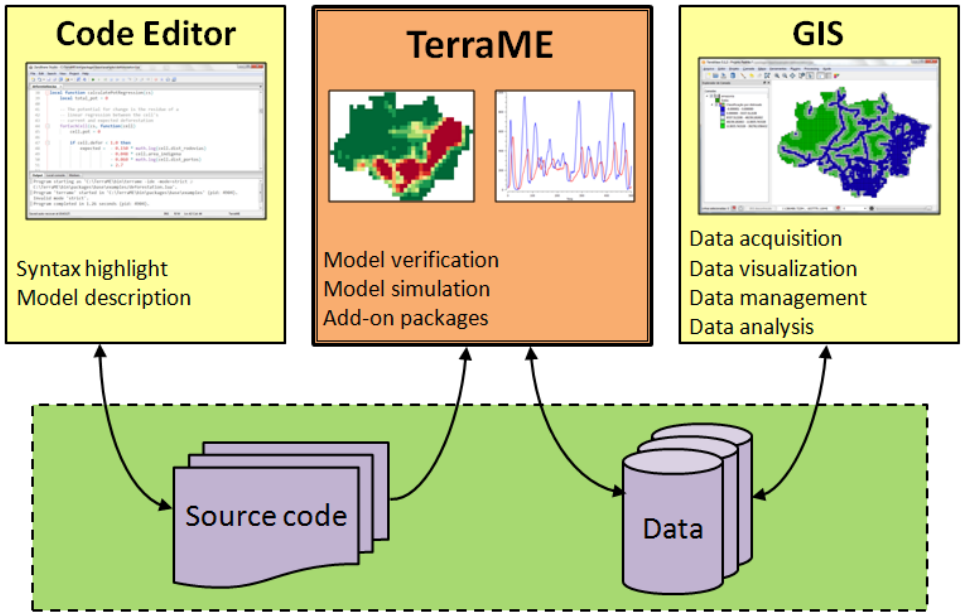
Este capítulo apresenta a arquitetura do *framework* proposto, denominado *DisSModel* (*Discrete Spatial Modeling*). Trata-se de uma plataforma voltada à modelagem dinâmica espacialmente explícita, desenvolvida no ecossistema Python e inspirada nos conceitos e funcionalidades do TerraME. A proposta busca atender aos requisitos identificados para esse tipo de modelagem, superando limitações do TerraME, como o uso da linguagem Lua[1], menos popular e com menor frequência de atualizações. A arquitetura é organizada em módulos que integram manipulação de dados geoespaciais, simulação de eventos discretos, modelagem multiparadigma e visualização, garantindo robustez e escalabilidade alinhadas às demandas da comunidade científica.

## 4.1 Identificação dos Requisitos para Modelos Dinâmicos

A modelagem dinâmica espacialmente explícita abrange uma ampla gama de paradigmas e aplicações, exigindo um *framework* genérico e robusto, capaz de sustentar diferentes abordagens científicas. Nesse contexto, o TerraME, desenvolvido pelo Instituto Nacional de Pesquisas Espaciais (INPE), foi adotado como referência neste trabalho, conforme justificado na metodologia, por ser consolidado na comunidade brasileira e integrar paradigmas como Autômatos Celulares (AC), Modelagem Baseada em Agentes (ABM) e simulação de eventos discretos (CARNEIRO et al., 2013). O TerraME utiliza a biblioteca geoespacial TerraLib para manipulação de dados vetoriais e raster e é amplamente empregado em estudos sobre dinâmica espacial, como desmatamento, mudanças no uso da terra e propagação de incêndios florestais. Na Figura 3 apresenta-se o esquema do *framework*, que ilustra a integração entre editor de código, código-fonte em Lua e o módulo TerraME, responsável por verificação, simulação e pacotes adicionais. Esse módulo interage com o Sistema de Informação Geográfica (SIG), permitindo aquisição, visualização e análise de dados espaciais, estabelecendo uma conexão fluida entre modelagem e manipulação geoespacial. Ainda assim, limitações como a menor popularidade da linguagem Lua e a ausência de suporte a longo prazo reforçaram a necessidade de desenvolver o *DisSModel* no ecossistema Python.

[1] Lua: <https://www.lua.org/>

Figura 3 – Esquema do *framework* TerraME, integrando código Lua, dados geoespaciais e simulação.




Fonte: (CARNEIRO et al., 2013).

Com base na análise do TerraME, foram definidos os requisitos essenciais para o *DisSModel*, apresentados na Tabela 3. Esses requisitos orientaram o desenvolvimento de uma arquitetura que combina flexibilidade, integração com SIG e suporte a simulações complexas, garantindo compatibilidade com as necessidades de modelagem dinâmica.

Tabela 3 – Requisitos para modelos dinâmicos no *framework DisSModel*

| Nº | Requisito | Descrição |
|---|---|---|
| (1) | Suporte a eventos discretos | Capacidade de modelar ações e processos que ocorrem em instantes específicos ou periodicamente, organizados em uma linha temporal controlada. |
| (2) | Manipulação de dados espaciais e integração com SIG | Integração nativa com dados geográficos vetoriais e raster, compatibilidade com bancos de dados espaciais e formatos comuns, permitindo carregamento e manipulação direta de dados georreferenciados. |
| (3) | Estruturas de vizinhança | Definição flexível das relações de proximidade entre células, incluindo estratégias clássicas (Moore, von Neumann) e matrizes generalizadas. |
| (4) | Modelagem orientada a objetos | Utilização de abstrações para encapsular atributos e comportamentos de entidades como agentes e autômatos. |
| (5) | Visualização gráfica | Ferramentas para exibir dados espaciais e séries temporais, com possibilidade de exportação para softwares externos. |
| (6) | Suporte multiparadigma | Combinação de diferentes paradigmas de modelagem, como autômatos celulares, agentes e eventos discretos, para representar sistemas complexos. |
| (7) | Modelagem híbrida | Capacidade de combinar componentes discretos e contínuos, ampliando o realismo das simulações socioambientais. |

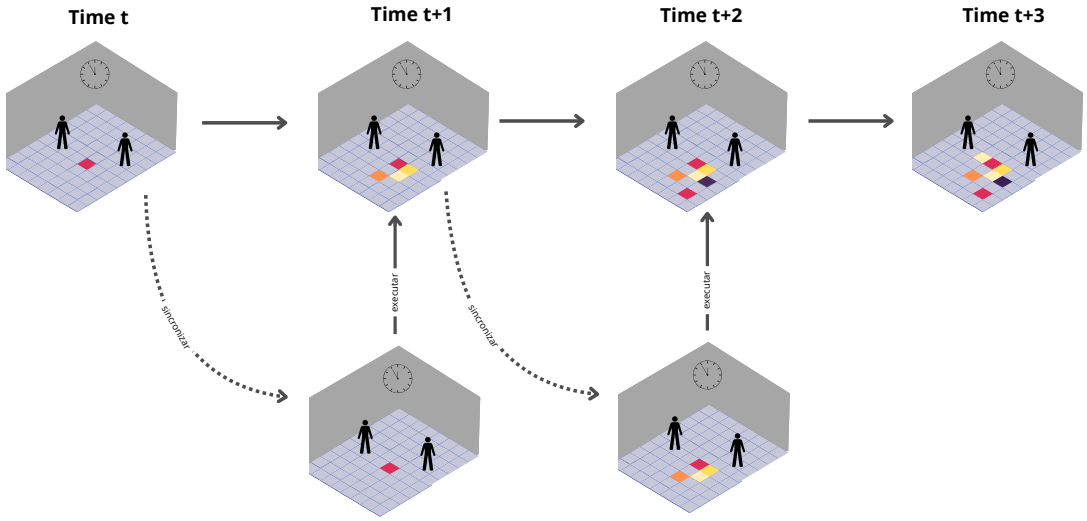
As funcionalidades do TerraME serviram como fundamento para o desenvolvimento do *DisSModel*, oferecendo uma base consolidada para modelagem dinâmica espacial. A seguir, detalham-se os principais componentes e suas aplicações, com exemplos práticos de implementação.

### 4.1.1 Modelagem baseada em eventos discretos

O TerraME suporta simulação de eventos discretos por meio dos componentes *Event*, *Timer* e *Environment*. O componente *Event* define ações que ocorrem em instantes específicos ou de forma periódica, enquanto o *Timer* gerencia a linha temporal, garantindo a execução sequencial dos eventos. O *Environment* integra esses elementos em um contexto unificado, abrangendo agentes, células espaciais e outros objetos.

Na Figura 4 é apresentado um exemplo de sincronização temporal em geo-campos, onde grades matriciais evoluem ao longo do tempo com atualizações discretas baseadas em regras locais (LANG et al., 2021b). Nesse contexto, agentes interagem com células da grade, influenciando processos espaciais dinâmicos, como mudanças no uso da terra ou propagação de eventos ambientais.

Figura 4 – Sincronização temporal em modelos espacialmente explícitos no TerraME.




Fonte: Autoria Própria.

Um exemplo prático, descrito em (CARNEIRO et al., 2013), modela o desmatamento na Amazônia entre 2000 e 2050, utilizando um *Timer* para coordenar eventos temporais. No Código 1 é apresentado um exemplo que simula a conversão de áreas de floresta em agricultura, onde um evento no tempo $t = 10$ altera o atributo *land* de células em uma grade $10 \times 10$.

**Código 1** Simulação de eventos discretos no TerraME

```
env = Environment{
    cells = CellularSpace{xdim = 10, ydim = 10} -- Grade 10x10
}
event = Event{
    time = 10, -- Ocorre no tempo t=10
    action = function()
        forEachCell(env.cells, function(cell)
            if cell.land == "floresta" then
                cell.land = "agricultura" -- Converte floresta em agricultura
            end
        end)
    end
}
timer = Timer{event}
timer:run(20)
```

### 4.1.2 Manipulação de Dados Espaciais e Integração com SIG

A manipulação de dados espaciais no TerraME é realizada pela biblioteca *TerraLib*, que suporta formatos vetoriais, como shapefiles, e raster, organizados em estruturas como *CellularSpace* e *Cell*. O *CellularSpace* representa áreas geográficas como grades regulares ou irregulares, enquanto a *Cell* armazena atributos estáticos e dinâmicos. No Código 2 é ilustrado a simulação de propagação de incêndio em uma grade de $10 \times 10$ células, onde cada célula é inicializada com vegetação do tipo "floresta" e umidade de 0.5, utilizando a vizinhança Moore. Um evento no tempo $t = 5$ altera o estado de uma célula central para "queimado", simulando o início de um incêndio.

**Código 2** Manipulação de dados espaciais no TerraME

```
cs = CellularSpace{xdim = 10, ydim = 10}
forEachCell(cs, function(cell)
    cell.vegetation = "floresta"
    cell.humidity = 0.5
end)
cs:createNeighborhood{strategy = "moore"}
env = Environment{cs}
fireEvent = Event{time = 5, action = function()
    cs:get(5, 5).vegetation = "queimado"
end}
timer = Timer{fireEvent}
timer:run(10)
```

A integração com SIG é suportada pela *TerraLib*, permitindo o uso de bancos de dados espaciais, como MySQL e PostgreSQL/PostGIS, e formatos como *shapefiles*. No Código 3 é mostrado o carregamento de um *shapefile* representando a região da Amazônia, com atributos espaciais associados a colunas e linhas, e um atributo de desmatamento ("defor_10").

**Código 3** Carregamento de shapefile no TerraME

```
amazonia = CellularSpace{
    file = filePath("amazonia.shp", "base"),
    xy = {"Col", "Lin"},
    as = {defor = "defor_10"}
}
Map{
    target = amazonia, select = "defor",
    min = 0,  max = 1,
    slices = 8, color = "RdYlGn",    invert = true
}
```

O Resultado da execução do Código 3 é ilustrado na Figura 5.

Figura 5 – Porcentagem de desmatamento na Amazônia Legal.

Fonte: (CARNEIRO et al., 2013).

### 4.1.3 Estruturas de vizinhança

O TerraME implementa relações de vizinhança por meio do método *createNeighborhood*, que suporta estratégias predefinidas, como Moore e von Neumann, ou matrizes de proximidade generalizadas (GPM). No Código 4 é apresentado a definição de uma vizinhança Moore em uma grade de $10 \times 10$ células, onde cada célula possui um atributo

de floresta inicializado com valor 1. Um evento calcula a média do atributo *forest* das células vizinhas, atualizando o atributo de desmatamento de cada célula com base nessa média (LIMA, 2010).

**Código 4** Definição de vizinhança no TerraME

```
cs = CellularSpace{xdim = 10, ydim = 10}
cs:createNeighborhood{strategy = "moore"}
forEachCell(cs, function(cell)
    cell.forest = 1
end)
event = Event{action = function()
    forEachCell(cs, function(cell)
        local forest = 0
        forEachNeighbor(cell, function(neigh)
            forest = forest + neigh.forest
        end)
        cell.deforestation = forest / 8
    end)
end}
timer = Timer{event}
timer:run(5)
```

### 4.1.4 Modelagem Orientada a Objetos

A modelagem orientada a objetos no TerraME utiliza tabelas Lua para simular encapsulamento e comportamentos de entidades, como *Agent* e *Automaton*. No Código 5 é definido uma entidade *Agent* com um atributo de energia inicializado em 100. Um método *move* reduz a energia (atributo *energy*) em 10 unidades a cada execução, desencadeada por um evento em um ambiente controlado por um *Timer* (LIMA, 2010).

**Código 5** Modelagem orientada a objetos no TerraME

```
agent = Agent{
    energy = 100,
    move = function(self)
        self.energy = self.energy - 10
    end
}
env = Environment{agent}
event = Event{action = function()
    agent:move()
end}
timer = Timer{event}
timer:run(5)
```

### 4.1.5 Visualização Gráfica

A visualização gráfica no TerraME é suportada pelos componentes *Map*, para dados espaciais, e *Chart*, para séries temporais, com possibilidade de exportação para ferramentas como TerraView. No Código 6 é exemplificado a criação de um mapa que exibe o atributo *water* de um objeto *world*, com valores entre 0 e 20, representados em uma escala de cores "Blues" dividida em cinco intervalos. Além disso, um gráfico de séries temporais exibe as variáveis *"susceptible", "infected" e "recovered"* do modelo, com cores associadas (verde, vermelho e azul, respectivamente).

**Código 6** Visualização gráfica no TerraME

```
map = Map{
    target = world,
    select = "water",
    min = 0,
    max = 20,
    color = "Blues",
    slices = 5
}
chart = Chart{
    target = model,
    select = {"susceptible", "infected", "recovered"},
    color = {"green", "red", "blue"}
}
```

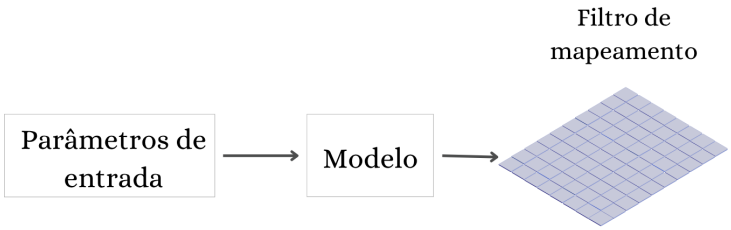
### 4.1.6 Suporte multiparadigma

O TerraME suporta modelagem multiparadigma, integrando simulação de eventos discretos, autômatos celulares e modelos baseados em agentes, o que permite a representação de sistemas complexos com interações espaciais e temporais. Essa abordagem, descrita em (CARNEIRO et al., 2013), é eficaz para modelar dinâmicas como mudanças no uso da terra, combinando diferentes escalas e tipos de processos em um ambiente unificado.

A modelagem multiparadigma no TerraME é estruturada em três paradigmas principais: (a) sistemas dinâmicos, b) autômatos celulares e (c) modelos baseados em agentes. Os sistemas dinâmicos representam estoques e fluxos, como biomassa ou taxas de processos ambientais, e utilizam relações causais e equações diferenciais para modelar retroalimentações globais (QIANG; LAM, 2015). Exemplos incluem modelos clássicos como *SIR* (para epidemias, como destacado por CHEN et al. (CHEN et al., 2020)), *Predator-Prey* (para ecologia) e *Daisyworld* (para clima), frequentemente aplicados em simulações multiescalares.

Os autômatos celulares dividem o espaço em uma grade de células que evoluem ao longo do tempo com base em regras locais, sendo adequados para simular processos espaciais como expansão urbana, propagação de incêndios ou desmatamento (RIMAL et al., 2018; SAJAN et al., 2022). Na Figura 6 é apresentado um esquema conceitual de autômatos celulares, ilustrando a organização de células em uma grade e a aplicação de regras de transição baseadas em interações com células vizinhas. Apesar de sua eficácia na representação de padrões espaciais, os autômatos celulares apresentam limitações para modelar decisões humanas ou dinâmicas sociais complexas.

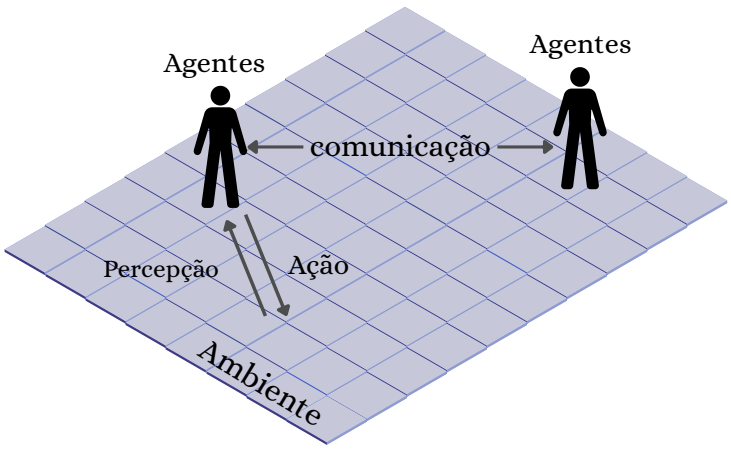
Figura 6 – Esquema conceitual de Autômatos Celulares (AC).




Fonte: Autoria Própria.

Os modelos baseados em agentes permitem simular o comportamento de atores individuais (como agricultores, gestores ou consumidores) em interação com o ambiente (COELHO, 2021). Essa abordagem captura a heterogeneidade comportamental e as decisões socioeconômicas que influenciam o uso da terra (TARIQ; MUMTAZ, 2023; XU; DU; ZHANG, 2016). Na Figura 7 é ilustrado a estrutura conceitual de modelos baseados em agentes, destacando a interação entre agentes, o sistema e o ambiente, onde os agentes tomam decisões com base em regras e condições ambientais.

Figura 7 – Estrutura conceitual de Modelos Baseados em Agentes (ABM), com Sistema, Agentes e Ambiente.




Fonte: Autoria Própria.

Modelos como *Sugarscape* e ABM teleacoplados são exemplos de aplicações que avaliam a influência de políticas públicas, mercados globais e preferências locais sobre dinâmicas espaciais complexas (DOU et al., 2019; BROWN et al., 2016).

O TerraME suporta modelos híbridos que integram autômatos celulares, agentes e componentes contínuos, permitindo a representação de sistemas socioambientais complexos. A utilização de Matrizes de Proximidade Generalizadas (GPM) possibilita modelar espaços anisotrópicos e transições graduais, como descrito em (CARNEIRO et al., 2013). Exemplos de aplicações incluem modelos como *Sugarscape* e ABM teleacoplados, que avaliam a influência de políticas públicas, mercados globais e preferências locais em dinâmicas espaciais (DOU et al., 2019; BROWN et al., 2016; LEMOS et al., 2017).

No Código 7 é exemplificado a simulação de conversão de áreas de floresta em agricultura em um *CellularSpace* de 25×25 km² ao longo de 20 períodos. O código define um ambiente com uma grade de células, onde um evento no tempo t=10 altera o atributo *"land"* de células com valor *"floresta"* para *"agricultura"*, simulando uma transição de uso da terra.

**Código 7** Código em Lua para simulação de mudança de uso da terra

```
env = Environment{cells = CellularSpace{xdim = 25, ydim = 25}}
event = Event{time = 10, action = function()
    forEachCell(env.cells, function(cell)
        if cell.land == "floresta" then
            cell.land = "agricultura"
        end
    end)
end}
timer = Timer{event}
timer:run(20)
```

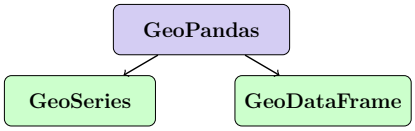
## 4.2 Análise do Ecossistema Python para dados espaciais

O Python consolidou-se como uma das linguagens mais utilizadas em ciência de dados, destacando-se pela flexibilidade, reprodutibilidade e vasta comunidade de desenvolvedores. Nos últimos anos, o ecossistema Python evoluiu significativamente para oferecer suporte a dados geoespaciais, cobrindo desde operações básicas até simulações complexas em ambientes espaciais dinâmicos. Este levantamento está estruturado em três dimensões principais: (i) representação e manipulação de dados vetoriais e matriciais, (ii) visualização de dados espaciais e (iii) suporte à simulação.

### 4.2.1 Representação e Manipulação de Dados Vetoriais e Matriciais

O ecossistema Python disponibiliza diversas bibliotecas para o tratamento de dados espaciais. A biblioteca **GeoPandas**, uma extensão do *pandas*, é amplamente utilizada para manipulação de dados vetoriais, suportando operações como interseção, união e buffer, além de formatos como *shapefiles*, GeoJSON e PostGIS (JORDAHL et al., 2021). A estrutura *GeoDataFrame* integra dados tabulares com geometrias espaciais por meio da *GeoSeries*, que inclui um atributo de Sistema de Referência de Coordenadas (CRS) para gerenciar projeções. Na Figura 8 é ilustrado a relação entre *GeoPandas*, *GeoSeries* e *GeoDataFrame*, destacando a hierarquia e dependência entre essas estruturas.

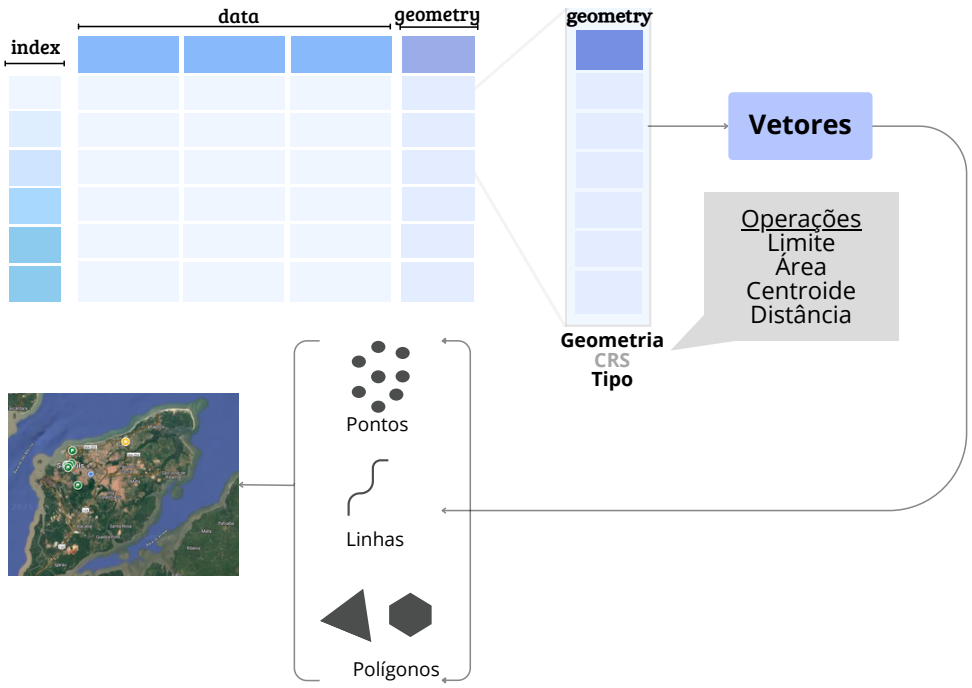
Figura 8 – Relação entre *GeoPandas*, *GeoSeries* e *GeoDataFrame*.




Fonte: Adaptado de (CÂMARA et al., 2001).

Na Figura 9 é apresentado a estrutura de um *GeoDataFrame*, que organiza informações tabulares associadas a geometrias espaciais, com uma coluna ativa de geometria acessível pelo atributo *geometry*. Essa organização facilita análises espaciais avançadas (JORDAHL et al., 2021).

Figura 9 – Estrutura de um *GeoDataFrame* no *GeoPandas*.




Fonte: Autoria Própria.

O *GeoPandas* integra bibliotecas complementares para manipulação de geometrias, incluindo **Shapely** (operações geométricas como união e interseção), **Fiona** (leitura e escrita de formatos geoespaciais) e **Pyproj** (gerenciamento de projeções e transformações de coordenadas) (JORDAHL et al., 2021). Para dados raster, a biblioteca **Rasterio** suporta formatos como GeoTIFF, oferecendo funcionalidades para leitura, escrita e transformações (LANG et al., 2021a). A biblioteca **Xarray** facilita o trabalho com dados raster multidimensionais, como séries temporais de imagens de satélite, enquanto **GDAL** fornece ferramentas robustas para manipulação de dados raster e vetoriais (LANG et al., 2021a). Essas bibliotecas, combinadas, formam um ecossistema abrangente para modelagem geoespacial, suportando análises estáticas e dinâmicas (LANG et al., 2021a; ROJAS et al., 2023).

### 4.2.2 Visualização de Dados Espaciais

O ecossistema Python oferece ferramentas para visualização espacial, com destaque para o **GeoPandas**, que utiliza o *Matplotlib* para gerar mapas estáticos a partir de um *GeoDataFrame*. Bibliotecas como **Folium** e **Plotly** permitem a criação de mapas interativos e gráficos dinâmicos, facilitando a comunicação de resultados e a exploração de padrões espaciais (ROJAS et al., 2023). Essas ferramentas complementam a análise geoespacial, oferecendo opções para visualização em diferentes contextos e formatos.

### 4.2.3 Suporte à Simulação Baseada em Eventos Discretos

A simulação de eventos discretos no Python é suportada por bibliotecas como **SimPy** (v4.1.2, 2025), que oferece execução paralela, e **Salabim** (v24.0.1, 2025), que proporciona animações interativas. No Código 8 é apresentado um exemplo de modelagem com *Salabim*, onde uma classe *Agent* é definida com um atributo de energia inicializado no método *setup*. O método *process* reduz a energia em uma unidade a cada intervalo de tempo, cancelando o agente quando a energia atinge zero. Essa abordagem permite simular dinâmicas temporais de forma orientada a objetos.

**Código 8** Modelagem orientada a objetos com *Salabim*

```
import salabim as sim

class Agent(sim.Component):
    def setup(self, energy):
        self.energy = energy
    def process(self):
        while True:
            self.energy -= 1
            self.hold(1)
            if self.energy <= 0:
                self.cancel()
```

A integração dessas bibliotecas com ferramentas de manipulação de dados espaciais viabiliza simulações que combinam processos dinâmicos e dados geoespaciais, atendendo aos requisitos de modelagem complexa (LANG et al., 2021a).

### 4.2.4 Síntese das Principais Bibliotecas

Na Tabela 4 são resumidas as funcionalidades das principais bibliotecas Python utilizadas no *DisSModel*, incluindo suas aplicações, dependências e limitações. Nessa Tabela é organizado as bibliotecas por suas funções principais, destacando a complementariedade entre ferramentas de manipulação de dados, análise estatística e simulação.

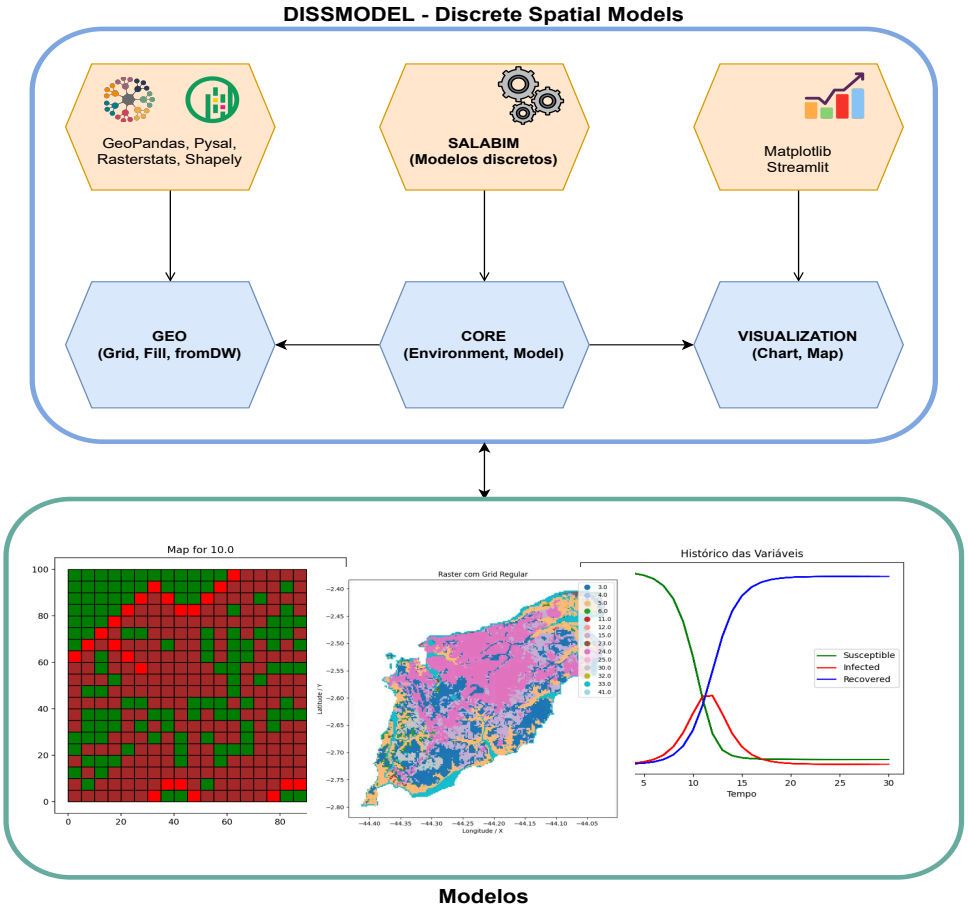
Tabela 4 – Funcionalidades das bibliotecas Python para modelagem espacial.

| **Biblioteca** | **Funcionalidade Principal** | **Aplicação** | **Dependências** | **Limitações** |
|---|---|---|---|---|
| GeoPandas | Manipulação de dados vetoriais | Operações geométricas e análise espacial | Shapely, Fiona, Pyproj | Sem suporte nativo a simulações |
| Rasterio | Processamento de dados raster | Leitura, escrita e transformação de imagens | GDAL | Foco em operações raster estáticas |
| PySAL | Análise estatística espacial | Autocorrelação, regressão e vizinhança | NumPy, SciPy | Não realiza simulação |
| Salabim | Simulação de eventos discretos | Modelagem temporal dinâmica | - | Pouco integrado a dados espaciais |
| Rasterstats | Estatísticas zonais em raster | Cálculo de métricas em geometrias vetoriais | Rasterio, NumPy | Foco em análise estática |

## 4.3 Arquitetura do DisSModel

A arquitetura do *DisSModel* é composta por quatro módulos principais: *Core*, *Models*, *Geo* e *Visualization*, que interagem para suportar modelagem dinâmica, análise espacial e visualização de resultados. Na Figura 10 é ilustrado a organização dos módulos e o fluxo de dados entre eles, destacando a integração entre simulação, manipulação de dados espaciais e visualização.

Figura 10 – Arquitetura do DisSModel, com módulos *Core, Models, Geo e Visualization* e fluxo de dados.




Fonte: Autoria Própria.

O módulo *Core* estabelece a base para as simulações, utilizando a classe *Environment* (herdada de *salabim.Environment*) para gerenciar a linha temporal e a classe *Model* (herdada de *salabim.Component*) para definir comportamentos fundamentais de todos os modelos. O módulo *Geo* suporta autômatos celulares por meio da classe *CellularAutomaton*, que estende *Model*, e oferece funções utilitárias como *regular_grid* (cria grades regulares), *fill* (inicializa atributos de células) e *Neighborhood* (define relações de vizinhança, como Moore ou von Neumann). O módulo *Models* organiza subpacotes, incluindo *ca* (com modelos como *GameOfLife*, *FireModelProb* e *Snow*) e *sysdyn* (com modelos como *SIR*,

*PredatorPrey* e *Daisyworld*), todos derivados das classes correspondentes do módulo *Core*. O módulo *Visualization* fornece ferramentas para geração de gráficos temporais (*Chart*), mapas espaciais (*Map*) e interfaces interativas por meio da biblioteca *Streamlit* (função *display_inputs*).

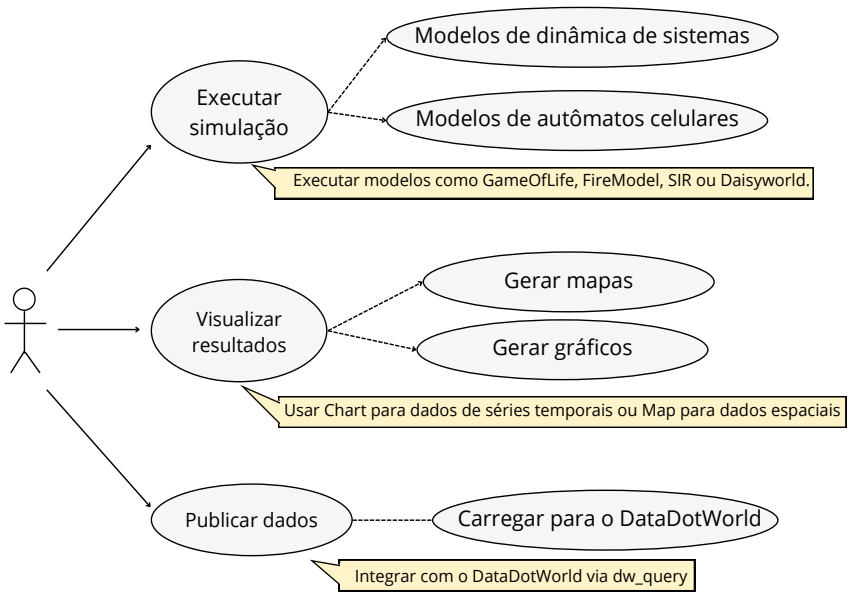
Na tabela 5 são resumidas as funcionalidades dos módulos principais do *framework* DisSModel, destacando suas responsabilidades e características.

Tabela 5 – Módulos principais do *framework* DisSModel e suas funcionalidades.

| **Módulo** | **Descrição** |
|---|---|
| Core | Fornece a base para simulações, com a classe *Environment* (herdada de *salabim.Environment*) para gerenciamento temporal e a classe *Model* (herdada de *salabim.Component*) para definição de comportamentos. |
| Geo | Suporta autômatos celulares via *CellularAutomaton*, com funções como *regular_grid*, *fill* e *Neighborhood*. |
| Models | Organiza subpacotes como *ca* (GameOfLife, FireModelProb, Snow) e *sysdyn* (*SIR, PredatorPrey, Daisyworld*). |
| Visualization | Oferece gráficos temporais (*Chart*), mapas espaciais (*Map*) e interfaces interativas via Streamlit. |

Na Figura 11 é apresentado o diagrama de casos de uso do *DisSModel*, que ilustra as interações principais do usuário com o *framework*, incluindo a execução de simulações, visualização de resultados (gráficos e mapas), configuração de parâmetros por meio de interfaces interativas e publicação de dados no *DataWorld*.

Figura 11 – Diagrama de casos de uso do DisSModel, com interações do usuário.




Fonte: Autoria Própria.

Assim, este capítulo apresentou a organização conceitual e os principais módulos que estruturam o *DisSModel*, evidenciando suas responsabilidades e funcionalidades. No capítulo seguinte, é detalhada a implementação dessa arquitetura, ilustrando como cada componente foi desenvolvido e integrado no ecossistema Python.

# 5 Implementação dos Requisitos no DisSModel

Este capítulo descreve a implementação dos requisitos identificados no *framework* DisSModel, projetado para modelagem dinâmica espacialmente explícita no ecossistema Python, conforme apresentado no Capítulo 4. Cada seção aborda um requisito (ou recurso) específico, detalhando como o *framework* suporta suas funcionalidades por meio de módulos, classes e funções. Exemplos de Código, Figuras e Tabelas ilustram as implementações, destacando a integração com bibliotecas Python e a capacidade de atender às necessidades da comunidade científica.

## 5.1 Modelagem baseada em eventos discretos

O suporte a eventos discretos, conforme definido na Tabela 3 do Capítulo 4, é implementado pelo módulo *Core* do *DisSModel*, que gerencia eventos discretos e a execução das simulações, utilizando a biblioteca Salabim (v24.0.1, 2025) (LANG et al., 2021c). A classe *Environment*, herdada de *salabim.Environment*, gerencia a linha temporal, definindo tempos inicial (*start_time*) e final (*end_time*) (CHIARADONNA; JEVTIć; STERNER, 2024). A classe *Model*, herdada de *salabim.Component*, define comportamentos dinâmicos dos modelos (CORDEIRO; PITOMBEIRA-NETO, 2023). No Código 9 é exemplificado a configuração de uma simulação simples, onde um modelo imprime o tempo atual a cada passo.

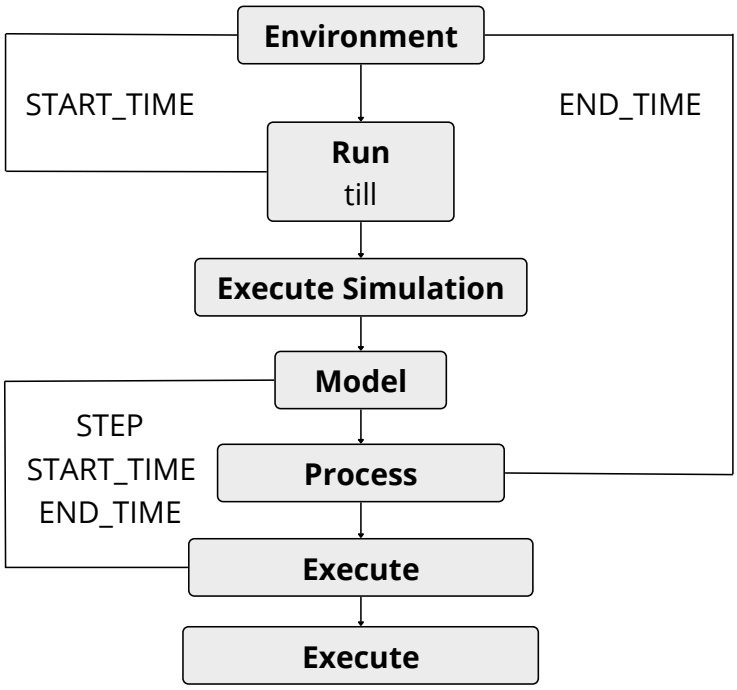
**Código 9** Configuração de simulação no módulo *core*

```
from dissmodel.core import Environment, Model

class SimpleModel(Model):
    def execute(self):
        print(f"Tempo: {self.env.now()}")

env = Environment(start_time=2010, end_time=2016)
model = SimpleModel()
env.run()
```

Na Figura 12 é apresentado o fluxograma do processo de simulação, ilustrando a sequência de operações da estrutura central do ambiente de simulação e o processo de execução: inicialização do ambiente, execução do modelo e iteração até o tempo final.

O fluxograma destaca a interação entre *Environment* e *Model*, enfatizando o controle temporal proporcionado pela biblioteca Salabim.

Figura 12 – Fluxograma do Fluxo de Simulação




Fonte: Autoria Própria.

## 5.2 Manipulação de Dados Espaciais e Integração com SIG

A manipulação de dados espaciais e a integração com Sistemas de Informação Geográfica (SIG) são suportadas pelo módulo *Geo*, que utiliza a biblioteca GeoPandas (JORDAHL et al., 2021). O *GeoDataFrame* permite carregar e manipular dados espaciais reais (e.g., *shapefiles*, GeoJSON) e integrá-los com simulações dinâmicas, como entrada e saída do modelo, facilitando análises espaciais visuais e quantitativas. No Código 10 é ilustrado o carregamento de um *shapefile* como *GeoDataFrame*, contendo geometrias e atributos como elevação (*Alt2*).

**Código 10** Leitura de um *shapefile* como *GeoDataFrame*.

```
1 import geopandas as gpd
2
3 gdf = gpd.read_file("filename.shp")
```

No Código 11 é apresentado um modelo que simula o aumento do nível do mar, incrementando o atributo de elevação (*Alt2*) e calculando sua média a cada passo de

tempo. A função *@track_plot* gera visualizações automáticas da média de elevação.

**Código 11** Simulação de aumento do nível do mar.

```
@track_plot("media_altitude", "blue")
class ElevacaoSimples(Model):
    seaLevelRiseRate: float
    media_altitude: float
    def setup(self, gdf, seaLevelRiseRate=0.01):
        self.gdf = gdf
        self.seaLevelRiseRate = seaLevelRiseRate
        self.media_altitude = 0
    def execute(self):
        self.gdf["Alt2"] += self.seaLevelRiseRate
        self.media_altitude = self.gdf["Alt2"].mean()
```

O módulo *Geo* implementa funções utilitárias, como *regular_grid*, que cria grades regulares vetoriais para simulações espaciais. No Código 12 demonstra-se três modos de geração de grades: a partir de um *GeoDataFrame*, limites espaciais ou dimensões específicas, a partir de um GeoDataFrame existente, limites manuais, dimensão e resolução (sem localização geográfica).

**Código 12** Exemplos de criação de grades regulares com a função *regular_grid*.

```
from dissmodel.geo import regular_grid
grid_from_gdf = regular_grid(
    gdf=meu_gdf,           # GeoDataFrame de referência
    resolution=100         # Tamanho da célula (ex.: metros)
)
grid_from_bounds = regular_grid(
    bounds=(0, 0, 1000, 1000),  # xmin, ymin, xmax, ymax
    resolution=100              # Células de 100x100 unidades
)
grid_from_dimension = regular_grid(
    dimension=(5, 4),      # 5 colunas x 4 linhas
    resolution=50          # Células de 50x50 unidades
)
```

Para o preenchimento de atributos espaciais, o método *fill* do módulo *Geo* permite associar dados contínuos, pontuais ou sintéticos às células de uma grade espacial. Esse recurso é central para a integração de dados geográficos reais, como camadas vetoriais e rasters, ou para a geração de dados sintéticos em grades regulares, quando se deseja construir modelos exploratórios controlados.

As estratégias de preenchimento abrangem tanto métodos baseados em dados de entrada (estatísticas zonais, distâncias mínimas) quanto métodos de amostragem ou padrões pré-definidos. Dessa forma, é possível, por exemplo, inicializar atributos a partir de mapas reais de elevação, uso da terra ou proximidade a feições geográficas, ou ainda criar configurações artificiais para simular fenômenos como propagação de fogo e o *Game of Life*.

Usando dados de entradas, é possível usar as estatísticas zonais que permitem integrar dados raster com dados vetoriais, nesse caso, com as grades regulares. Cada célula da grade (vetorial) é associada a valores extraídos de um raster (por exemplo, elevação ou temperatura), sintetizados por estatísticas como média, mínimo e máximo. Esse processo permite caracterizar cada zona (célula) com informações agregadas da superfície contínua.

No Código 13 é ilustrado o preenchimento com estatísticas zonais, agregando valores de elevação de um raster, em que cada célula recebe os valores agregados de elevação contidos em seu polígono.

**Código 13** Preenchimento com estatísticas zonais.

```
fill(
    strategy=FillStrategy.ZONAL_STATS,
    vectors=grade,
    raster_data=raster,
    affine=affine,
    stats=["mean", "min", "max"],
    prefix="alt_"
)
```

Existe ainda cenários que os modeladores precisam incluir informações oriundas de outros dados vetoriais, como os cálculos de distâncias mínimas para cada célula. Permitindo calcular a menor distância euclidiana até feições como rios, estradas ou centros urbanos. No Código 14 é calculado a distância mínima entre células e feições geográficas, como rios, atribuindo o resultado ao atributo *dist_rio*.

**Código 14** Preenchimento com distância mínima.

```
fill(
    strategy=FillStrategy.MIN_DISTANCE,
    from_gdf=grade,
    to_gdf=rios,
    attr_name="dist_rio"
)
```

Esse procedimento é especialmente útil para modelar influências espaciais ou restrições baseadas em proximidade.

Na construção de cenários hipotéticos, por exemplo, na simulação da ocupação inicial de uma área, é comum o uso de estrategias com a amostragem aleatória. Esse algoritmo gera então atributos sintéticos a partir de uma distribuição pré-definida. No Código 15 é utilizado a amostragem aleatória para inicializar ocupação em 70% das células com valor 0 e 30% com valor 1.

**Código 15** Preenchimento com amostragem aleatória.

```
fill(
    strategy=FillStrategy.RANDOM_SAMPLE,
    gdf=grade,
    attr="ocupacao",
    data={0:0.7, 1:0.3},
    seed=42
)
```

Além da amostragem aleatória, inclui-se também a possibilidade de definir padrões fixos sobre a grade permite definir arranjos espaciais controlados, facilitando testes de modelos e simulações sintéticas como autômatos celulares (por exemplo, o Game of Life) ou modelos de propagação de incêndios. No Código 16 é aplicado um padrão fixo triangular às células, útil para simulações como o *GameOfLife*.

**Código 16** Preenchimento com padrão fixo.

```
fill(
    strategy=FillStrategy.PATTERN,
    gdf=grade,
    attr="tipo",
    pattern=[[1,0,0], [0,1,0],  [0,0,1] ],
    start_x=0, start_y=0
)
```

Este exemplo aplica um padrão triangular sobre a grade, inicializando as células com valores fixos que se repetem segundo o arranjo definido.

Em síntese, o módulo *fill* oferece flexibilidade tanto para integrar dados geográficos reais (por meio de estatísticas zonais e distâncias) quanto para gerar dados sintéticos controlados (por amostragem aleatória ou padrões), o que é essencial para experimentação, validação e demonstração de modelos dinâmicos espaciais.

Na Tabela 6 resume-se as estratégias de preenchimento suportadas pelo módulo *Geo*, detalhando suas aplicações e exemplos.

Tabela 6 – Estratégias de preenchimento de atributos no módulo Geo.

| **Estratégia** | **Descrição** | **Exemplo** |
|---|---|---|
| *ZONAL_STATS* | Agrega valores de um raster (e.g., elevação) em células vetoriais usando estatísticas como média, mínimo e máximo. | Código 13 |
| *MIN_DISTANCE* | Calcula a distância euclidiana mínima entre células e feições geográficas (e.g., rios). | Código 14 |
| *RANDOM_SAMPLE* | Inicializa atributos com valores aleatórios com base em uma distribuição definida. | Código 15 |
| *PATTERN* | Aplica um padrão fixo às células, útil para simulações controladas como *GameOfLife*. | Código 16 |

## 5.3 Estruturas de Vizinhança

O módulo *Geo* suporta estruturas de vizinhança, como Moore, von Neumann e Queen, para modelar interações locais entre células. A função *Neighborhood* define relações de proximidade, permitindo configurações personalizadas. No Código 17 é exemplificado a criação de uma vizinhança Moore em uma grade, onde cada célula atualiza seu atributo com base na média dos vizinhos.

**Código 17** Definição de vizinhança Moore.

```
from dissmodel.geo import Neighborhood

grid = regular_grid(dimension=(10, 10), resolution=100)
grid = fill(gdf=grid, strategy=FillStrategy.RANDOM_SAMPLE, attr="value",
↪  data={0: 0.5, 1: 0.5})
neighborhood = Neighborhood(grid, strategy="moore")

for cell in grid.itertuples():
    neighbors = neighborhood.get_neighbors(cell.Index)
    cell.value = sum(n.value for n in neighbors) / len(neighbors)
```

## 5.4 Modelagem orientada a objetos

A modelagem orientada a objetos é implementada por meio das classes *Model* e *CellularAutomaton*, que encapsulam atributos e comportamentos. No Código 18 é definido um agente com um atributo de energia que diminui a cada passo, demonstrando encapsulamento e comportamento dinâmico.

**Código 18** Modelagem orientada a objetos.

```
from dissmodel.core import Model
import salabim as sim

class Agent(Model):
    def setup(self, energy):
        self.energy = energy

    def execute(self):
        self.energy -= 1
        if self.energy <= 0:
            self.cancel()
```

## 5.5 Visualização de modelos espaciais

O módulo *Visualization* suporta a geração de mapas e gráficos temporais usando *Matplotlib* e *Streamlit*. No Código 19 é ilustrado a criação de um mapa a partir de um *GeoDataFrame* e um gráfico de séries temporais para variáveis do modelo SIR.

**Código 19** Visualização de mapas e séries temporais.

```
from dissmodel.visualization import Map, Chart

map = Map(
    target=gdf,
    select="value",
    min=0,
    max=1,
    color="Blues",
    slices=5
)

chart = Chart(
    target=model,
    select=["susceptible", "infected", "recovered"],
    color=["green", "red", "blue"]
)
```

## 5.6 Suporte multiparadigma

O DisSModel suporta modelagem multiparadigma, integrando AC, ABM e eventos discretos (DES). O módulo *Models* organiza subpacotes como *ca* (e.g., *GameOfLife*) e

*sysdyn* (e.g., *SIR*). No Código 20 é implementado o *GameOfLife*, um autômato celular clássico, utilizando a classe *CellularAutomaton*.

**Código 20** Implementação do GameOfLife.

```
from dissmodel.geo import CellularAutomaton, regular_grid, fill, FillStrategy

grid = regular_grid(dimension=(10, 10), resolution=100)
grid = fill(gdf=grid, strategy=FillStrategy.RANDOM_SAMPLE, attr="state",
↪  data={0: 0.7, 1: 0.3})

class GameOfLife(CellularAutomaton):
    def execute(self):
        new_states = []
        for cell in self.gdf.itertuples():
            neighbors = self.neighborhood.get_neighbors(cell.Index)
            live_neighbors = sum(n.state for n in neighbors)
            state = 1 if (cell.state == 1 and 2 <= live_neighbors <= 3) or
            ↪  (cell.state == 0 and live_neighbors == 3) else 0
            new_states.append(state)
        self.gdf["state"] = new_states
```

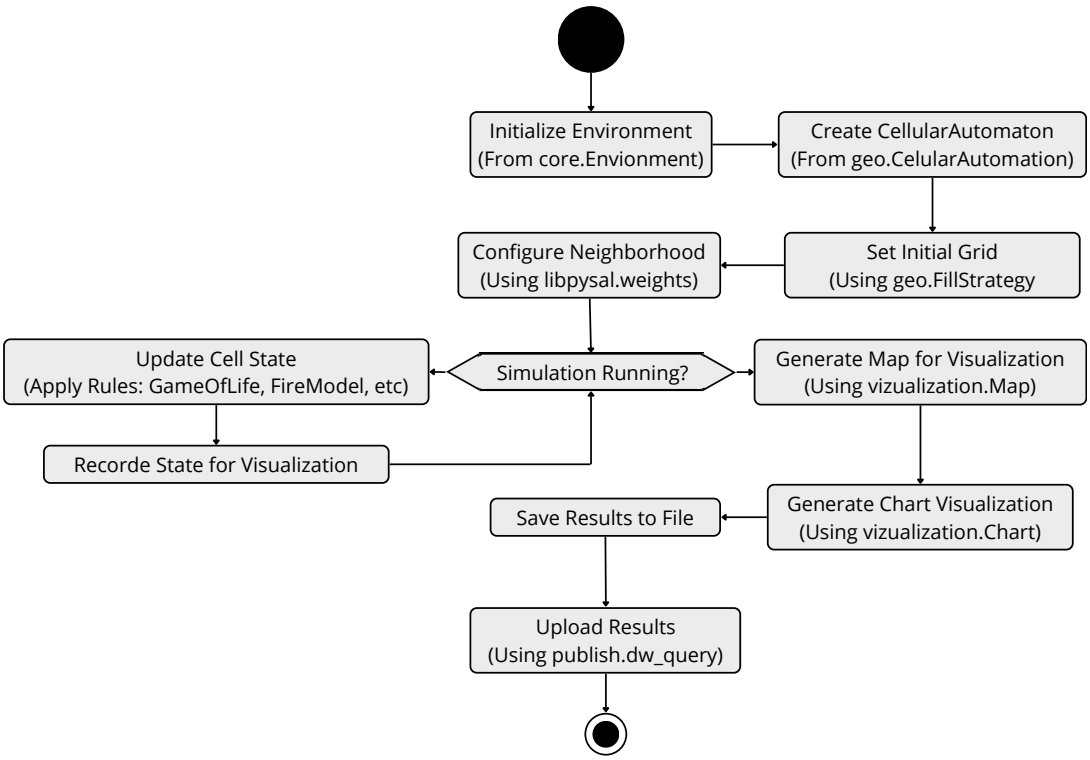
A classe *CellularAutomaton* utiliza *GeoDataFrame* para armazenar geometrias e estados, com métodos como *create_neighborhood* e *rule*, e herda da classe *Model* do módulo *Core*, permitindo integração com simulações temporais discretas.

Os principais métodos da classe *CellularAutomaton* são:

- *initialize()*: Método abstrato para definir a configuração inicial dos estados das células, a ser implementado pelas subclasses.
- *create_neighborhood(strategy, neighbors_dict, *args, **kwargs)*: Cria uma vizinhança para as células, utilizando estratégias como *Queen* ou *Rook* da biblioteca *libpysal*, ou um dicionário de vizinhanças pré-definido, anexado ao *GeoDataFrame* na coluna *_neighs*.
- *neighs_id(idx)*: Retorna os índices das células vizinhas para uma célula de índice *idx*.
- *neighs(idx)*: Retorna as células vizinhas, lançando erro se a vizinhança não foi criada.
- *rule(idx)*: Método abstrato que define a regra de transição de estado para a célula *idx*, a ser implementado pelas subclasses.
- *execute()*: Aplica a regra de transição a todas as células, atualizando o atributo de estado no *GeoDataFrame*.

O fluxo de execução de uma simulação com a classe *CellularAutomaton* é ilustrado na Figura 13, que apresenta o diagrama de atividades detalhando a inicialização do ambiente, configuração da grade e vizinhança, aplicação iterativa das regras de transição e atualização das visualizações.

Figura 13 – Diagrama de atividades do *framework DisSModel* para simulações com autômatos celulares.




Fonte: Autoria própria.

Na Seção 6.4 será apresentado alguns exemplos de modelos de autômatos celulares que utilizam esta classe.

# 6 Resultados

Este capítulo apresenta os resultados do desenvolvimento e implementação dos modelos e paradigmas suportados pelo *framework* DisSModel, projetado para a modelagem dinâmica espacialmente explícita, conforme descrito nos Capítulos 4 e 5.

Os modelos e exemplos apresentados nesta seção, como *Game of Life*, *FireModelProb*, *Snow*, entre outros, são implementações de exemplo disponíveis no site oficial do TerraME. Especificamente, os modelos de autômatos celulares estão disponíveis no repositório em <https://www.terrame.org/package/ca/>, enquanto os modelos de sistemas dinâmicos podem ser encontrados em <https://www.terrame.org/package/sysdyn/>. Esses modelos servem como demonstrações práticas das capacidades do *framework* e foram adaptados para o DisSModel, destacando a interoperabilidade e as melhorias propostas em relação ao TerraME.

São apresentados exemplos práticos que demonstram a integração de sistemas dinâmicos, autômatos celulares (AC) e modelos híbridos ao *framework*, destacando suas funcionalidades de simulação, manipulação de dados geoespaciais e visualização. As seções subsequentes detalham casos de uso representativos, organizados por paradigma, destacando a integração com os módulos *Core*, *Geo*, *Models* e *Visualization*.

O código-fonte dos modelos implementados no *DisSModel* é comparado com suas versões equivalentes no *framework* Terrame, no qual o *DisSModel* se baseia, para destacar semelhanças, melhorias e adaptações realizadas.

Para contextualizar a execução prática dos modelos, descreve-se inicialmente as formas de execução suportadas pelo *DisSModel*, evidenciando sua flexibilidade e adequação a diferentes perfis de usuários e cenários de aplicação. Cada modelo é ilustrado com tabelas, trechos de código e figuras explicativas, com referência à Tabela 5 do Capítulo 4 para contextualização dos módulos. Assim, este capítulo consolida os resultados, demonstrando a viabilidade e os avanços do *DisSModel* em relação ao TerraME para diferentes abordagens de modelagem.

## 6.1 Executando os modelos

Esta seção descreve a estrutura de execução dos modelos desenvolvidos no *framework* *DisSModel*, projetada para atender a diferentes perfis de usuários e contextos de aplicação. Cada paradigma de modelagem – sistemas dinâmicos, autômatos celulares e modelos híbridos – possui exemplos organizados em três abordagens principais: **linha de comando (*CLI*), *notebooks* interativos** e **aplicações web com *Streamlit***. Essa organização

modular promove reprodutibilidade, exploração incremental e disseminação de experimentos em ambientes acadêmicos ou profissionais.

### 6.1.1 Estrutura de Diretórios

Na Tabela 7 é ilustrado a estrutura típica de diretórios para cada paradigma:

Tabela 7 – Estrutura de diretórios dos exemplos por paradigma

| **Diretório Base** | **Subpastas** |
|---|---|
| `examples/sysdyn/` | `cli/`, `notebook/`, `streamlit/` |
| `examples/geo/` | `cli/`, `notebook/`, `streamlit/` |
| `examples/cellular/` | `cli/`, `notebook/`, `streamlit/` |

### 6.1.2 Execução via Linha de Comando (CLI)

A execução em *Command Line Interface* é indicada para experimentos em lote, automação de cenários ou execução em servidores. O Código 21 mostra um exemplo de *script* para execução do modelo *SIR*.

**Código 21** Configuração de simulação do modelo *SIR* via *CLI*

```
from dissmodel.models.sysdyn import SIR
from dissmodel import Environment, Chart

env = Environment()

SIR(
    susceptible=9998, infected=2, recovered=0,
    duration=2, contacts=6, probability=0.25
)

Chart(show_legend=True)
env.run(30)
```

A execução deste *script* é realizada pelo comando:

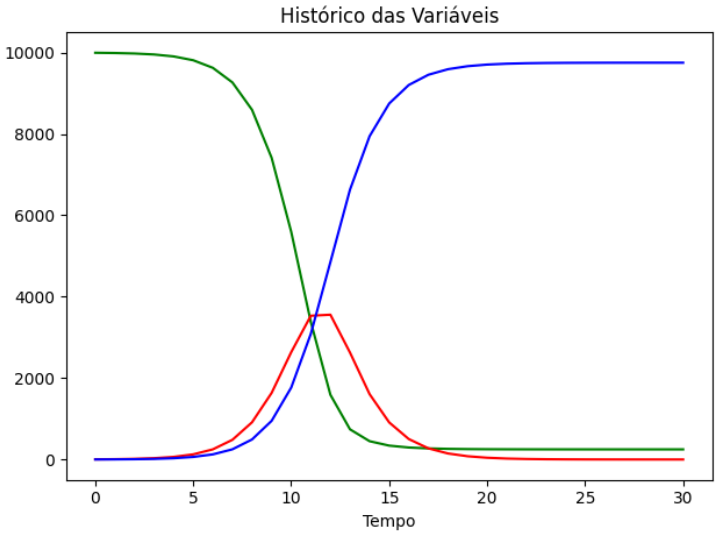
```
python examples/sysdyn/cli/sir_model.py
```

### 6.1.3 Execução via Notebooks Interativos

Os notebooks interativos permitem a execução incremental dos modelos, ajustes de parâmetros em tempo real e análise visual dos resultados. Na Figura 14 é ilustrado o histórico das variáveis simuladas no notebook.

Figura 14 – Histórico das variáveis Susceptíveis, Infectados e Recuperados em *notebook*.

```
In [3]: env = Environment()

In [4]: SIR(susceptible=9998, infected=2, recovered=0, duration=2, contacts=6,
            probability=0.25, final_time=30)

Out[4]: SIR (sir.0)

In [5]: Chart()

Out[5]: Chart (chart.0)

In [6]: env.run(30)
```




Fonte: Autoria própria.

### 6.1.4 Execução via Aplicação Web com *Streamlit*

Para facilitar o uso por não especialistas e promover demonstrações interativas, o *framework* DisSModel oferece uma interface web baseada na biblioteca *Streamlit*. No Código 22 é apresentada a configuração de um modelo SIR executável via navegador.

A execução deste *script* é realizada pelo comando:

```
streamlit run examples/sysdyn/streamlit/sir_model.py
```

A interface gerada por este código é ilustrada na Figura 15. Ela apresenta uma barra lateral onde o usuário pode selecionar o modelo desejado (por exemplo, *Coffee*, *SIR* ou *PredatorPrey*) e ajustar os parâmetros específicos, como temperatura inicial, taxas de crescimento ou probabilidade de infecção. Após a seleção do modelo e a definição dos valores, a simulação é executada, e os resultados são apresentados em gráficos temporais gerados automaticamente pelo módulo *Visualization*. Essa funcionalidade é uma

**Código 22** Execução do modelo SIR via Streamlit

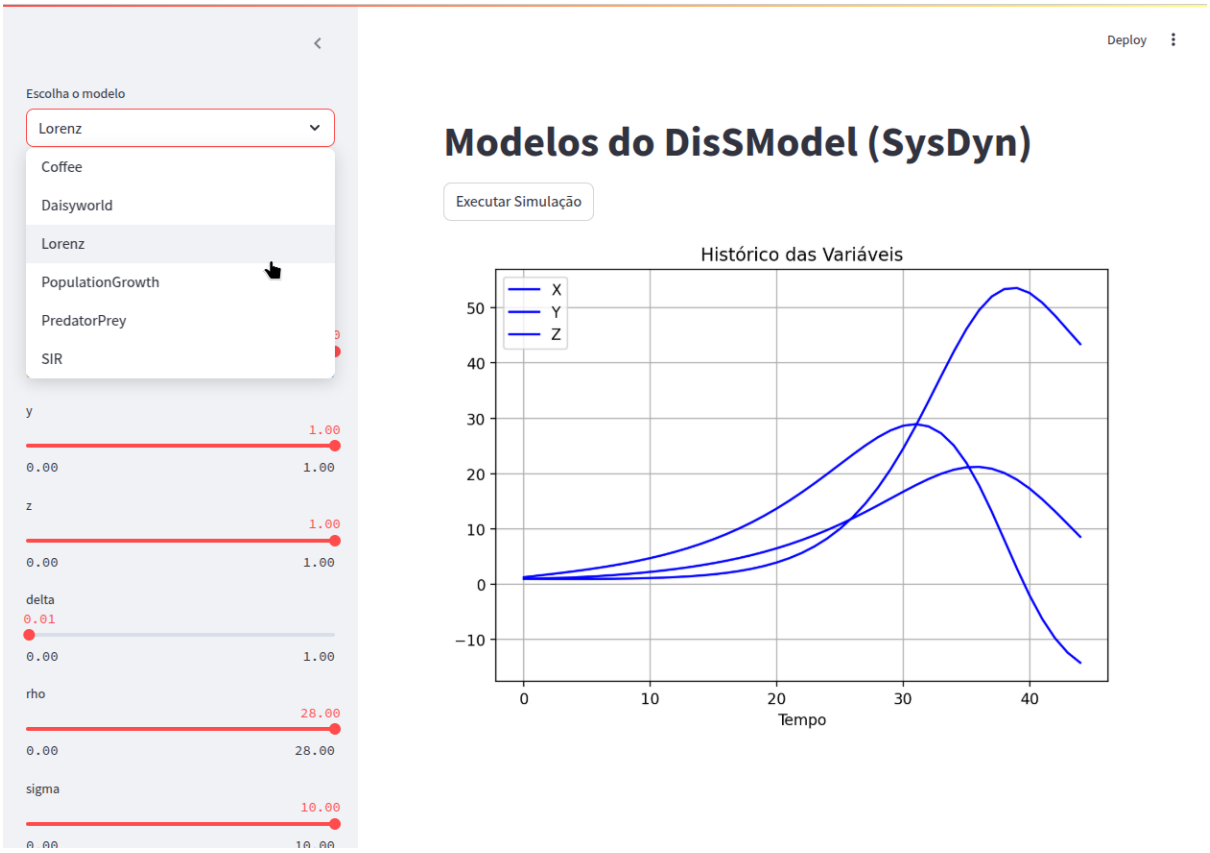
```python
import streamlit as st

from dissmodel.core import Environment
from dissmodel.visualization import Chart, display_inputs
from dissmodel.models.sysdyn import SIR

st.set_page_config(page_title="SIR Model", layout="centered")
st.title("SIR Model (DisSModel)")

st.sidebar.title("Parâmetros do Modelo")
steps = st.sidebar.slider("Número de passos da simulação", min_value=1,
↪  max_value=50, value=10)
executar = st.button("Executar Simulação")

env = Environment(end_time=steps, start_time=0)

sir = SIR(
    susceptible=9998, infected=2, recovered=0,
    duration=2, contacts=6, probability=0.25,
    final_time=30
)

display_inputs(sir, st.sidebar)
Chart(plot_area=st.empty())

if executar:
    env.reset()
    env.run()
```

vantagem significativa do DisSModel em relação ao TerraME, pois permite uma interação mais acessível e intuitiva, especialmente para usuários sem conhecimento avançado de programação. No TerraME, a configuração de parâmetros é realizada diretamente no código Lua ou por meio de interfaces menos flexíveis, o que pode limitar a experimentação rápida.

Além da execução individual de cada *script*, o *framework* também disponibiliza um arquivo `run_all.py` em cada subpasta `streamlit/` de cada paradigma. Este *script* realiza o carregamento automático de todos os modelos localizados na pasta `dissmodel/models`, reunindo-os em uma única interface interativa. Essa funcionalidade permite que o usuário explore múltiplos modelos de forma centralizada, facilitando demonstrações integradas.

Esta abordagem confirma a flexibilidade do *framework*, permitindo a escolha da forma de execução mais apropriada para experimentação, ensino ou extensão. Cada paradigma pode ser executado de três formas distintas — linha de comando (CLI),

Figura 15 – Interface do Streamlit para execução de modelos de sistemas dinâmicos no *framework* DisSModel.




Fonte: Autoria própria.

*notebook* interativo ou aplicação web com Streamlit — de acordo com o perfil do usuário e o contexto de uso.

## 6.2 Visão Geral dos Modelos

O módulo *Models* organiza os modelos em subpacotes *sysdyn* (sistemas dinâmicos) e *ca* (autômatos celulares), com suporte a modelos híbridos que integram ambos os paradigmas. Na Tabela 8 é apresentado uma visão geral dos modelos implementados, incluindo seus paradigmas, descrições e principais aplicações, proporcionando uma síntese das funcionalidades disponíveis no *framework DisSModel*.

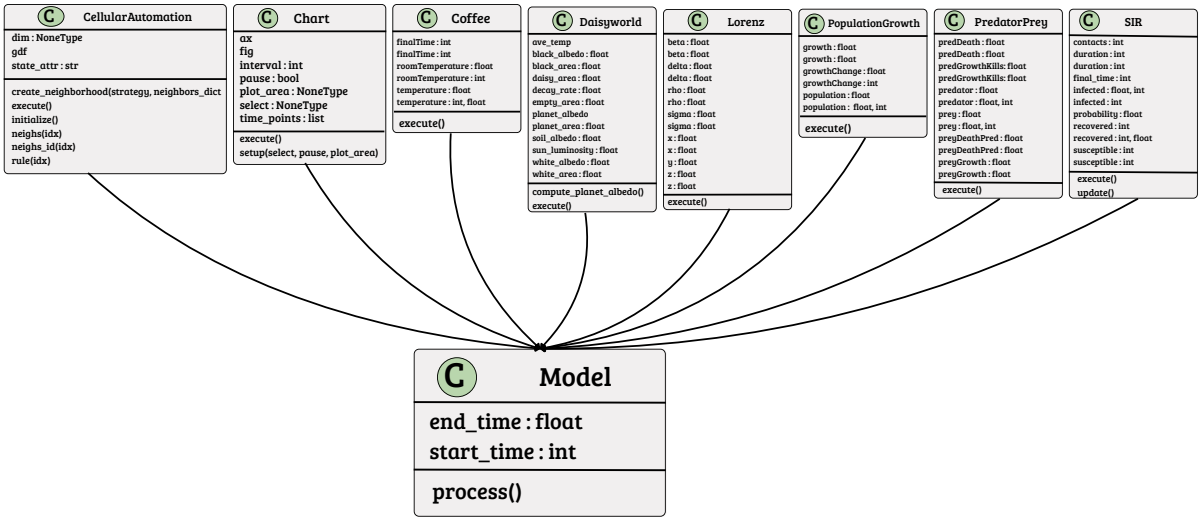
Tabela 8 – Modelos implementados no framework DisSModel

| **Modelo** | **Paradigma** | **Descrição** |
|---|---|---|
| SIR | SisDyn | Simula propagação de doenças infecciosas |
| PredatorPrey | SisDyn | Interação entre predadores e presas |
| Daisyworld | SisDyn | Regulação climática simplificada |
| Lorenz | SisDyn | Sistema caótico de convecção |
| PopulationGrowth | SisDyn | Crescimento populacional logístico |
| Tub | SisDyn | Fluxos de entrada e saída de recursos |
| Coffee | SisDyn | Convergência de temperatura |
| Game of Life | CA | Evolução de células com regras locais |
| FireModelProb | CA | Propagação probabilística de incêndios |
| Anneal | CA | Sistema binário com regras locais |
| Snow | CA | Acumulação e deslocamento de neve |
| Propagation | CA | Propagação de estados ativos |
| Growth | CA | Crescimento a partir de célula central |

## 6.3 Sistemas Dinâmicos

Os sistemas dinâmicos modelam a evolução temporal de variáveis contínuas por meio de equações diferenciais discretizadas, implementadas no subpacote *sysdyn*. No *framework DisSModel*, esses modelos derivam da classe *Model* do módulo *Core*, que gerencia a inicialização, execução iterativa e integração com visualização, conforme detalhado na Seção 5.1. A estrutura orientada a objetos do *framework* é ilustrada na Figura 16, que apresenta o diagrama de classes referente aos modelos de sistemas dinâmicos, destacando as heranças e associações específicas do subpacote *sysdyn*, que oferece suporte à definição e execução de modelos dinâmicos contínuos e discretos.

Figura 16 – Diagrama de classes dos modelos de sistemas dinâmicos no *framework DisSModel*.




Fonte: Autoria própria.

A seguir, são apresentados três exemplos representativos de sistemas dinâmicos: os modelos *Coffee*, *SIR* e *PredatorPrey*. Cada modelo é descrito em termos de sua formulação matemática, implementação em código e resultados de simulação.

### 6.3.1 Modelo *Coffee*

O modelo *Coffee* simula a convergência da temperatura de um líquido à temperatura ambiente, utilizando uma equação de resfriamento discretizada. A cada iteração, a diferença entre a temperatura atual (*temperature*) e a temperatura ambiente (*roomTemperature*) é reduzida proporcionalmente, resultando em estabilização após um número finito de passos.

A implementação do modelo no *framework* TerraME, escrita em Lua, é apresentada no Código 23.

**Código 23** Implementação do modelo *Coffee* no *framework* TerraME.

```
Coffee = Model{
    temperature = 80,
    roomTemperature = 20,
    finalTime = 20,

    execute = function(model)
        local difference = model.temperature - model.roomTemperature
        model.temperature = model.temperature - difference * 0.1
    end,

    init = function(model)
        model.chart = Chart{target = model, select = {"temperature"}}
        model.timer = Timer{
            Event{action = model},
            Event{action = model.chart}
        }
    end
}
```

No TerraME, a implementação define a temperatura inicial do líquido (80°C por padrão), a temperatura ambiente (20°C) e o tempo final de simulação (20 iterações) diretamente no modelo. A função *execute* atualiza a temperatura a cada iteração, reduzindo a diferença em 10% conforme a equação de resfriamento. A visualização é configurada explicitamente no método *init*, utilizando o componente *Chart* para gerar gráficos, com eventos de visualização definidos manualmente no *Timer*.

Em comparação, a implementação no *framework* DisSModel, escrita em Python, é apresentada no Código 24. Esta versão utiliza a classe *Model* e a função decoradora

*@track_plot* para automatizar a geração de visualizações.

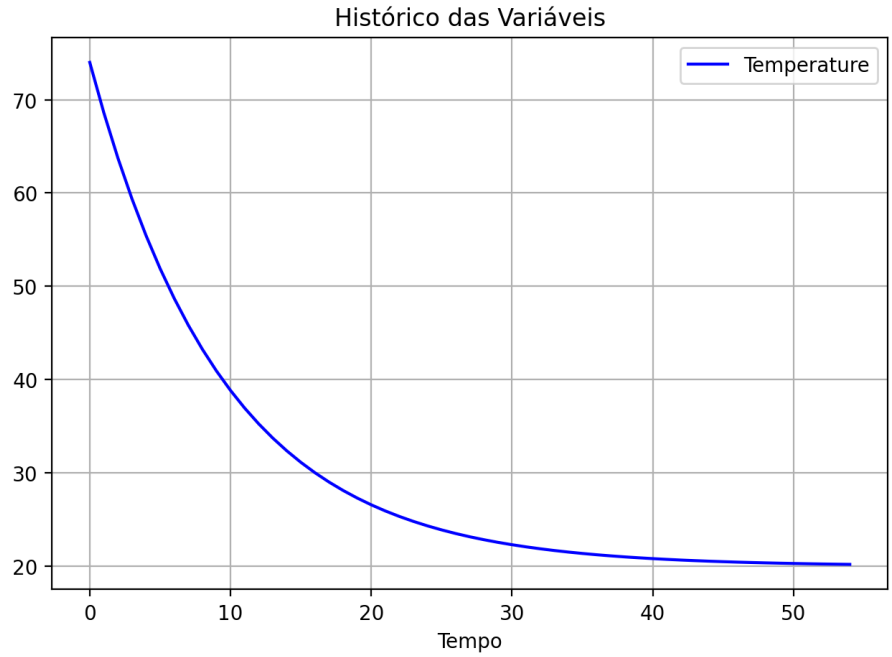
**Código 24** Implementação do modelo *Coffee* no *framework DisSModel*.

```python
from dissmodel.core import Model
from dissmodel.visualization import track_plot

@track_plot("Temperature", "blue")
class Coffee(Model):
    temperature: float
    roomTemperature: float

    def setup(self, temperature=80, roomTemperature=20, finalTime=20):
        self.temperature = temperature
        self.roomTemperature = roomTemperature

    def execute(self):
        difference = self.temperature - self.roomTemperature
        self.temperature -= difference * 0.1
```

Na implementação do DisSModel, o método *setup* inicializa a temperatura do líquido (80°C por padrão), a temperatura ambiente (20°C) e o tempo final de simulação (20 iterações). O método *execute* atualiza a temperatura a cada iteração, seguindo a mesma equação de resfriamento. A função *@track_plot* rastreia automaticamente a variável *temperature*, gerando um gráfico da evolução temporal de forma simplificada. Comparado ao TerraME, o DisSModel beneficia-se da integração com bibliotecas Python, como *Matplotlib*, oferecendo maior flexibilidade na personalização dos gráficos e eliminando a necessidade de configurar eventos de visualização manualmente.

O resultado da simulação no DisSModel é ilustrado na Figura 17, que mostra a convergência da temperatura do líquido à temperatura ambiente (20°C) após 54 iterações, demonstrando o comportamento esperado do modelo. A execução via Streamlit, conforme descrito na Seção 6.1, permite ajustar parâmetros, como a temperatura inicial ou a taxa de resfriamento, e visualizar os resultados em tempo real, tornando o processo mais interativo.

Figura 17 – Resultado da simulação do modelo *Coffee* após 54 iterações.




Fonte: Autoria própria.

A abordagem do DisSModel oferece uma implementação mais modular e acessível, especialmente para usuários que preferem Python e desejam uma integração simplificada com visualizações automáticas, em contraste com a configuração mais explícita exigida pelo TerraME.

### 6.3.2 Modelo *PredatorPrey*

O modelo *PredatorPrey* representa as interações ecológicas entre populações de presas e predadores, considerando taxas de crescimento, mortalidade e consumo. As equações discretizadas que governam o modelo são:

$$\text{prey}_{t+1} = \text{prey}_t + \text{preyGrowth} \cdot \text{prey}_t - \text{preyDeathPred} \cdot \text{prey}_t \cdot \text{predator}_t, \tag{6.1}$$

$$\text{predator}_{t+1} = \text{predator}_t - \text{predDeath} \cdot \text{predator}_t + \text{predGrowthKills} \cdot \text{prey}_t \cdot \text{predator}_t. \tag{6.2}$$

A implementação do modelo no *framework* TerraME, escrita em Lua, é apresentada no Código 25.

**Código 25** Implementação do modelo *PredatorPrey* no *framework* TerraME.

```
PredatorPrey = Model{
    predator = Choice{min = 1, default = 40},
    prey = Choice{min = 1, default = 1000},
    preyGrowth = Choice{min = 0.000001, max = 1, default = 0.08},
    preyDeathPred = Choice{min = 0.000001, max = 0.5, default = 0.001},
    predDeath = Choice{min = 0.000001, max = 0.5, default = 0.02},
    predGrowthKills = Choice{min = 0, max = 0.5, default = 0.00002},
    finalTime = Choice{min = 5, default = 500},
    period = Choice{min = 0.0001, default = 1},

    execute = function(model)
        model.prey = model.prey + model.preyGrowth * model.prey
            - model.preyDeathPred * model.prey * model.predator
        model.predator = model.predator - model.predDeath * model.predator
            + model.predGrowthKills * model.prey * model.predator
    end,

    init = function(model)
        model.chart1 = Chart{target = model, select = {"prey", "predator"}}
        model.chart2 = Chart{target = model, select = "predator", xAxis =
        ↪  "prey"}
        if model.period == 1 then model.period = nil end
        model.timer = Timer{
            Event{action = model, period = model.period},
            Event{action = model.chart1},
            Event{action = model.chart2}
        }
    end
}
```

No TerraME, a configuração de parâmetros utiliza o componente *Choice* para definir valores padrão (e.g., 1000 presas, 40 predadores) e limites para as taxas de crescimento, mortalidade e interação. A função *execute* atualiza as populações de presas e predadores em cada iteração, aplicando as equações discretizadas (6.1 e 6.2). A visualização requer a criação explícita de dois gráficos no método *init*: *chart1* para a evolução temporal das populações e *chart2* para um gráfico de fase (*predator* versus *prey*). Eventos de visualização são configurados manualmente no *Timer*, o que exige maior intervenção do usuário em comparação com abordagens mais automatizadas.

Em contraste, a implementação no *framework* DisSModel, escrita em Python, é apresentada no Código 26. Esta versão utiliza a classe *Model* e a função decoradora *@track_plot* para simplificar a visualização.

**Código 26** Implementação do modelo *PredatorPrey* no *framework DisSModel*.

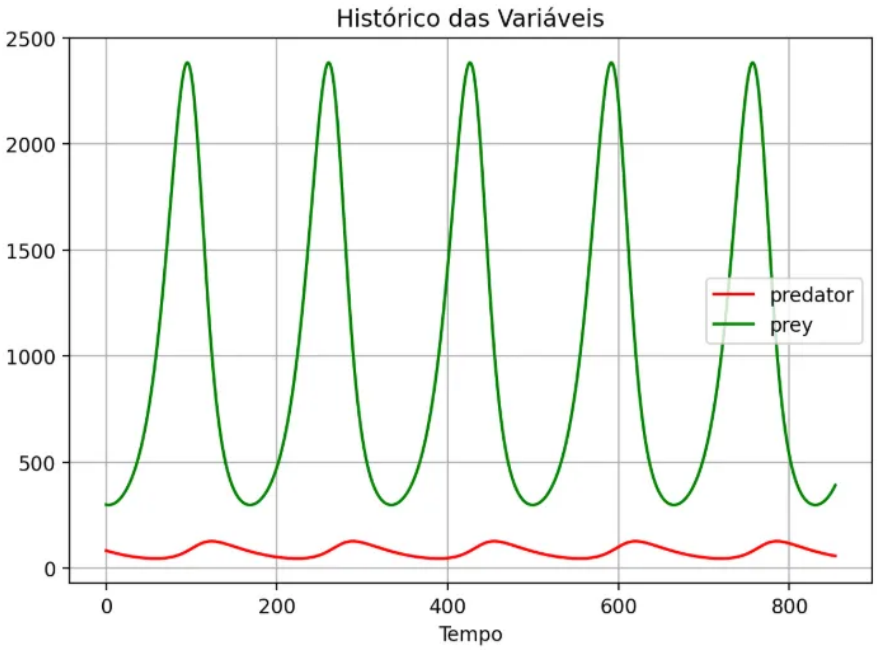
```python
from dissmodel.core import Model
from dissmodel.visualization import track_plot

@track_plot("prey", "green")
@track_plot("predator", "red")
class PredatorPrey(Model):
    predator: float
    prey: float
    preyGrowth: float
    preyDeathPred: float
    predDeath: float
    predGrowthKills: float

    def setup(self, predator=40, prey=1000,
              preyGrowth=0.08, preyDeathPred=0.001,
              predDeath=0.02, predGrowthKills=0.00002):
        self.predator = predator
        self.prey = prey
        self.preyGrowth = preyGrowth
        self.preyDeathPred = preyDeathPred
        self.predDeath = predDeath
        self.predGrowthKills = predGrowthKills

    def execute(self):
        self.prey += (
            self.preyGrowth * self.prey
            - self.preyDeathPred * self.prey * self.predator
        )
        self.predator += (
            -self.predDeath * self.predator
            + self.predGrowthKills * self.prey * self.predator
        )
```

Na implementação do DisSModel, o método *setup* inicializa o modelo com 1000 presas, 40 predadores e taxas específicas de crescimento (*preyGrowth*=0.08), mortalidade (*predDeath*=0.02), e interação (*preyDeathPred*=0.001, *predGrowthKills*=0.00002). O método *execute* atualiza as populações em cada iteração, aplicando as mesmas equações discretizadas. A função *@track_plot* rastreia automaticamente as variáveis *prey* e *predator*, gerando um gráfico temporal em cores distintas (verde para presas, vermelho para predadores). Comparado ao TerraME, o DisSModel simplifica a visualização ao eliminar a necessidade de configurar gráficos manualmente, além de se beneficiar da integração com bibliotecas Python, como *Matplotlib*, para maior flexibilidade na personalização. A separação entre a lógica do modelo e a execução também torna o código mais modular, facilitando sua

reutilização em contextos como CLI ou interface gráfica via Streamlit, conforme descrito na Seção 6.1.

O resultado da simulação no DisSModel é ilustrado na Figura 18, que exibe a evolução temporal das populações de presas e predadores com os parâmetros padrão, permitindo a análise das interações ecológicas.

Figura 18 – Resultado da simulação do modelo *PredatorPrey*.




Fonte: Autoria própria.

Na Figura 18 é mostrado as oscilações características das populações de presas e predadores, refletindo o comportamento dinâmico esperado do modelo. A integração com Streamlit, como destacado na Seção 6.1, permite ajustar dinamicamente os parâmetros e visualizar os resultados em tempo real, oferecendo uma interação mais acessível e intuitiva em comparação com a abordagem do TerraME, que depende de configurações mais explícitas no código Lua.

### 6.3.3 Modelo SIR

O modelo *SIR* simula a dinâmica epidemiológica de uma doença infecciosa, dividindo a população em três compartimentos: suscetíveis ($S$), infectados ($I$) e recuperados ($R$). As equações discretizadas que governam o modelo são:

$$S_{t+1} = S_t - \alpha \frac{S_t I_t}{N}, \tag{6.3}$$

$$I_{t+1} = I_t + \alpha \frac{S_t I_t}{N} - \beta I_t, \tag{6.4}$$

$$R_{t+1} = R_t + \beta I_t, \tag{6.5}$$

onde $N = S_t + I_t + R_t$ é a população total, $\alpha = \mathit{contacts} \cdot \mathit{probability}$ representa a taxa de contágio, e $\beta = 1/\mathit{duration}$ é a taxa de recuperação. A implementação é apresentada no Código 27, que utiliza a classe *Model* e múltiplas funções *@track_plot* para visualizar a evolução dos compartimentos.

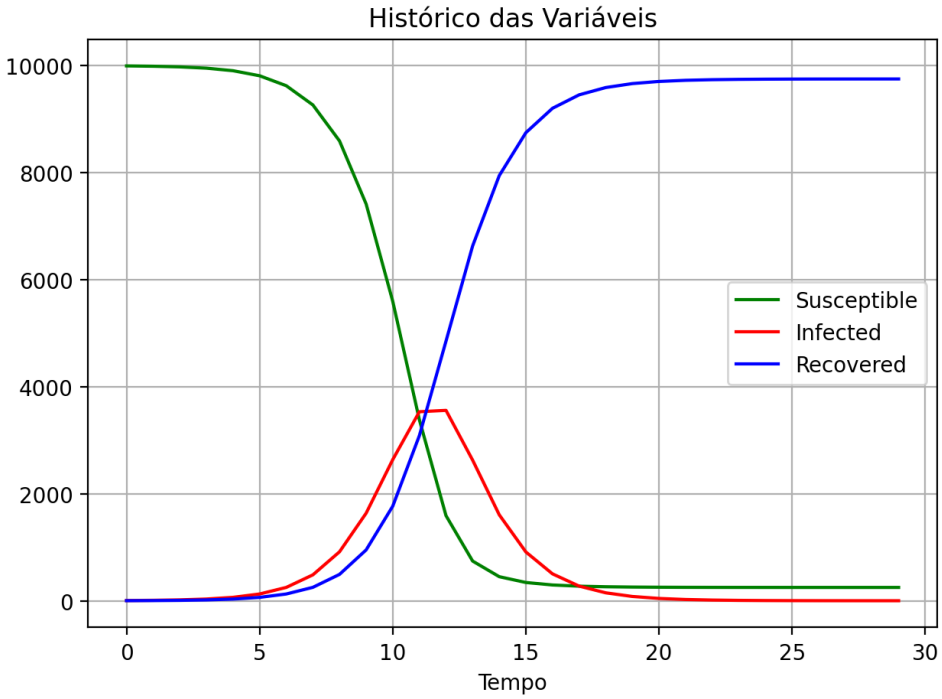
**Código 27** Implementação do modelo *SIR* no *framework DisSModel*.

```python
from dissmodel.core import Model
from dissmodel.visualization import track_plot

@track_plot("Susceptible", "green")
@track_plot("Infected", "red")
@track_plot("Recovered", "blue")
class SIR(Model):
    susceptible: int
    infected: int
    recovered: int
    duration: int
    probability: float

    def setup(self, susceptible=9998, infected=2, recovered=0, duration=2,
              contacts=6, probability=0.25):
        self.susceptible = susceptible
        self.infected = infected
        self.recovered = recovered
        self.duration = duration
        self.contacts = contacts
        self.probability = probability

    def execute(self):
        total = self.susceptible + self.infected + self.recovered
        alpha = self.contacts * self.probability
        prop = self.susceptible / total
        new_infected = self.infected * alpha * prop
        new_recovered = self.infected / self.duration
        self.susceptible -= new_infected
        self.infected += new_infected - new_recovered
        self.recovered += new_recovered
```

O método *setup* define os parâmetros iniciais, incluindo uma população de 9998

suscetíveis, 2 infectados, taxa de contágio baseada em 6 contatos diários com probabilidade de infecção de 0,25, e uma duração média de infecção de 2 dias. O método *execute* atualiza os compartimentos em cada iteração, aplicando as equações discretizadas. A visualização gerada por *@track_plot* exibe a evolução temporal de $S$, $I$ e $R$, permitindo a análise da propagação da doença. Na Figura 19 é possível visualizar os resultados após 29 iterações com os parâmetros padrão.

Figura 19 – Resultado da simulação do modelo *SIR* após 29 iterações.



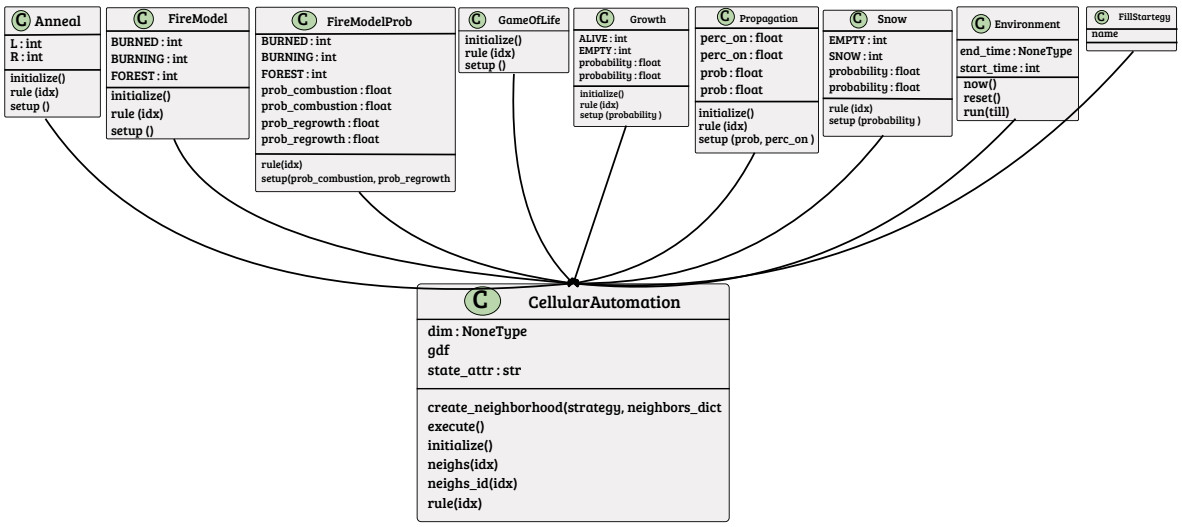

Fonte: Autoria própria.

Na Figura 19 é ilustrado a evolução dos compartimentos ao longo de 29 iterações, mostrando o pico de infectados e a subsequente recuperação da população. A interface Streamlit, conforme apresentado na Figura 15, permite ajustar parâmetros como a taxa de contágio (*contacts* e *probability*) ou a duração da infecção, facilitando a análise de diferentes cenários epidemiológicos.

## 6.4 Autômatos Celulares

Os autômatos celulares (AC) modelam dinâmicas espaciais por meio de uma grade de células que evoluem com base em regras locais, implementadas no subpacote *ca* do módulo *Geo*, conforme descrito na Seção 5.3. Esse paradigma é particularmente adequado para simulações espaciais e estudos de padrões emergentes, como propagação de incêndios ou desmatamento.

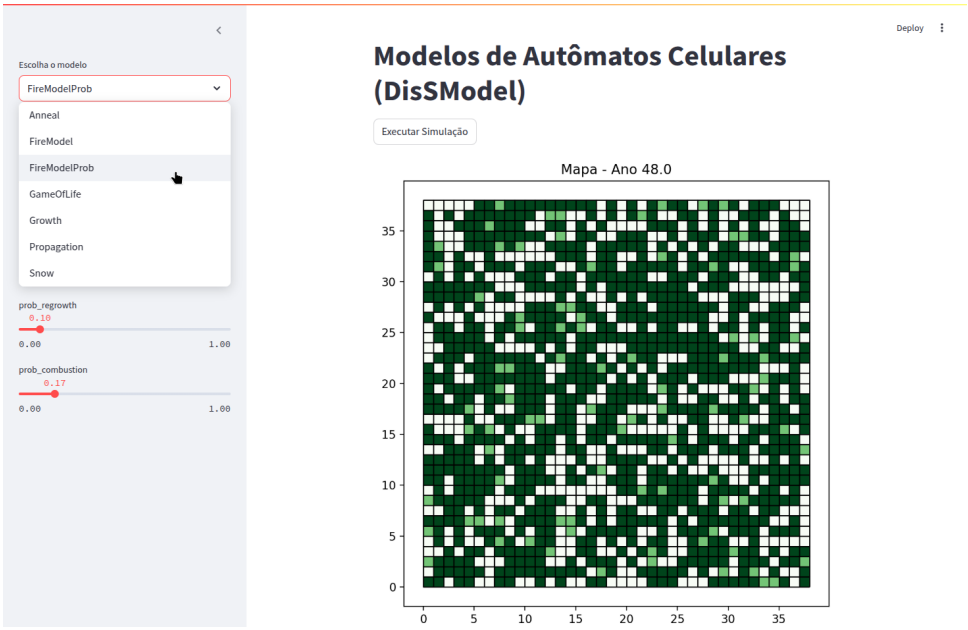
Figura 20 – Diagrama de classes do *framework DisSModel*, com ênfase nos autômatos celulares.




Fonte: Autoria própria.

Para os modelos de autômatos celulares, a interface Streamlit também foi utilizada, como ilustrado na Figura 21. A barra lateral permite selecionar modelos como *Game of Life*, *FireModelProb* ou *Snow*, configurando parâmetros como dimensões da grade, probabilidade de estados iniciais ou regras de transição. Os resultados são exibidos como mapas espaciais, mostrando a evolução das células ao longo do tempo.

Figura 21 – Interface do Streamlit para execução de modelos de autômatos celulares no *framework* DisSModel.




Fonte: Autoria própria.

Na Figura 21 é destacado a visualização de mapas para o modelo *Snow*, permitindo ao usuário interagir com os parâmetros e observar os padrões espaciais emergentes. Essa abordagem é mais flexível que o TerraME, onde a visualização de autômatos celulares exige a configuração explícita de componentes como *Map* e *Timer*, como visto no Código 23 para o *Game of Life*.

A seguir, são apresentados três exemplos de autômatos celulares: *Game of Life*, *FireModelProb* e *Snow*. Cada modelo é descrito em termos de sua formulação, implementação e resultados de simulação, acompanhados por códigos e figuras.

### 6.4.1 Game of Life

O modelo *Game of Life* é um autômato celular clássico onde células vivas (*state=1*) sobrevivem com 2 ou 3 vizinhos vivos e morrem caso contrário, enquanto células mortas (*state=0*) nascem com exatamente 3 vizinhos vivos. A implementação no *framework* DisSModel, escrita em Python, é apresentada no Código 28, utilizando uma vizinhança *Queen* e inicialização aleatória.

**Código 28** Implementação do modelo *Game of Life* no *framework DisSModel*.

```python
from dissmodel.geo import CellularAutomaton, FillStrategy, Queen

class GameOfLife(CellularAutomaton):
    def initialize(self):
        fill(strategy=FillStrategy.RANDOM_SAMPLE, gdf=self.gdf, attr="state",
        ↪  data={1: 0.5, 0: 0.5}, seed=42)
    def setup(self):
        self.create_neighborhood(strategy=Queen, use_index=True)
    def rule(self, idx):
        value = self.gdf.loc[idx, self.state_attr]
        neighs = self.neighs(idx)
        count = neighs[self.state_attr].fillna(0).sum()
        if value == 1: return 1 if 2 <= count <= 3 else 0
        else: return 1 if count == 3 else 0
```

No *framework* DisSModel, o modelo é definido pela classe *GameOfLife*, que herda de *CellularAutomaton*. O método *initialize* configura os estados iniciais das células com 50% de probabilidade de serem vivas (*state=1*), utilizando a estratégia *RANDOM_SAMPLE* com uma semente fixa (*seed=42*) para reprodutibilidade. O método *setup* define a vizinhança *Queen*, e o método *rule* implementa as regras de transição, contando os vizinhos vivos e aplicando as condições do *Game of Life*. Além da inicialização aleatória, o modelo suporta padrões predefinidos, como *glider* e *blinker*, para estudo de dinâmicas específicas. A implementação no *framework* TerraME, escrita em Lua, é apresentada no Código 29.

**Código 29** Implementação do modelo *Game of Life* no *framework* TerraME.

```
Life = Model{
  finalTime = 100,
  dim = 30,
  pattern = Choice(patterns),
  random = true,
  init = function(model)
    model.cell = Cell{
      init = function(cell)
        cell.state = "dead"
      end,
      execute = function(cell)
        local alive = countNeighbors(cell, "alive")
        if alive < 2 then
          cell.state = "dead"
        elseif alive > 3 then
          cell.state = "dead"
        elseif alive == 3 and cell.past.state == "dead" then
          cell.state = "alive"
        end
      end
    }
    model.cell.init = function(cell)
      if Random():number() > 0.5 then
        cell.state = "alive"
      else
        cell.state = "dead"
      end
    end
    model.cs = CellularSpace{
      xdim = model.dim,
      instance = model.cell
    }
    model.cs:createNeighborhood{wrap = true}
    model.map = Map{
      target = model.cs,
      select = "state",
      value = {"dead", "alive"},
      color = {"black", "white"}
    }
    model.timer = Timer{
      Event{action = model.cs},
      Event{action = model.map}
    }
  end
}
```

No TerraME, a implementação define uma grade de 30x30 células (*dim=30*)

com inicialização aleatória, onde cada célula tem 50% de probabilidade de estar viva (*state="alive"*). A função *execute* implementa as regras de transição, contando os vizinhos vivos com *countNeighbors* e atualizando o estado da célula com base nas condições de sobrevivência ou nascimento. A visualização é configurada no método *init* utilizando o componente *Map*, que associa os estados (*dead*, *alive*) às cores (*black*, *white*). A criação da grade (*CellularSpace*) e da vizinhança (*createNeighborhood*) é feita diretamente no código, o que acopla a lógica do modelo à execução e à visualização, reduzindo a modularidade.

A execução do modelo no DisSModel via CLI é apresentada no Código 30.

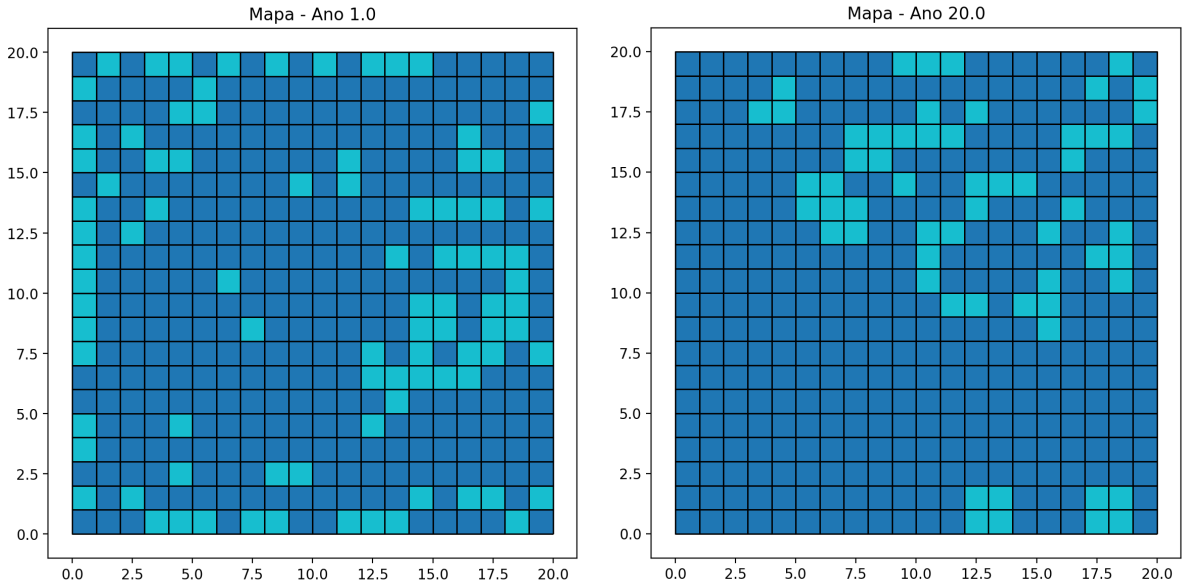
**Código 30** Execução do modelo *Game of Life* no *framework DisSModel*.

```
from dissmodel.models.ca import GameOfLife
from dissmodel.utils import regular_grid, Map
from matplotlib.colors import ListedColormap
from dissmodel.environment import Environment

gdf = regular_grid((20, 20), 1, attrs={'state': 0})
env = Environment(start_time=0, end_time=10)
life = GameOfLife(gdf=gdf)
life.initialize()
Map(
    gdf=gdf,
    plot_params={"column": "state",
        "cmap": ListedColormap(['green', 'red']),
        "ec": "black"}
)
env.run()
```

No Código 30 é ilustrado a execução via CLI, onde uma grade 20x20 é criada com *regular_grid*, e a visualização é configurada com o componente *Map*, mapeando estados para cores (verde para células mortas, vermelho para células vivas). A separação entre a lógica do modelo (*GameOfLife*) e a execução (*Environment*, *Map*) aumenta a modularidade, permitindo a reutilização do modelo em diferentes contextos, como CLI ou interface Streamlit, conforme descrito na Seção 6.1.

Os resultados da simulação no DisSModel são apresentados na Figura 22, que mostra os mapas de duas iterações, ilustrando a evolução espacial das células nas iterações 1 e 20.

Figura 22 – Mapas do modelo *Game of Life* para as duas primeiras iterações.




(a) Mapa de iteração ano 1.0 do modelo. (b) Mapa de iteração ano 20.0 do modelo.

Fonte: Autoria própria.

Na Figura 22 é destacado a evolução espacial das células, evidenciando a dinâmica característica do *Game of Life*. A execução via Streamlit, conforme ilustrado na Figura 21, permite configurar a grade e padrões iniciais, como *glider* ou *blinker*, e visualizar os resultados dinamicamente. Comparado ao TerraME, o DisSModel oferece maior flexibilidade e modularidade, facilitando a integração com bibliotecas Python, como *Matplotlib*, e a reutilização do modelo em diferentes cenários, enquanto a abordagem do TerraME acopla a lógica do modelo à execução e à visualização.

### 6.4.2 FireModelProb

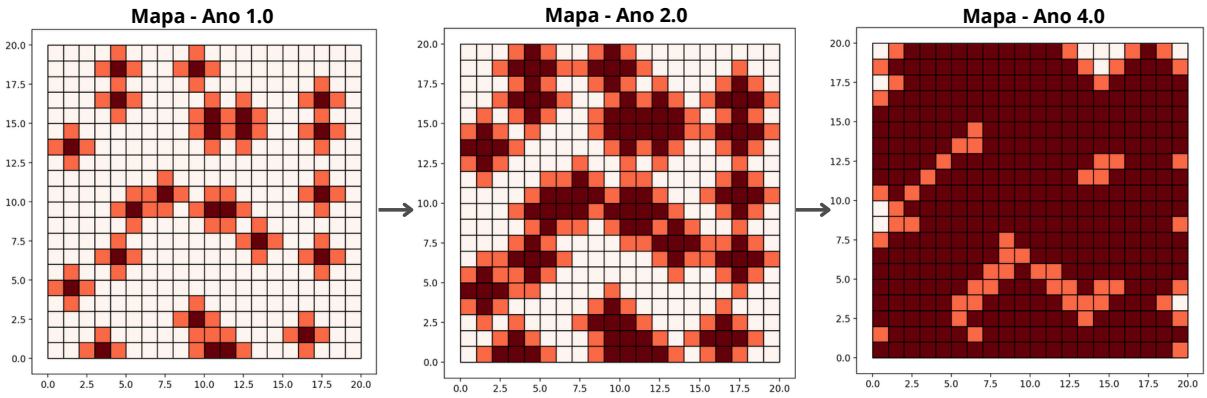
O modelo *FireModelProb* simula a propagação de incêndios florestais com três estados: floresta intacta ($FOREST$=0), em chamas ($BURNING$=1) e queimada ($BURNED$=2). As transições dependem da presença de vizinhos em chamas e de probabilidades de combustão e regeneração. A implementação é apresentada no Código 31, que utiliza uma vizinhança *Rook*.

**Código 31** Implementação do modelo *FireModelProb* no *framework DisSModel*.

```
from dissmodel.geo import CellularAutomaton, FillStrategy, Rook

class FireModel(CellularAutomaton):
    FOREST = 0
    BURNING = 1
    BURNED = 2
    def initialize(self):
        fill(strategy=FillStrategy.RANDOM_SAMPLE, gdf=self.gdf, attr="state",
        ↪  data={self.FOREST: 0.95, self.BURNING: 0.05}, seed=42)
    def setup(self):
        self.create_neighborhood(strategy=Rook, use_index=True)
        self.initialize()
    def rule(self, idx):
        state = self.gdf.loc[idx, self.state_attr]
        if state == self.BURNING:
            return self.BURNED
        elif state == self.FOREST:
            neighs = self.neighs(idx)
            if (neighs[self.state_attr] == self.BURNING).any():
                return self.BURNING
            else:
                return state
        else:
            return state
```

Na Figura 23 é ilustrado os mapas da simulação em diferentes anos, destacando a propagação do incêndio.

Figura 23 – Mapas do modelo *FireModelProb* para diferentes anos.




Fonte: Autoria própria.

O método *initialize* configura 95% das células como floresta intacta e 5% como em

chamas, utilizando a estratégia *RANDOM_SAMPLE*. O método *setup* define a vizinhança *Rook*, e o método *rule* implementa as regras de transição, onde uma célula em chamas torna-se queimada, e uma célula de floresta pode entrar em chamas se pelo menos um vizinho estiver no estado *BURNING*.

A execução via Streamlit, conforme mostrado na Figura 21, permite ajustar a proporção inicial de células em chamas ou a probabilidade de combustão, visualizando os resultados dinamicamente. Em comparação, no Código 32 é apresentado a implementação no TerraME.

**Código 32** Implementação do modelo *FireModelProb* no *framework* TerraME.

```
Fire = Model{
  finalTime = 100,
  empty = Choice{min = 0, max = 1, default = 0.1},
  dim = 60,
  random = true,
  init = function(model)
    model.cell = Cell{
      init = function(cell)
        if Random():number() > model.empty then
          cell.state = "forest"
        else
          cell.state = "empty"
        end
      end,
      execute = function(cell)
        if cell.past.state == "burning" then
          cell.state = "burned"
        elseif cell.past.state == "forest" then
          local burning = countNeighbors(cell, "burning")
          if burning > 0 then
            cell.state = "burning"
          end
        end
      end
    }

    ...

  end
}
```

### 6.4.3 Snow

O modelo *Snow* representa o acúmulo e deslocamento vertical de neve em uma grade regular, com estados $EMPTY$=0 e $SNOW$=1. Células no topo da grade geram

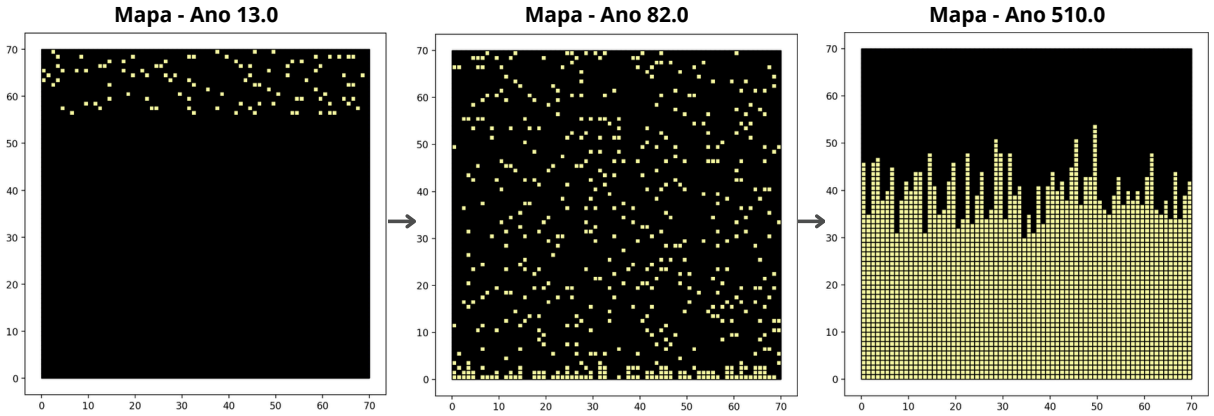
neve com uma probabilidade definida, e a neve desce se a célula inferior estiver vazia. A implementação é apresentada no Código 33.

**Código 33** Implementação do modelo *Snow* no *framework DisSModel*.

```
from dissmodel.geo import CellularAutomaton
import random

class Snow(CellularAutomaton):
    EMPTY = 0
    SNOW = 1

    probability: float

    def setup(self, probability=0.02):
        self.probability = probability
    def rule(self, idx):
        cell = self.gdf.loc[idx]
        x, y = parse_idx(idx)
        t = self.env.now()
        if y == self.dim - 1:
            if cell.state == self.EMPTY and t < (self.end_time - self.dim) \
                and random.random() < self.probability:
                return self.SNOW
            return self.EMPTY
        below_idx = f"{y - 1}-{x}" if y - 1 >= 0 else None
        if cell.state == self.SNOW and y == 0:
            return self.SNOW
        elif cell.state == self.SNOW and below_idx:
            below_state = self.gdf.loc[below_idx, "state"]
            if below_state == self.EMPTY:
                return self.EMPTY
            else:
                return self.SNOW
        above_idx = f"{y + 1}-{x}" if y + 1 < self.dim else None
        if above_idx and self.gdf.loc[above_idx, "state"] == self.SNOW:
            return self.SNOW
        return self.EMPTY
```

O método *setup* define a probabilidade de geração de neve (2% por padrão). O método *rule* implementa as regras de transição, onde células no topo geram neve aleatoriamente, e a neve desce se a célula inferior estiver vazia. Na Figura 24 é ilustrado os mapas da simulação em diferentes anos, destacando o acúmulo e deslocamento da neve.

Figura 24 – Mapas do modelo *Snow* para diferentes anos.




Fonte: Autoria própria.

A execução via Streamlit, conforme mostrado na Figura 21, permite ajustar a probabilidade de geração de neve e visualizar os padrões de acúmulo, destacando a flexibilidade do DisSModel em relação ao TerraME.

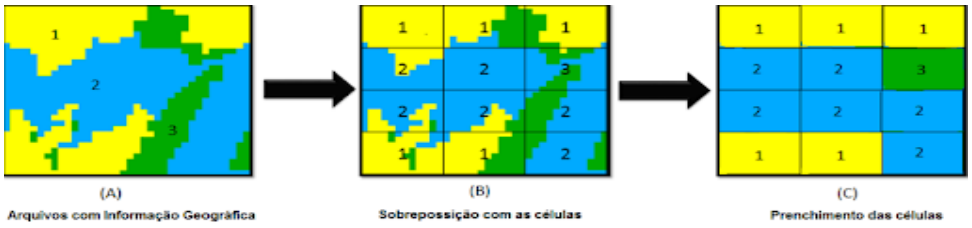
## 6.5 Integração com Dados Geográficos

A integração de dados geográficos é uma etapa fundamental na construção de modelos espaciais dinâmicos. Antes de implementar qualquer lógica de simulação, é necessário harmonizar diferentes fontes de informação espacial em uma estrutura comum: a grade regular. No *framework DisSModel*, essa integração é viabilizada pela flexibilidade do ecossistema *Python* e por bibliotecas como *GeoPandas*, *Rasterio* e *matplotlib*, associadas à abstração de grades no módulo *Geo*.

O processo envolve duas etapas principais: (1) criação da grade regular e (2) integração dos diferentes tipos de dados geográficos a essa estrutura. Essa abordagem garante que todas as variáveis relevantes sejam harmonizadas em um mesmo referencial espacial antes da fase de modelagem propriamente dita. Frequentemente, os dados de entrada possuem resoluções distintas — por exemplo, imagens de satélite com 30m x 30m de resolução — enquanto as simulações podem operar em grades de 10km x 10km, exigindo processos de reamostragem, agregação ou interpolação.

Os modelos desenvolvidos utilizam o espaço celular para homogeneizar variáveis ambientais, socioeconômicas e de infraestrutura, independentemente do formato original dos dados (vetorial, matricial etc.), conforme ilustrado na Figura 25. Assim, operadores espaciais são aplicados para agregar essas variáveis em cada célula, como distância mínima a estradas, porcentagem de cobertura florestal ou densidade populacional.

Figura 25 – Esquema de homogeneização espacial utilizando grade celular. (A) Arquivos com informações geográficas originais; (B) sobreposição dessas informações com a grade de células regulares; (C) preenchimento das células com a classe predominante em cada uma. Adaptado de BEZERRA et al. (BEZERRA et al., 2022).




Fonte: Autoria Própria.

Embora tais cálculos possam ser realizados em Sistemas de Informação Geográfica (SIG), como QGIS ou TerraView, isso pode comprometer a reprodutibilidade e a rastreabilidade dos dados processados. Por esse motivo, o *DisSModel* inclui rotinas que automatizam essa etapa de integração, criando uma base sólida para a construção de modelos. Um exemplo é o modelo desenvolvido para simular os impactos da elevação do nível do mar no ecossistema de manguezal da Ilha do Maranhão, que utilizou modelagem espacialmente explícita baseada em autômatos celulares para avaliar cenários de deslocamento potencial (BEZERRA et al., 2022).

### 6.5.1 Criação da Grade Regular

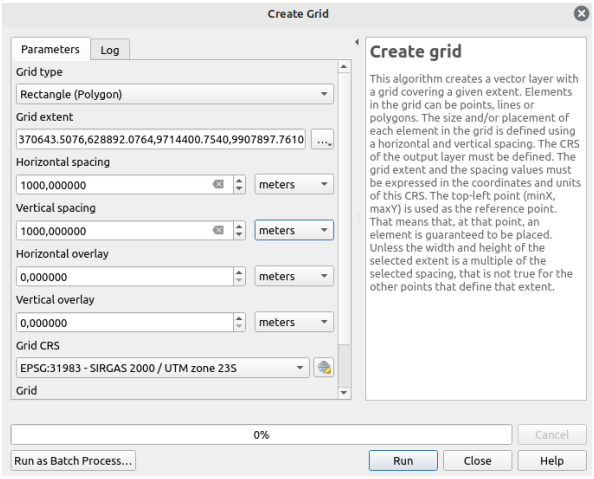
A criação de uma grade regular consiste em definir uma malha poligonal com base nas dimensões espaciais de um raster de referência, como uma imagem classificada do *MapBiomas*. No Código 34 é apresentado a implementação, que extrai os limites e o sistema de coordenadas do raster para gerar uma grade com resolução de 1000 metros.

**Código 34** Criação de uma grade regular com base em um raster de referência.

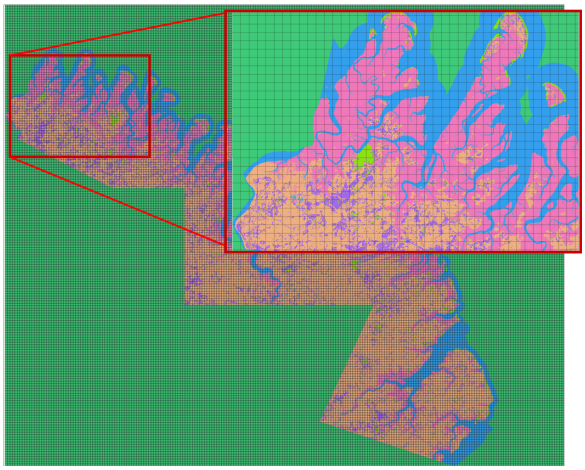
```
import rasterio
from dissmodel.geo import regular_grid

with rasterio.open(mapbiomas_tif) as src:
    bounds = src.bounds
    resolution = max(src.res)
    crs = src.crs.to_string()
grid = regular_grid(
    bounds=(bounds.left, bounds.bottom, bounds.right, bounds.top),
    resolution=1000,
    crs=crs
)
```

O código utiliza a biblioteca *rasterio* para abrir o raster e extrair seus limites, resolução e sistema de coordenadas. A função *regular_grid* gera uma grade poligonal compatível com o sistema de coordenadas, permitindo a associação com atributos geográficos. Essa operação pode ser realizada em sistemas de informação geográfica, como *QGIS* ou *TerraView*, mas sua implementação no *framework* garante a reprodutibilidade direta via *script*. Na Figura 26 é ilustrado a criação da grade no *QGIS* (subfigura 26a) e o resultado da grade sobreposta ao raster de uso do solo (subfigura 26b), destacando a possibilidade de múltiplos usos em uma mesma célula devido à resolução.

Figura 26 – Ilustrações do processo de criação e aplicação da grade regular.




(a) Criação de uma grade regular no *QGIS*.



(b) Grade regular sobreposta ao raster de uso do solo.

Fonte: Autoria própria.

### 6.5.2 Integração com Dados Geográficos

Após a criação da grade, utiliza-se a função *fill* com a estratégia *zonal_stats* para associar atributos geográficos às células, como classes de uso do solo, elevação média e tipos de solo. Na Tabela 9 é resumido as integrações realizadas, em que cada uma delas é detalhada a seguir.

Tabela 9 – Resumo das integrações de dados geográficos.

| **Tipo de Dado** | **Estatística** | **Atributo Gerado** | **Aplicação** |
|---|---|---|---|
| Uso do solo | Majority | *uso_majority* | Condição inicial ou restrição |
| Elevação | Mean | *mde_mean* | Regras de transição altimétricas |
| Tipos de solo | Majority | *solo_majority* | Classificação do solo dominante |

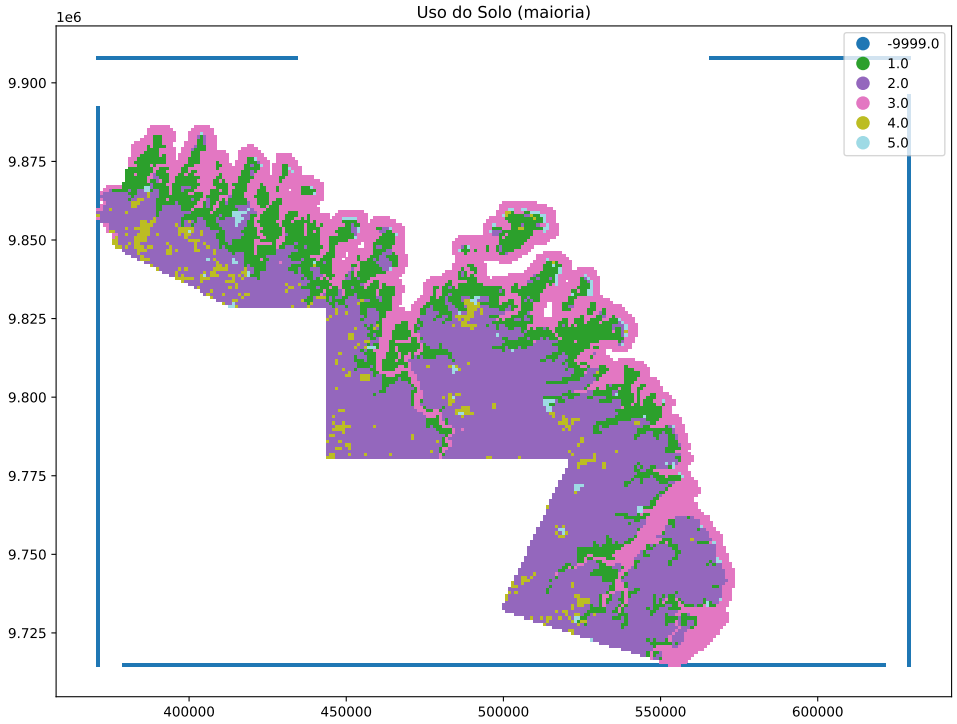
Para dados de uso do solo, a função *fill* associa a classe majoritária de um raster do *MapBiomas* a cada célula, conforme apresentado no Código 35.

**Código 35** Integração de dados de uso do solo com estatística zonal.

```
from dissmodel.geo import fill

fill(
    strategy="zonal_stats",
    vectors=grid,
    raster_data=raster_mapbiomas,
    nodata=nodata_mapbiomas,
    affine=affine_mapbiomas,
    stats=["majority"],
    prefix="uso_"
)
```

O código calcula a classe majoritária de uso do solo em cada célula, armazenando o resultado no atributo *uso_majority*. Esse atributo serve como condição inicial ou restrição em simulações de ocupação ou conservação. Na Figura 27 é ilustrado a visualização do atributo *uso_majority* na grade.

Figura 27 – Visualização do atributo *uso_majority* na grade.




Fonte: Autoria própria.

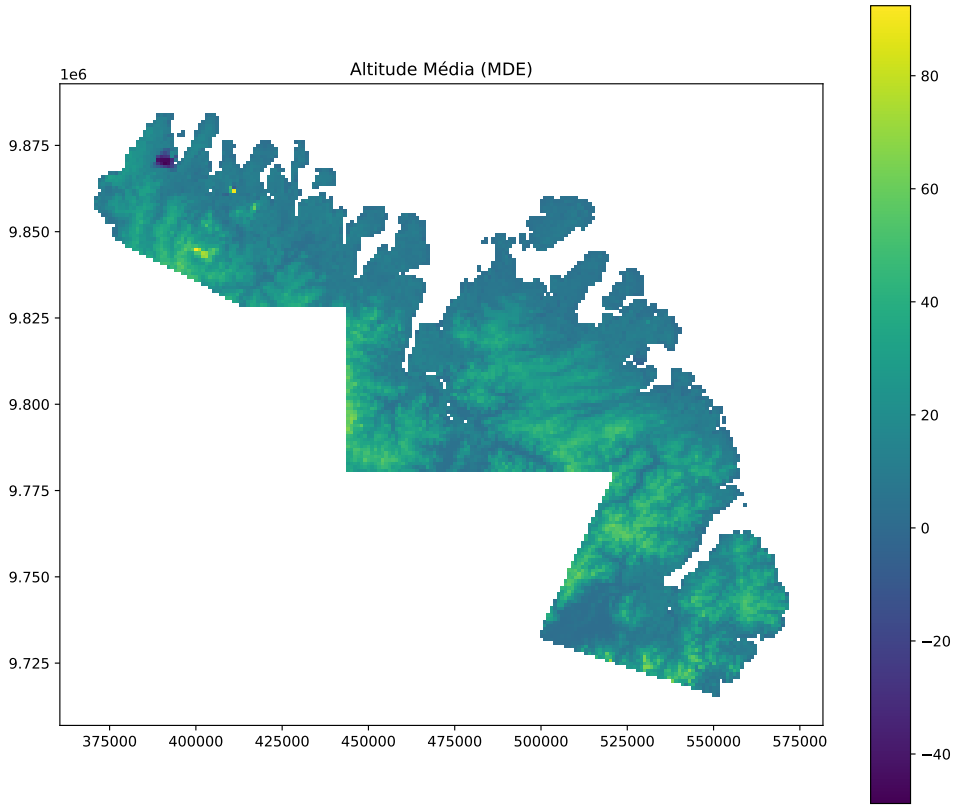
Para dados de elevação, a função *fill* calcula a média altimétrica por célula a partir de um modelo digital de elevação (MDE), conforme apresentado no Código 36.

**Código 36** Integração de dados de elevação com estatística zonal.

```
from dissmodel.geo import fill

fill(
    strategy="zonal_stats",
    vectors=grid,
    raster_data=raster_mde,
    nodata=nodata_mde,
    affine=affine_mde,
    stats=["mean"],
    prefix="mde_"
)
```

O código calcula a média altimétrica em cada célula, armazenando o resultado no atributo *mde_mean*, que pode influenciar regras de transição em modelos baseados em altitude. Na Figura 28 é ilustrado a visualização do atributo *mde_mean* na grade.

Figura 28 – Visualização do atributo *mde_mean* na grade.




Fonte: Autoria própria.

Para dados vetoriais de solo, como shapefiles ou geopackages, é necessária uma conversão inicial para raster, seguida da aplicação de estatística zonal. No Código 37 é

apresentado a conversão de categorias de solo em códigos numéricos e sua rasterização.

**Código 37** Conversão de dados vetoriais de solo para raster.

```
from rasterio.features import rasterize

# Converter categorias de solo para códigos numéricos
gdf_solo["solo_id"] = gdf_solo["grande_gru"].astype("category").cat.codes + 1

# Rasterizar vetor de solos
shapes = ((geom, value) for geom, value in zip(gdf_solo.geometry,
↪  gdf_solo["solo_id"]))
raster_solo = rasterize(
    shapes=shapes,
    out_shape=(height, width),
    fill=0,
    transform=transform_solo,
    dtype='int32'
)
```

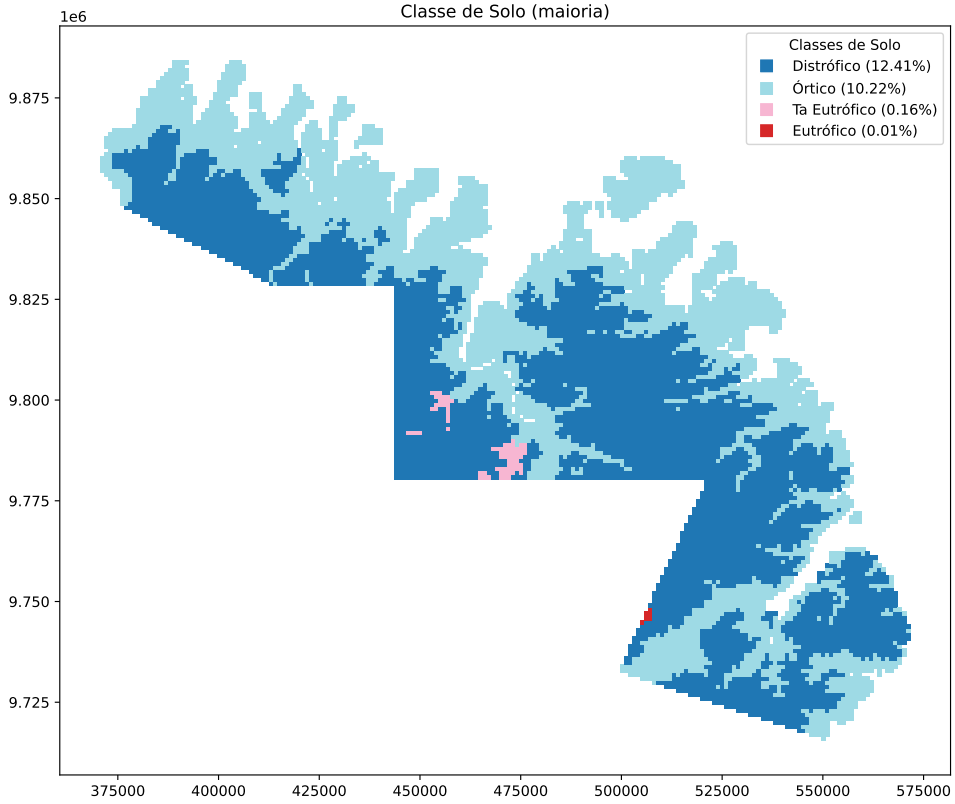
No Código 38 agrega-se a classe de solo dominante à grade.

**Código 38** Agregação de dados de solo à grade com estatística zonal.

```
from dissmodel.geo import fill

fill(
    strategy="zonal_stats",
    vectors=grid,
    raster_data=raster_solo,
    nodata=0,
    affine=transform_solo,
    stats=["majority"],
    prefix="solo_"
)
```

O código associa a classe de solo dominante a cada célula, armazenando o resultado no atributo *solo_majority*. Na Figura 29 é possível ver o atributo *solo_majority* na grade.

Figura 29 – Visualização do atributo *solo_majority* na grade.




Fonte: Autoria própria.

A grade enriquecida com atributos geográficos pode ser exportada para formatos vetoriais padrão, como apresentado no Código 39.

**Código 39** Salvamento da grade enriquecida em formato Shapefile.

```
1 grid.to_file("grade_brmangue_completa.shp")
```

A integração descrita demonstra a capacidade do *framework DisSModel* de combinar múltiplas camadas de dados geográficos em modelos espacialmente explícitos. Essa abordagem é essencial para simular fenômenos complexos, como o deslocamento de ecossistemas de manguezal no modelo *BRMangue*, que utiliza essas técnicas para representar dinâmicas ambientais em resposta a mudanças climáticas.

## 6.6 Comparação entre TerraME, o Ecossistema Python e DisSModel

Os resultados evidenciam que o *framework* TerraME, amplamente utilizado por pesquisadores do Instituto Nacional de Pesquisas Espaciais (INPE), é uma ferramenta

importante para modelagem dinâmica espacial. Seu diferencial é a integração nativa de simulações baseadas em autômatos celulares, agentes e eventos discretos, estruturados em grades fixas por meio do `CellularSpace`. No entanto, o uso da linguagem Lua para definição de modelos, embora simples para prototipagem, limita a integração com bibliotecas modernas de análise de dados e visualização, além de contar com uma comunidade mais restrita e atualizações menos frequentes — sendo o último release em agosto de 2020.

Em contrapartida, o ecossistema Python reúne bibliotecas amplamente testadas, atualizadas e mantidas por grandes comunidades de software livre, potencializando a integração de fluxos de trabalho complexos envolvendo modelagem, análise espacial e visualização interativa. Para simulações de eventos discretos, por exemplo, `SimPy` e `Salabim` disponibilizam recursos modernos como paralelização e animações em tempo real, conforme ilustrado no Código 40.

**Código 40** Exemplo de modelagem orientada a objetos com Salabim

```
1  import salabim as sim
2
3  class Agent(sim.Component):
4      def setup(self, energy):
5          self.energy = energy
6      def process(self):
7          while True:
8              self.energy -= 1
9              self.hold(1)
10             if self.energy <= 0:
11                 self.cancel()
```

Na manipulação de dados espaciais, bibliotecas como `GeoPandas` e `Rasterio` viabilizam o trabalho integrado com dados vetoriais e raster, enquanto o `PySAL` disponibiliza estratégias de vizinhança avançadas (Queen, Rook, Kernel), superando as opções mais restritas do TerraME. Para visualização, `Matplotlib`, `Folium` e `Streamlit` permitem desde gráficos estáticos até dashboards dinâmicos, como mostrado no Código 41.

**Código 41** Exemplo de visualização de grade de infecção com Matplotlib

```
import matplotlib.pyplot as plt

infection_grid = [[cell.infection for cell in row] for row in grid]
plt.imshow(infection_grid, cmap='Reds')
plt.colorbar(label='Nível de Infecção')
plt.savefig('infection_map.png')
```

Nesse cenário, o *DisSModel* surge como uma alternativa em Python, capaz de aproveitar toda a infraestrutura existente de bibliotecas robustas, sem depender de *scripts* em Lua. Ao adotar classes Python nativas e integrar ferramentas como `GeoPandas`, `PySAL`, `Salabim` e `Streamlit`, o DisSModel amplia as possibilidades de extensão e reutilização em aplicações como mudanças de uso da terra, propagação de epidemias ou outros processos dinâmicos espaciais, beneficiando-se de uma linguagem popular e de uma comunidade ativa.

Na Tabela 10 é resumido os principais aspectos comparativos entre o TerraME e as alternativas disponíveis no ecossistema Python, evidenciando os pontos fortes e limitações de cada abordagem.

Tabela 10 – Comparação entre TerraME e Ecossistema Python.

| **Aspecto** | **TerraME** | **Ecossistema Python** |
|---|---|---|
| Estrutura Espacial | `CellularSpace` (grade fixa) | `GeoDataFrame` (vetorial, flexível) |
| Tipos de Dados | Vetoriais (shapefiles, bancos de dados) | Vetoriais + raster (`Rasterio`) |
| Vizinhança | Moore, von Neumann, GPM | `PySAL` (Queen, Rook, Kernel) |
| Simulação Dinâmica | Nativo (`Event`, `Timer`) | `Salabim`, `SimPy`, `PyCX` |
| Integração SIG | `TerraLib` | `GeoPandas`, `GDAL` |
| Visualização | `Map`, `Chart` (exportação para GIS) | `Matplotlib`, `Folium`, `Streamlit` |

Observa-se que, enquanto o TerraME oferece recursos específicos e consolidados para determinados tipos de simulação dinâmica, o DisSModel se posiciona como uma alternativa mais flexível e atualizada, explorando ao máximo o potencial de bibliotecas Python consolidadas, com maior interoperabilidade, suporte comunitário e integração a fluxos de trabalho modernos de análise de dados e visualização.

# 7 Conclusão

Este trabalho apresentou o desenvolvimento do *framework* DisSModel, uma solução modular, acessível e extensível para modelagem dinâmica espacialmente explícita no ecossistema Python. Motivado pelas limitações identificadas no TerraME — especialmente sua dependência da linguagem Lua e sua restrita integração com fluxos de trabalho modernos em ciência de dados —, o DisSModel buscou reinterpretar e adaptar suas principais funcionalidades, reunindo paradigmas como autômatos celulares, modelos baseados em agentes e sistemas dinâmicos em uma arquitetura flexível, apoiada por bibliotecas robustas e amplamente utilizadas pela comunidade científica.

A metodologia empregada, estruturada em etapas de análise, projeto, implementação e validação, possibilitou a consolidação de um conjunto de ferramentas capaz de integrar dados vetoriais e raster, manipular vizinhanças espaciais complexas e realizar simulações comparáveis às do TerraME, mas com ganhos significativos em termos de interoperabilidade e reprodutibilidade. Os estudos de caso demonstraram a versatilidade do DisSModel na representação de fenômenos socioambientais dinâmicos — como propagação de incêndios, padrões espaciais emergentes e processos epidemiológicos — reforçando seu potencial como alternativa aberta e escalável para pesquisas em LUCC e outras aplicações.

Além de ampliar o acesso a técnicas de modelagem espacial dinâmica, o DisSModel evidencia o potencial do ecossistema Python para suportar fluxos de trabalho integrados, desde a manipulação de grandes volumes de dados geoespaciais até a visualização interativa de resultados. Ao se beneficiar de uma linguagem amplamente adotada e de bibliotecas mantidas por comunidades ativas, o DisSModel também contribui para aproximar pesquisadores de diferentes áreas, democratizando o desenvolvimento de modelos complexos e facilitando sua replicação em contextos variados.

Como perspectivas futuras, destaca-se a possibilidade de expandir o suporte a novos paradigmas, explorar mecanismos de paralelização para grandes áreas de estudo e integrar o DisSModel a ambientes de computação em nuvem e plataformas de ciência aberta. Além disso, planeja-se disseminar o *framework* por meio de publicações, workshops e repositórios abertos, incentivando sua adoção pela comunidade científica. Um plano de longo prazo para manutenção do sistema será desenvolvido, capacitando usuários a contribuir com melhorias e atualizações. Para reforçar a robustez do DisSModel, serão realizadas comparações mais precisas com outros *frameworks*, bem como avaliações detalhadas de escalabilidade em cenários de grande volume de dados. A inclusão de medidas de incerteza e a realização de estudos de sensibilidade também estão previstas, visando aprimorar a confiabilidade e a interpretação dos modelos gerados. Tais aprimoramentos

visam potencializar o uso do DisSModel em políticas públicas, planejamento territorial e estudos voltados à sustentabilidade, tornando a modelagem de mudanças no uso e cobertura da terra mais transparente, colaborativa e acessível a diferentes perfis de usuários.

Em síntese, o DisSModel representa um avanço significativo na modelagem dinâmica espacialmente explícita no contexto do Python, consolidando-se como uma ferramenta promissora para apoiar pesquisas que buscam entender e antecipar transformações em sistemas socioambientais complexos.

# Referências

AGUIAR, A. P. D. et al. Luccme-terrame: an open-source framework for spatially explicit land use change modelling. *GLP News*, v. 8, n. 8, p. 21–23, 2012. Citado na página 12.

AGUIAR, A. P. D.; VIEIRA, I. C. G.; ASSIS, T. O.; DALLA-NORA, E. L.; TOLEDO, P. M.; SANTOS-JUNIOR, R. A. O.; BATISTELLA, M.; COELHO, A. S.; SAVAGET, E. K.; ARAGÃO, L. E. O. C. et al. Land use change emission scenarios: anticipating a forest transition process in the brazilian amazon. *Global change biology*, Wiley Online Library, v. 22, n. 5, p. 1821–1840, 2016. Citado na página 12.

ANDRADE, P.; CâMARA, G.; MARETTO, R.; MONTEIRO, A.; CARNEIRO, T.; FEITOSA, F. Experiences with a socio-environmental modeling course. *Modelling in Science Education and Learning*, v. 8, p. 71, 01 2015. Citado na página 18.

BEZERRA, F. G. S.; RANDOW, C. V.; ASSIS, T. O.; BEZERRA, K. R. A.; TEJADA, G.; CASTRO, A. A.; GOMES, D. M. d. P.; AVANCINI, R.; AGUIAR, A. P. New land-use change scenarios for brazil: Refining global ssps with a regional spatially-explicit allocation model. *PLOS ONE*, Public Library of Science, v. 17, n. 4, p. 1–17, 04 2022. Disponível em: <https://doi.org/10.1371/journal.pone.0256052>. Citado 2 vezes nas páginas 4 e 71.

BROWN, C.; BAKAM, I.; SMITH, P.; MATTHEWS, R. An agent-based modelling approach to evaluate factors influencing bioenergy crop adoption in north-east scotland. *Gcb Bioenergy*, Wiley Online Library, v. 8, n. 1, p. 226–244, 2016. Citado na página 32.

CÂMARA, G.; DAVIS, C.; MONTEIRO, A. M. V.; D'ALGE, J. Introdução à ciência da geoinformação. *São José dos Campos: INPE*, v. 345, 2001. Citado na página 33.

CARNEIRO, T. G. de S.; ANDRADE, P. R. de; CâMARA, G.; MONTEIRO, A. M. V.; PEREIRA, R. R. An extensible toolbox for modeling nature–society interactions. *Environmental Modelling & Software*, v. 46, p. 104–117, 2013. ISSN 1364-8152. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1364815213000534>. Citado 9 vezes nas páginas 12, 18, 19, 24, 25, 26, 28, 30 e 32.

CHEN, Y.-C.; LU, P.-E.; CHANG, C.-S.; LIU, T.-H. A time-dependent sir model for covid-19 with undetectable infected persons. *IEEE Transactions on Network Science and Engineering*, v. 7, n. 4, p. 3279–3294, 2020. Citado na página 30.

CHIARADONNA, S.; JEVTIć, P.; STERNER, B. Mpat: Modular petri net assembly toolkit. *SoftwareX*, v. 28, p. 101913, 2024. ISSN 2352-7110. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2352711024002838>. Citado na página 39.

COELHO, C. G. C. *Modelo de Simulação baseado em Agentes Interativos com Teoria de Jogos para Uso e Cobertura da Terra*. Tese (Doutorado) — Universidade de Brasília, 2021. Citado na página 31.

CORDEIRO, E. A.; PITOMBEIRA-NETO, A. R. Deep reinforcement learning for the dynamic vehicle dispatching problem: An event-based approach. *arXiv preprint arXiv:2307.07508*, 2023. Citado na página 39.

DOU, Y.; MILLINGTON, J. D.; SILVA, R. F. B. D.; MCCORD, P.; VIÑA, A.; SONG, Q.; YU, Q.; WU, W.; BATISTELLA, M.; MORAN, E. et al. Land-use changes across distant places: design of a telecoupled agent-based model. *Journal of Land Use Science*, Taylor & Francis, v. 14, n. 3, p. 191–209, 2019. Citado na página 32.

GOODCHILD, M. F.; LONGLEY, P. A. Geographic information science. In: *Handbook of Regional Science*. [S.l.]: Springer, 2021. p. 1597–1614. Citado na página 16.

HAEDRICH, C.; PETRAS, V.; PETRASOVA, A.; BLUMENTRATH, S.; MITASOVA, H. Integrating grass gis and jupyter notebooks to facilitate advanced geospatial modeling education. *Transactions in GIS*, John Wiley and Sons Inc, v. 27, n. 3, p. 686 – 702, 2023. ISSN 13611682. Cited by: 1; All Open Access, Hybrid Gold Open Access. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85149451746&doi=10.1111%2ftgis.13031&partnerID=40&md5=a3db9206b894f56be76d3e57bc02f3f1>. Citado na página 19.

Instituto Nacional de Pesquisas Espaciais (INPE). *Monitoramento do Desmatamento da Floresta Amazônica Brasileira por Satélite*. 2020. Disponível em: <http://www.obt.inpe.br/OBT/assuntos/programas/amazonia/prodes>. Citado na página 12.

JORDAHL, K.; BOSSCHE, J. Van den; WASSERMAN, J.; MCBRIDE, J.; GERARD, J.; TRATNER, J.; PERRY, M.; FARMER, C. geopandas/geopandas: v0. 5.0. *Zenodo*, 2021. Citado 4 vezes nas páginas 19, 33, 34 e 40.

KAISER, K. E.; FLORES, A.; VERNON, C. R. Janus: a python package for agent-based modeling of land use and land cover change. *Journal of Open Research Software*, Pacific Northwest National Laboratory (PNNL), Richland, WA (United States), v. 8, n. PNNL-SA-148545, 2020. Citado na página 19.

LAMBIN, E. F.; GEIST, H. J.; LEPERS, E. Dynamics of land-use and land-cover change in tropical regions. *Annual Review of Environment and Resources*, Annual Reviews, v. 28, n. Volume 28, 2003, p. 205–241, 2003. ISSN 1545-2050. Disponível em: <https://www.annualreviews.org/content/journals/10.1146/annurev.energy.28.050302.105459>. Citado na página 18.

LANG, S.; KUETGENS, M.; REICHARDT, P.; REGGELIN, T. Modeling production scheduling problems as reinforcement learning environments based on discrete-event simulation and openai gym. *IFAC-PapersOnLine*, Elsevier, v. 54, n. 1, p. 793–798, 2021. Citado 3 vezes nas páginas 18, 34 e 35.

LANG, S.; KUETGENS, M.; REICHARDT, P.; REGGELIN, T. Modeling production scheduling problems as reinforcement learning environments based on discrete-event simulation and openai gym. *IFAC-PapersOnLine*, v. 54, n. 1, p. 793–798, 2021. ISSN 2405-8963. 17th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2021. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2405896321008399>. Citado na página 26.

LANG, S.; REGGELIN, T.; MüLLER, M.; NAHHAS, A. Open-source discrete-event simulation software for applications in production and logistics: An alternative to commercial tools? *Procedia Computer Science*, v. 180, p. 978–987, 2021. ISSN 1877-0509. Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020). Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050921004038>. Citado na página 39.

LEMOS, C. M. G.; MEDEIROS, L. C. de C.; RIBEIRO, K.; AVANCINI, R. Agent-based model implemented using the terrame framework to simulate the dynamic transmission of dengue fever. *Revista Geografias*, p. 85–98, 2017. Citado na página 32.

LIMA, T.; CARNEIRO, T.; FARIA, S.; SILVA, P.; PESSOA, M. Terrame gims: An eclipse plug-in for environmental modeling. In: *2013 3rd International Workshop on Developing Tools as Plug-Ins (TOPI)*. [S.l.: s.n.], 2013. p. 37–42. Citado na página 12.

LIMA, T. F. M. de. Terra me gims-uma interface gráfica para a descrição de modelos ambientais para a plataforma terrame. Universidade Federal de Minas Gerais, 2010. Citado na página 29.

MAS, J.-F.; KOLB, M.; PAEGELOW, M.; OLMEDO, M. T. C.; HOUET, T. Inductive pattern-based land use/cover change models: A comparison of four software packages. *Environmental Modelling & Software*, Elsevier, v. 51, p. 94–111, 2014. Citado na página 19.

PINTO, L. V.; INáCIO, M.; GOMES, E.; PEREIRA, P. A protocol to model future land use scenarios using dinamica-ego. *MethodsX*, v. 14, p. 103283, 2025. ISSN 2215-0161. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2215016125001293>. Citado na página 19.

PONTIUS, R. G.; CASTELLA, J.-C.; NIJS, T. D.; DUAN, Z.; FOTSING, E.; GOLDSTEIN, N.; KOK, K.; KOOMEN, E.; LIPPITT, C. D.; MCCONNELL, W. et al. Lessons and challenges in land change modeling derived from synthesis of cross-case comparisons. *Trends in spatial analysis and modelling: Decision-support and planning strategies*, Springer, p. 143–164, 2018. Citado 2 vezes nas páginas 12 e 18.

QIANG, Y.; LAM, N. S. Modeling land use and land cover changes in a vulnerable coastal region using artificial neural networks and cellular automata. *Environmental monitoring and assessment*, Springer, v. 187, p. 1–16, 2015. Citado na página 30.

REN, Y.; Lü, Y.; COMBER, A.; FU, B.; HARRIS, P.; WU, L. Spatially explicit simulation of land use/land cover changes: Current coverage and future prospects. *Earth-Science Reviews*, v. 190, p. 398–415, 2019. ISSN 0012-8252. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0012825218300059>. Citado na página 18.

RIMAL, B.; ZHANG, L.; KESHTKAR, H.; HAACK, B. N.; RIJAL, S.; ZHANG, P. Land use/land cover dynamics and modeling of urban land expansion by the integration of cellular automata and markov chain. *ISPRS International Journal of Geo-Information*, MDPI, v. 7, n. 4, p. 154, 2018. Citado na página 31.

ROJAS, C.; LINFATI, R.; SCHERER, R. F.; PRADENAS, L. Using geopandas for locating virtual stations in a free-floating bike sharing system. *Heliyon*, Elsevier, v. 9, n. 1, 2023. Citado na página 34.

ROMANELLO, M.; WALAWENDER, M.; HSU, S.-C.; MOSKELAND, A.; PALMEIRO-SILVA, Y.; SCAMMAN, D.; ALI, Z.; AMELI, N.; ANGELOVA, D.; AYEB-KARLSSON, S. et al. The 2024 report of the lancet countdown on health and climate change: facing record-breaking threats from delayed action. *The Lancet*, Elsevier, v. 404, n. 10465, p. 1847–1896, 2024. Citado na página 12.

SAJAN, B.; MISHRA, V. N.; KANGA, S.; MERAJ, G.; SINGH, S. K.; KUMAR, P. Cellular automata-based artificial neural network model for assessing past, present, and future land use/land cover dynamics. *Agronomy*, MDPI, v. 12, n. 11, p. 2772, 2022. Citado 2 vezes nas páginas 12 e 31.

TARIQ, A.; MUMTAZ, F. A series of spatio-temporal analyses and predicting modeling of land use and land cover changes using an integrated markov chain and cellular automata models. *Environmental Science and Pollution Research*, Springer, v. 30, n. 16, p. 47470–47484, 2023. Citado na página 31.

TURNER, B. L.; SKOLE, D.; SANDERSON, S.; FISCHER, G.; FRESCO, L.; LEEMANS, R. Land-use and land-cover change: science/research plan. *IGBP Report*, Scanning Microscopy International, jan 1995. ISSN 0586-5581. Citado na página 12.

VARNIER, M.; WEBER, E. J. Evaluating the accuracy of land-use change models for predicting vegetation loss across brazilian biomes. *Land*, v. 14, n. 3, 2025. ISSN 2073-445X. Disponível em: <https://www.mdpi.com/2073-445X/14/3/560>. Citado na página 19.

VELDKAMP, A.; LAMBIN, E. Predicting land-use change. *Agriculture, Ecosystems & Environment*, v. 85, n. 1, p. 1–6, 2001. ISSN 0167-8809. Predicting Land-Use Change. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167880901001992>. Citado na página 18.

VERBURG, P. H.; KOK, K.; JR, R. G. P.; VELDKAMP, A. Modeling land-use and land-cover change. In: *Land-use and land-cover change: local processes and global impacts*. [S.l.]: Springer, 2006. p. 117–135. Citado na página 18.

XU, X.; DU, Z.; ZHANG, H. Integrating the system dynamic and cellular automata models to predict land use and land cover change. *International journal of applied earth observation and geoinformation*, Elsevier, v. 52, p. 568–579, 2016. Citado na página 31.