flame: A Framework for Learning in Agent-based ModEls

Ayush Chopra*†
Massachusetts Institute of Technology
Cambridge, USA
ayushc@mit.edu

Jayakumar Subramanian*
Adobe
Noida, India
jasubram@adobe.com

Balaji Krishnamurthy
Adobe
Noida, India
kbalaji@adobe.com

Ramesh Raskar
Massachusetts Institute of Technology
Cambridge, USA
raskar@media.mit.edu

ABSTRACT

Agent-based models (ABMs) are discrete simulators comprising agents that act and interact in a computational world. Despite wide applicability, infrastructure for ABMs has been fragmented and lacks a standard framework to integrate benefits of recent computing advances, especially in machine learning and automatic differentiation (autograd). To alleviate this gap we introduce **flame**: a framework to define, simulate and optimize differentiable agent-based models. First, flame introduces a domain-specific language that describes ABMs with stochastic dynamics across several domains and can be implemented using abstractions of autograd. Second, flame models can execute simulations on GPU, process millions of interactions per second and seamlessly scale from few hundred agents to million-size populations. Third, flame provides custom utilities to implement fully differentiable ABMs which can benefit from gradient-based learning and integrate with deep neural networks (DNNs), in several ways. Specifically, ABMs can now use supervised and reinforcement learning to calibrate simulation parameters, optimize agent actions and learn expressive interaction rules. Finally, flame is easily accessible with a simple Python API. We validate flame through multiple case studies that study tissue morphogenesis over bio-electric networks, infectious disease epidemiology over physical networks and opinion dynamics over social networks. We hope flame can ignite further innovation at the intersection of AI and ABMs. Our code is here.

KEYWORDS

Differentiable Agent-based Modeling; Deep Neural Networks; Automatic Differentiation

ACM Reference Format:
Ayush Chopra, Jayakumar Subramanian, Balaji Krishnamurthy, and Ramesh Raskar. 2024. flame: A Framework for Learning in Agent-based ModEls. In *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), Auckland, New Zealand, May 6 – 10, 2024*, IFAAMAS, 9 pages.

*Equal contribution.
†Corresponding author

Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), N. Alechina, V. Dignum, M. Dastani, J.S. Sichman (eds.), May 6 – 10, 2024, Auckland, New Zealand. © 2024 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

1 INTRODUCTION

Agent-based models (ABMs) [9] are discrete simulators that comprise a collection of agents that can act and interact within a computational world. They can explicitly represent the heterogeneity of an interacting population via underlying contact networks and model the adaptability of individual agent behavior for more realistic simulations. This enables domain experts to ground simulations in mechanistic understanding and explore the emergent effects of agent behavior and external interventions. ABMs are used to simulate heterogeneous systems across biological [19, 20, 32], physical [5, 14, 41], digital [2, 18, 30] and financial [22, 33] realms. For instance, ABMs have helped simulate: i) cells in a tumor microenvironment to evaluate antibody treatments for tumor suppression [20], ii) diseased humans in the physical world to decide lockdown strategies [26] and prioritize vaccination schedules [39], iii) avatars in a digital environment to counter misinformation [10] and vaccine hesitancy [2] and iv) firms in a financial network to predict housing market crashes [33]. Despite wide applicability, the adoption of ABMs for general-purpose decision-making has been scarce which can largely be attributed to computational constraints.

Conventional ABM frameworks [25, 45], while easy to use, are very slow to execute, difficult to scale to million-size populations, tough to calibrate, and only enable modeling with simple handcrafted rules. Some works have sought to alleviate performance bottlenecks through high-performance clusters [8] or customized C++ code [21]. However, these implementations are difficult to use and generalize into an accessible modeling framework; and are not compatible with data-driven machine learning.

Motivated by parallel efforts in differentiable scientific computation for molecular dynamics [16, 40], computational chemistry [43], and fluid dynamics [15], some recent works have sought to achieve highly performant ABMs by making them compatible with automatic differentiation (autograd). These *differentiable ABMs* [4, 14] have shown promising results to accelerate simulations on CPUs and GPUs [13], improve calibration using heterogeneous data by integrating with DNNs [14], learn expressive rule sets via neural model specification [32] and accelerate sensitivity analyzes with gradients [37]. While interesting proofs-of-concept, these benefits have been restricted to specific ad-hoc implementations as no general framework exists to design differentiable ABMs. This is because: i) standard ABM frameworks (Mesa, NetLogo, MASON, Agents.jl) are not designed to support differentiable simulation and interventions over millions of agents or integrate with DNNs; ii)

	Tissue Morphogenesis (Biological)	Spatial Epidemiology (Physical)	Opinion Dynamics (Digital)
Agents	Cells	Citizens	Marketers, Consumers
Objects	N/A	Virus (R0, generation-time) Pubs (lat-long, capacity)	Products (quality, cost)
Environment	2D grid space (cell-cell)	Real-world contact graph (citizen-pub, citizen-citizen)	Random graph (consumer-consumer)
Scale	400 agents; 1000 steps	7.5 million agents; 180 steps	8100 agents; 100 steps
Substeps	EvolveCell	TwoDoseVaccination NewTransmission SEIRMProgression	PurchaseProduct
Learnable	*Agent evolution rule*	*Object properties*	*Agent action policy*
Technique	Embed DNN inside ABM	End-to-End integrate DNN with ABM	Optimize scalar ABM parameters using autograd
Algorithm	Supervised Learning	Supervised Learning	Reinforcement Learning

Figure 1: `flame` can be used to define diverse ABMs across biological, digital, and physical realms; execute million-scale simulation; and use gradient-based learning and integrate with DNNs. This enables ABMs to leverage supervised learning and reinforcement learning to calibrate simulation properties, optimize agent actions and learn expressive interaction rules.

autograd frameworks (Pytorch, JAX) are optimized for training large DNN models and, not tailored for executing ABMs with mechanistic dynamics and interventions. `flame` alleviates this gap as a framework that can describe expressive ABMs and, execute them using the capabilities of autograd.

`flame` is guided by four key design principles. 1) **flexible definition**: `flame` allows the definition of complex ABMs with sequence of stochastic dynamics and multiple interventions, generalizes across several disciplines, and is intended to be a viable toolkit for both scientific exploration and real-world decision making. 2) **scalable execution**: `flame` is engineered to execute on both CPUs and GPUs, process millions of inter-agent interactions per second on commodity hardware, and scale from handling populations of a few hundred agents in a synthetic grid to managing millions of agents in complex, real-world contact graphs. 3) **learning-first design**: `flame` models are fully differentiable which allows them to utilize gradient-based learning and integrate with DNNs, in several ways. ABMs designed with `flame` can use supervised and reinforcement learning to calibrate simulation parameters, optimize agent actions, or learn interaction rules. 4) **easy accessibility**: `flame` provides a simple Python-API to define and simulate these million-scale ABMs and facilitates quick integration with Pytorch (in 3 lines of code) to leverage autograd or build hybrid DNN-ABM pipelines. We validate `flame` through multiple case studies which involve learning the purchase behavior of consumers on social platforms, simulating inter-cellular interactions for tissue formation, and designing immunization policies against infectious disease for 6.5 million people. Our code is open-source at: github.com/AgentTorch/AgentTorch

2 RELATED WORK

Automatic differentiation is becoming integral to scientific computation for faster and data-driven simulations. This is being enabled by modern simulation frameworks that tailor domain-specific understanding with the computational abilities of autograd and neural networks. Some examples include JAX.MD [40] and TorchMD [16] for molecular dynamics, TorchDyn [36] for neural differential equations, JAXFluids [7] for fluid dynamics. These have unlocked highly performant applications across atmospheric modeling[12], catalytic discovery [43], protein modeling [1, 23], computational finance [11] and rigid body dynamics [15, 17]. For agent-based modeling, some recent works have demonstrated the utility of automatic differentiation [4, 14] to accelerate simulations [13], improve calibration by integrating with deep neural networks [14, 38], conduct one-shot sensitivity analysis using gradients [37], and replace mechanistic rules with neural networks [32, 34]. However, this has been restricted to a few ad-hoc examples, and no general-purpose framework exists to design differentiable ABMs. Designing such a framework for ABMs presents unique challenges due to multi-scale dynamics, stochastic interventions, and the need to support diverse applicability; all while ensuring compatibility with autograd.

Conventional ABM frameworks [25, 29, 45] are slow to execute, don't scale to million-size populations, only enable simulating with hard-crafted rules, and hence are incompatible with machine learning. Implementations such as [8, 21] alleviate performance bottlenecks but cannot support differentiable computation or neural network integrations. [3] allows using reinforcement learning for ABMs but is restricted to a few hundred agents (no GPU acceleration) and only policy gradient methods as their simulations

are not differentiable. Some recent multi-agent learning frameworks [28, 31] support tensorization and GPU-accelerated simulations for black-box reinforcement learning, but they are not designed for ABMs (cannot describe stochastic dynamics or interventions) and are, importantly, also not differentiable. In contrast, `flame` is specifically designed to handle ABMs with stochastic dynamics and interventions, generalizes to multiple domains, scales to millions of interacting agents, and is fully differentiable which allows using both policy gradient and gradient-descent algorithms.

3 AGENT-BASED MODELING WITH `FLAME`

`flame` models agent-based simulations and interventions over large-scale populations and enables learning using automatic differentiation. First, we define a differentiable ABM. Second, we introduce the design specification to build an ABM using `flame`. Third, we provide a proof sketch and implementation primitives to validate the differentiability of each flame ABM. Finally, we describe learning with `flame` which enables building hybrid DNN-ABM pipelines.

3.1 Differentiable Agent-based Model

Consider $X_N = ABM(X_0; \theta)$ where ABM is a stochastic N-step function with input state X_0, params θ, output state X_N. ABM is differentiable if, given a smooth objective $Y = c(X_N)$, the gradients $dY/d\theta$ and dY/dX_0 can be computed using autograd. This gradient is useful for calibration of θ [14], sensitivity analysis [37] and integrating ABM with DNN [32]. Practically, this is constrained by slow execution and non-differentiable operators in the ABM. Our proposed framework `flame` resolves these by providing a composable ABM definition with utilities to implement it via differentiable operators and execute it via fast tensorization.

3.2 Definition of a `flame` model

Definition 1 (`flame` model). *A* `flame` *model is defined by the following tuple: $\langle \mathcal{S}, \mathcal{G} \rangle$. These are defined as given below. When reading the terminology below, consider the example in Figure 2, which simulates the spread of an infectious disease (like COVID-19).*

(1) $\mathcal{S} = \langle \mathcal{S}_{Ag}, \mathcal{S}_{Ob}, \mathcal{S}_{Env} \rangle$, represents the set of states of the three kinds of entities in `flame`*, which are:*
   (a) Agents which observe, act, and interact within a computational world. For instance, these can be infected citizens that spread diseases (with properties like {age, disease_stage}).
   (b) Objects which interface between agents but don't have the agency to act. For instance, these can be a virus that carries infection (with properties like {RO}) or a pub where citizens co-locate (with properties like {lat-long, capacity}).
   (c) Environments which facilitate the interactions of agents with other agents or objects. The interaction graphs are of two types: agent-agent and agent-object. For instance, a citizen-pub (agent-object) graph can represent the interaction of citizens across different pubs in a geo-locality.

   Each state property is initialized once to define the initial state and may be transformed during the simulation. This transformation is managed by the substeps defined below.

(2) Each episode or rollout of the simulation is assumed to run for T steps, where each step $t \in \{1, \ldots, T\}$ comprises multiple substeps. Thus, in `flame`*, a substep is the main repeating operational unit, and $\mathcal{G}$ represents the set of substeps within each step. Each substep is composed of the three functions:*
   (a) Substep Observation which uses the current state and returns an observation for the agents: $o : \mathcal{S} \rightarrow \mathcal{O}$, where $\mathcal{O}$ is the space of all observations. For instance, an agent can receive an observation regarding the state of infection and vaccination status of its immediate neighbors (`observation = observe_neighbors(state)`*).*
   (b) Substep Policy which uses this generated observation, along with the entire history of earlier observations (compiled as `observation_history`*) to yield the agents' actions: $\pi : \mathcal{H} \rightarrow \mathcal{A}$, where $\mathcal{H}$ is the set of trajectories of all historical observations for the agent and $\mathcal{A}$ is the set of all actions (over all agents). For instance, an agent uses its current observations of vaccination by neighbors and historical deaths to decide whether to vaccinate itself (*`action = choose_vaccine(observation_history)`*).*
   (c) Substep Transition which uses the current state and agent actions to generate the next state: $t : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. For instance, a non-vaccinated agent may choose to accept the dose (`next_state = update_vacc_status(state, action)`*).*

Note that in each substep, one or more classes of agents may interact with each other and the environment.

3.3 Differentiability of a `flame` model

Proof Sketch Each `flame` ABM is a finitely iterated, nested composition of differentiable functions where differentiability follows from the chain rule. From the definition above, ABM is composed of repeated simulation steps of function f, such that $X_{t+1} = f(X_t; \theta))$. f transforms X_t by composing k structured substeps invoked in sequence. Each substep i (=1 to k) takes the state $(X_{t,i})$ to produce a new simulation state $(X_{t,i+1})$ by composing observation (o_i), policy(p_i) and transition(g_i) functions, all substep functions(o_i, p_i, g_i) are constructed via differentiable (mechanistic and neural) torch and flame operators (section 3.6) and, hence are differentiable w.r.t their parameters and inputs. The differentiability of substep functions, via the chain rule, invokes differentiability of each substep and which implies differentiability of step function f. Since f is differentiable, an objective $Y = c(X_N, \theta)$ is differentiable with respect to the parameters θ, state X_N and by chain rule with respect to initial state X_0. A detailed proof is included in the appendix.

Implementation `flame` is built using modern autograd libraries that are compatible with hardware accelerators (GPU/TPU) to ensure differentiability and performance efficiency. In this paper, we use abstractions underlying PyTorch [35], which is a popular autograd framework. Specifically, the simulation state is described using `nn.ParameterDict` and the functions in each substep are defined using the component `nn.ModuleDict` from PyTorch. This design choice allow us to leverage in-built support for tensorized GPU execution (via BLAS/CUDA) and autograd when simulating ABMs. Furthermore, it also streamlines integration of mechanistic ABMs with DNNs to build hybrid DNN-ABM pipelines. Specifically, any substep function, independently of others, can be specified with mechanistic rules or with a deep neural network (DNN). The learning-first design of `flame` enables calibrating arguments of any rule-based mechanism using autograd and learning parameters of neural specification without affecting the rest. Each substep extends

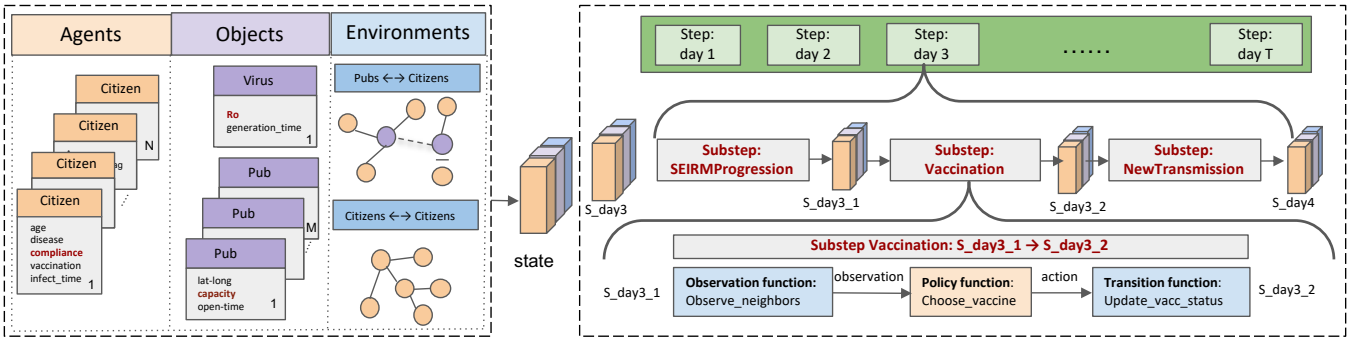

Figure 2: Defining a `flame` model for spatial epidemiology. The simulation has N citizens (Agents) that interact through direct mobility and co-locate across pubs (Object) to spread the virus (Object). The simulator state is a collection of properties that describe each of these entities, is initialized once, and transformed during T simulation steps. Each step models the disease progression of infected agents (`SEIRMProgression`), vaccination of susceptible agents (`Vaccination`), and transmission of new infections (`NewTransmission`) to recursively transform the simulation state over these substeps. `flame` is designed to ensure gradient flow through all simulation steps and enables automatic differentiation of any state property or substep function.

`nn.ModuleDict` class. This design helps `flame` simulations ensure gradient flow (and parameter tracking) through each substep and across all the steps of a simulation.

3.4 High-level modules and API

`flame` has multiple high-level modules to define models, execute simulations and track variables. `Config` and `Runner` are exposed to the user for defining the model and executing simulation episodes. Internally, these interface with `Controller` to initialize the simulator state, register control flow of substeps and track variables.

The model is defined by creating a `config = Config()` object. This `config` enables adding agents and objects, inserting interaction environments, defining metadata and creating simulation substeps. The code listing below uses the `config` to create infectious citizens agents (line 3) and infecting virus object (line 4), define citizen mobility networks (line 6), describe a infection transmission substep (line 7) and execute the simulation for 10 episodes (line 5).

```
1 from flame import Config
2 config = Config()
3 config.add_agents(name="citizens", num_citizens,
      prop_list)
4 config.add_objects(name="virus", num_strains, prop_list)
5 config.add_metadata("num_episodes", 10)
6 config.add_environment(type="agent-agent", src="from_file
      ", path="citizen_citizen.networkx")
7 conf.add_substep(name="NewTransmission",active_agents="
      citizens")
```

Listing 1: Using the `flame` `Config` API to define a model

The simulation is executed by creating `runner = Runner(config)` which links with the `Controller`. The user can invoke various `runner` functions which are described below:

- First, `runner.init()` is used to initialize all `state` properties and create a tracking registry of substep functions (`registry_dict`) via `controller.initialize()`
- Second, `runner.step()` is used to run all steps in a simulation episode. Each episode step invokes all substeps in sequence via `controller.execute_substep(state, registry_di`
- Third, `runner.reset()` is used to reinitialize the state of the simulator before the start of subsequent episodes. While the default is to just use `runner.init()`, this function is often overloaded to specify custom reset functions (as in Case Study 1 using state from prior episodes).
- Fourth, `runner.parameters()` tracks and return all learnable parameters in the simulation episode. These parameters can be properties of the simulator state or arguments of substep functions; and are used when defining optimizers.
- Finally, `runner.trajectory` tracks the simulation state across multiple steps and episodes and, is used to define loss functions and plot outputs.

The code listing below demonstrates use of `flame` to define, simulate and optimize an ABM. Further, (lines 19 and 34-37) shows how `flame` can interface with `torch` to define custom optimizers for ABM parameters and minimize user-specified objective functions.

```
1  from flame import Config, Runner
2  from torch import optim
3
4  # Step 1: define entities, metadata and substeps
5  config = Config()
6  config.add_agents(...)
7  config.add_objects(...)
8  config.add_environments(...)
9  config.add_metadata(...)
10 config.add_substeps(...)
11
12 # Step 2: create simulation instance
13 runner = Runner(config)
14
15 # Step 3: initialize simulation state and create registry
16 runner.init()
17
18 # Step 4: create optimizer using learnable simulation
       parameters
19 opt = optim.SGD(list(runner.parameters()), lr=config_lr)
20
21 for episode in range(num_episodes):
22     opt.zero_grad()
23
24     # Step 5: reset state before each episode
```

```
    runner.reset()

    # Step 6: execute all substeps in sequence
    runner.step(num_steps)

    # Step 7: read the trajectory to extract output
    trajectory = runner.trajectory
    output = generate_output(trajectory)

    # Step 8: compute loss and optimize parameters
    loss = loss_fn(output, ground_truth)
    loss.backward()
    opt.step()
```

Listing 2: Using `flame` to define simulate and optimize ABMs

3.5 Differentiating with `flame`

As stated earlier, using autograd primitives to design `flame` allows to represent an ABM with substeps of dynamics and interventions on a compute graph, which streamlines backpropagation. However, unlike DNNs on computation graphs, defining ABM substeps requires mechanistic operators (such as `torch.max`, `torch.compare`) which are conventionally non-differentiable and can cause incompatibility with autograd. Recent differentiable ABM have used straight-through estimators for discrete distributions [24] to represent stochastic dynamics [14]. However, this is insufficient to differentiate through interventions (eg: offer a vaccine if age < 60; purchase product with min price etc.). For `flame`, we generalize the straight-through-trick [6] to build a library of foundational operator such as `flame.compare, flame.max, flame.logical_and` etc. For each operator, we define a smooth approximation to obtain the gradient while using the exact function for computation. One such implementation of `flame.compare`, is given below.

```
@flame.helpers
def compare(a, b):
    '''return 1 if a>b; 0 otherwise'''
    def compare_soft(a, b, hardness=0.8):
        # approximate gradient
        return torch.sigmoid(hardness * (a - b))

    def compare_hard(a, b):
        # exact computation
        return (a > b).float()

    soft = compare_soft(a, b)
    return compare_hard(a, b) + soft - soft.detach()
```

Listing 3: Diferentiable operators like `flame.compare` help define autograd-compatible dynamics and interventions.

3.6 Gradient-based Optimization

As explained in the previous section, by only using differentiable operators, `flame` ensures gradient-flow through all substeps of the simulation and thus, is compatible with automatic differentiation. This allows using gradient-based learning to update properties of the state or arguments of any substep function. All learnable parameters across the simulator can be accessed via `runner.parameters()` and used in `torch.optim` to define custom optimizers, which can then be used to optimize any user-specified objective function. `flame` supports both supervised learning (SL) and reinforcement learning (RL) using first-order gradient estimates [42] and leverages the PyTorch API for optimization. There are three modes:

- `Mode 1`: *Optimize scalar/tensor ABM parameters.* For instance, this may involve calibrating the R0 parameter of a virus to death statistics using SL (sec 4.2) or learning a purchase policy to maximize expected utility using RL (sec 4.3). The optimizer is defined as `torch.optim.SGD(runner.parameters())`.
- `Mode 2`: *Embed DNN inside ABM to learn substep functions.* Often the substep functions in any conventional ABM are defined as simple mechanistic functions such as cellular automata rules. The purpose of the ABM is then to simulate the effect of these simple functions when applied repeatedly. However, this approach cannot facilitate learning structurally unknown substep functions, which can, in-principle be modeled using any universal function approximator. For instance, an unknown function in a substep can be parameterized with a neural network without affecting any other components of the simulator; and then the parameters of this neural network can be learned to reproduce observed simulation output using SL (sec 4.1). This neural substep function is defined in the `config` and its parameters tracked in `runner.parameters()`. The optimizer is defined as `torch.optim.SGD(runner.parameters())`.
- `Mode 3`: *Integrate ABM with DNN pipelines.* Instead of optimizing components of the simulation, a `flame` model can become an objective function and provide gradients to learn an external black-box models (`external_nn`). For instance, they may be used to jointly forecast infections across multiple counties with distinct simulators via SL (sec 4.2). The hybrid optimizers can be defined with `torch.optim.SGD(list(external_nn.parameters() + runner.parameters()))` or by using multiple optimizers for the various groups of learnable parameters.

3.7 Summary of Contributions

Our proposed framework, `flame`, allows to define, simulate and optimize million-scale agent-based models. First, we introduce a definition that make ABMs compatible with automatic differentiation (autograd). The modular structure of `flame` involves decomposing each simulation step into repeating substeps, each of which comprises of three functions. This architecture allows us to construct and execute simulations with diverse dynamics and interventions using the same building blocks and, be implemented using abstractions of autograd. Second, we provide a utils library of straight-through differentiable operators with `flame`. These operators allow differentiation of traditionally non-differentiable mechanistic and stochastic operators used in conventional ABMs. This feature enables design of end-to-end differentiable ABMs and, supports gradient-based calibration and optimization. Third, a useful consequence of having a differentiable ABM is that the entire ABM can be treated as a differentiable black-box function, which can then be integrated with any learnable pipeline using DNNs.

4 CASE STUDIES

Here, we present diverse case studies to show the flexibility of `flame` in definition, simulation, and optimization. These case studies span digital, physical, and biological realms; scale from a few hundred agents in synthetic grid spaces to millions of agents over

```
config = Config()
config.add_substep(EvolveCell, transition=MyCustomDNN())

runner = Runner(config)

opt = Optim.SGD(list(runner.parameters()))

for i in range(num_episodes):
  opt.zero_grad()
  runner.reset()

  runner.step(num_steps)

  emergent_pattern = get_cell_states(runner.trajectory['last_state'])
  supervised_loss = nn.MSELoss()(emergent_pattern, true_pattern)

  supervised_loss.backward()
  opt.step()
```

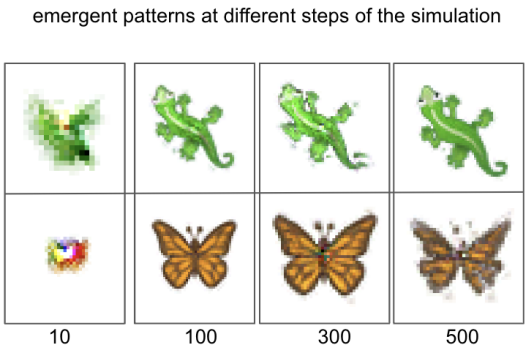

Figure 3: C1: Embed DNN inside ABM with `flame`. [32] uses cellular automata to simulate morphogenesis and parameterizes the update rules with a CNN. `flame` is used to learn these rules by representing the substep transition function with a DNN (`MyCustomDNN()`) and optimizing with an MSE loss (`supervised_loss`). Enabling this requires only a few additional lines, as shown in the pseudocode on the left, and is an instance of `Mode 2` in sec 3.6. Result, on the right, shows the emergent pattern for two shapes (lizard and butterfly) at different steps along the simulation.

city-scale contact networks; and involve learning simulation parameters, agent policies, and transition rules. Specifically, these include cells in a bio-electric micro-environment assembling organs, human citizens in a physical environment spreading infections, and avatars in a digital environment sharing opinions. The flexibility in design is coupled with computational benefits realized by tensorization, GPU execution, and support for automatic differentiation which unlocks new capabilities via seamless (end-to-end) integration with deep neural networks. For this analysis, we implement previously introduced simulators using `flame`. The key objective is to demonstrate the capabilities of the design of `flame`, which allows the specification of diverse multi-agent scenarios, executes million-scale simulations, conducts gradient-based optimization, and evaluates interventions in agent-based models.

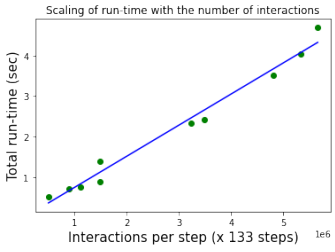

Figure 4: Benchmarking run-time performance for `flame` simulations. `flame` can simulate millions of interactions in a few seconds on commodity GPUs which demonstrates real world utility. This plot follows from section 4.2

4.1 Morphogenesis via Neural Cellular Automata

Morphogenesis is the process of an organism's shape development where cells interact over bio-electric networks to self-assemble into tissues and organs. The process is extremely robust to perturbations where several species have the ability to regenerate entire organs by repairing damage (to intermediate states) or produce viable organs even from atypical initial states. Understanding the mechanism behind morphogenesis is an active area of research and key to progress in regenerative medicine. We follow from [32], which extends cellular automata (CAs) to identify cell-level rules that result in adaptive and robust morphogenesis. CAs consist of a grid of cells that are iteratively updated with the same set of rules applied to each cell at every step. The new state of a cell depends only upon the state of a few cells in its immediate neighborhood. Conventionally, the rules in CAs are fairly simple deterministic rules. The complexity in the system modeled using CAs is an emergent property of these simple rules. However, in several real-world use cases, we may not know these rules, or these rules may themselves be fairly complex, requiring function approximation with a neural network to track state evolution using these rules. `flame` allows us to define rules of arbitrary complexity using our modular substep architecture, where the function within a rule can be a deep neural network (DNN) as well. In this case study, the goal of the simulation is twofold: a) learn the cell-level mechanism by representing it with a DNN, and b) validate the robustness of the learned mechanism to perturbations in the initial state.

Following [32], in the simulation, agents are cells with a 16-dim property state. Agent-agent interactions are described over a 2D grid space environment. The state is initialized with a single active agent, producing a multi-cellular pattern through simulation. The simulation has a single substep (`EvolveCell`) with a transition

function that is parameterized with a convolutional neural network (CNN) and describes how cells interact with neighbors to update their state. The simulation output is a 2D grid pattern of all cell states (denoting organism shape), and the learning objective is supervised mean-squared error loss with respect to a pre-specified shape (or grid pattern). The goal is to learn a transition function robust to perturbations in the initial state and involves jointly optimizing over multiple simulations with varying initial states.

`flame` demonstrates two key capabilities in this case study:

- **C1**: `flame` allows to embed DNN inside an ABM. Here, the transition function of `EvolveCell` substep is parameterized with a CNN. This is captured by `runner.parameters()` and can be used with an optimizer as shown by psuedocode in Figure 3.
- **C2**: `flame` enables joint optimization and parameter sharing between multiple ABMs. Here, multiple `runner` objects created for different initial states utilize a shared optimizer. Pseudocode and results for this experiment are included in the appendix.

4.2 Spatial Infectious Disease Epidemiology

Infectious diseases spread through contact with infected agents and have two phases: transmission to new agents and disease progression in infected agents. Modeling both phases is crucial for designing effective interventions. For COVID-19, this involved deciding lockdowns, vaccination schedules, and testing strategies. These decisions are complex, requiring consideration of population scale, individual behavior, and intervention properties. Decision-making is further complicated by delayed feedback from interventions and their non-linear interactions. Examples of such interventions include formulating and implementing effective public health policies during COVID-19, including decisions such as delaying the administration of the second vaccine dose [39], prioritizing test speed over specificity [27] etc. Evaluating these decisions in-silico requires granular and data-driven simulations, fast calibration, and sensitivity analysis. For this case study, the goals of this simulation are: a) recreate infection dynamics in real-world million-scale populations, b) improve the calibration of simulation parameters using DNNs, c) analyze the sensitivity of diverse interventions. Specifically, we model spread of COVID-19 and Influenza over 6.5 million people across 12 counties in Massachusetts, calibrate using real CDC data and validate performance by forecasting county and state-level infections consistent with CDC guidelines.

Following from [13, 14, 21, 39], in the simulation, agents are citizens with 5-dim property state (age, occupation, disease-stage, infected-time, vaccine-status) that spread Covid-19 infection. Objects include both the infecting virus and co-location centers like pubs, schools, and care homes. Environments are obtained using real-world contact graphs and describe interactions in citizen-citizen mobility networks and citizen-pub co-location networks. The simulator state is initialized with a few infected agents. Each simulation step has two substeps `InfectionTransmission` and `SEIRMProgression`, which describe the transmission of new infections and a SEIRM progression of previously infected agents, respectively. Discrete stochasticity in the simulation is handled by reparameterizing with Gumbel-softmax gradient estimator to ensure differentiability. The simulation output is the histogram of citizen disease stages, and the learning objective for calibration is a supervised loss against ground truth case statistics (from CDC).

`flame` demonstrates the following capabilities in this case study:

- **C3**: `flame` enables realistic simulations with million-scale populations and real-world contact networks, all while abstracting away the engineering complexity from the modeler. The same API scales to millions of agents and can support real-world contact graphs. The run-time performance is benchmarked in Figure 4.
- **C4**: `flame` enables composing ABMs and DNNs end-to-end. Here, this pipeline calibrates simulation parameters using gradient-based learning by designing hybrid optimizers. This is evaluated using forecasting results in figure 5(left). Corresponding pseudocodes and a pipeline visualization are included in the appendix.
- **C5**: `flame` allows flexible experimentation through its modular design. First, `flame` can evaluate policy interventions through white-box scenario analysis. Figure 5 (right) shows results that evaluate the efficacy of a delayed vaccination schedule. Second, `flame` can generalize across simulation assumptions by changing a few lines of code. Figure 5 (left) shows a model built for COVID-19 can be adapted to Influenza by just replacing a single substep (`SEIRMProgression` with `SIRSProgression`). More details about these experiments are in the appendix.

4.3 Social Opinion Dynamics

Digital interactions are already ubiquitous and have become increasingly relevant with the advent of autonomous agents. Such agents, trained to act strategically, will become integral to society and business as they redefine interfaces with humans to mitigate vaccine hesitancy against diseases, advertise new products in competitive markets etc. Typically, agents in these systems interact in two ways – directly via communicating with each other (influenced by their individual follower tendencies) and indirectly via affecting the environment or objects. We implement a standard opinion dynamics model, focusing on direct interactions, used in literature to simulate effects such as lock-in of consumer behavior to a particular product or service. In this case study, the goal of this simulation is: a) learn agent policies that maximize utility over time horizon.

Following from [18], in the simulation, consumers and marketers are the two types of agents. Objects include products that the marketers advertise to consumers. We specifically consider a duopoly with two products. Environment describes interactions through consumer-consumer networks defined using the simulation's grid graphs. The simulation has a single substep `PurchaseProduct` where agents observe the purchase behavior of neighbors to make a discrete purchase decision. The simulation output is the observed utility for all agents, and the learning objective is to maximize each agent's expected utility over the finite time horizon. Specifically, the agents wish to optimize their cumulative experienced product quality. To achieve this, they need to balance exploitation and exploration in their product purchase choices. Here, exploitation corresponds to the agents sticking to their own opinions based on their historical product usage experience, and exploration corresponds to modifying their opinions about the products using the opinions of their neighboring agents. In terms of learning, in this simulation, we make the follower tendency, i.e., the degree of exploration of the agents, learnable parameters for each agent in the

	ABM Calibration algorithm	COVID-19			Influenza		
		ND	RMSE	MAE	ND	RMSE	MAE
a	Direct autograd on parameters.(Mode 1)	2.39 ± 0.39	205.14 ± 42.56	73.6 ± 10.8	0.88 ± 0.14	2.97 ± 0.44	2.64 ± 0.43
b	Compose DNN with ABM (Mode 3)	1.15 ± 0.24	67.09 ± 23.89	35.5 ± 7.36	0.50 ± 0.19	1.78 ± 0.62	1.50 ± 0.57
c	Compose DNN with *multiple* ABM (Mode 3)	0.97 ± 0.18	50.99 ± 12.12	30.1 ± 5.60	0.41 ± 0.02	1.47 ± 0.06	1.22 ± 0.06

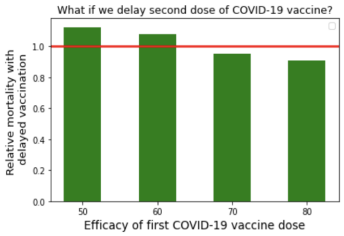

Figure 5: C3-C5: `flame` simulate ABMs with millions of agents, build hybrid DNN-ABM pipelines, and jointly optimize multiple ABMs by changing only a few lines of code. `flame` is used to calibrate ABM on 6.5 million agents and forecast the spread of two diseases - COVID-19 and Influenza over different learning situations denoted by (a), (b), and (c). This calibrated model answers policy questions, as shown on the right. The setup follows from [14] and more details, with pseudocodes, are in the appendix.

```
config = Config()
runner = Runner(config)

opt = Optim.SGD(list(runner.parameters()))

for i in range(num_episodes):
  opt.zero_grad()
  runner.reset()

  runner.step(num_steps)

  episode_return = get_agents_utility(runner.trajectory)
  policy_loss = -episode_return

  policy_loss.backward()
  opt.step()
```

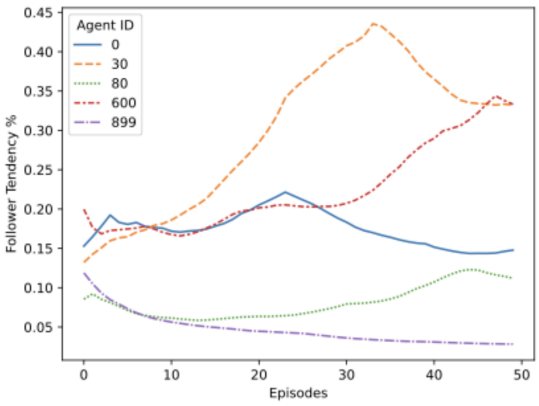

Figure 6: C6 - Policy optimization with `flame`. In social opinion dynamics [18], purchase behavior is parameterized with a follower tendency, and agents learn personalized optimal follower tendencies to maximize utilities over timesteps (`episode_return`) by minimizing `policy_loss`. This policy learning is enabled easily in `flame` with parameters in `runner` as shown in the pseudocode on the left. The learning curve with follower tendencies versus episodes for a few sample agents is shown on the right. The experimental setup primarily follows from [18], except for the learning dynamics. More details are given in the appendix.

system, which was not explored in the reference paper [18]. The discrete stochasticity in the simulation, arising from agent purchase behavior, is reparameterized with the Gumbel softmax gradient estimator and also other straight-through differentiable approximators of `max`, `min` operators in `flame`. This allows automatic differentiation, with first-order gradient estimates, through time. In principle, a score-function gradient estimate (variant of REINFORCE) can also be used, but are less performant in practice [17, 44]. The goal here to demonstrate the utility of `flame` for sequential decision-making, independent of the specific algorithm.

- **C6**: `flame` allows learning agent policies by specifying custom reward functions. End-to-end differentiability of `flame` enables reinforcement learning with first-order policy gradients [44]. Pseudocode and specific example in the context of opinion dynamics are shown in Figure 6.

5 CONCLUSION

We introduce `flame`: a framework to define, simulate and optimize agent-based models (ABMs). First, the `flame` model definition has been used to design diverse ABMs across biological, digital and physical realms; and is implemented with the primitives of autograd. Second, `flame` simulations execute on GPUs and seamlessly scale to million-scale populations. Third, autograd compatibility and custom `flame.helpers` allows simulating fully differentiable ABMs which can utilize gradient-based learning and integrate with DNNs. `flame` ABMs can use use supervised and reinforcement learning to calibrate simulation parameters, optimize agent actions and learning interaction rules. `flame` originated during COVID-19 and these capabilities have been used by clinicians and policy makers to evaluate immunization strategies that impacted millions of people.

REFERENCES

[1] Mohammed AlQuraishi. 2019. End-to-end differentiable learning of protein structure. *Cell systems* 8, 4 (2019), 292–301.

[2] Camilla Ancona, Francesco Lo Iudice, Franco Garofalo, and Pietro De Lellis. 2022. A model-based opinion dynamics approach to tackle vaccine hesitancy. *Scientific Reports* 12, 1 (2022), 11835.

[3] Leo Ardon, Jared Vann, Deepeka Garg, Thomas Spooner, and Sumitra Ganesh. 2023. Phantom-A RL-driven Multi-Agent Framework to Model Complex Systems. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*. 2742–2744.

[4] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. 2022. Automatic differentiation of programs with discrete randomness. *Advances in Neural Information Processing Systems* 35 (2022), 10435–10447.

[5] Joseph Aylett-Bullock, Carolina Cuesta-Lazaro, Arnau Quera-Bofarull, Miguel Icaza-Lizaola, Aidan Sedgewick, Henry Truong, Aoife Curran, Edward Elliott, Tristan Caulfield, Kevin Fong, et al. 2021. JUNE: open-source individual-based epidemiology simulation. *Royal Society open science* 8, 7 (2021), 210506.

[6] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

[7] Deniz A Bezgin, Aaron B Buhendwa, and Nikolaus A Adams. 2023. JAX-Fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications* 282 (2023), 108527.

[8] Keith R Bisset, Jiangzhuo Chen, Xizhou Feng, VS Anil Kumar, and Madhav V Marathe. 2009. EpiFast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems. In *Proceedings of the 23rd international conference on Supercomputing*. 430–439.

[9] Eric Bonabeau. 2002. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the national academy of sciences* 99, suppl_3 (2002), 7280–7287.

[10] Julii Brainard, PR Hunter, and Ian R Hall. 2020. An agent-based model about the effects of fake news on a norovirus outbreak. *Revue d'epidemiologie et de sante publique* 68, 2 (2020), 99–107.

[11] Luca Capriotti. 2010. Fast Greeks by algorithmic differentiation. *Available at SSRN 1619626* (2010).

[12] Gregory R Carmichael, Adrian Sandu, et al. 1997. Sensitivity analysis for atmospheric chemistry models via automatic differentiation. *Atmospheric Environment* 31, 3 (1997), 475–489.

[13] Ayush Chopra, Esma Gel, Jayakumar Subramanian, Balaji Krishnamurthy, Santiago Romero-Brufau, Kalyan S Pasupathy, Thomas C Kingsley, and Ramesh Raskar. 2021. DeepABM: scalable, efficient and differentiable agent-based simulations via graph neural networks. In *Winter Simulation Conference (WSC)*.

[14] Ayush Chopra, Alexander Rodriguez, Jayakumar Subramanian, Balaji Krishnamurthy, B Aditya Prakash, and Ramesh Raskar. 2023. Differentiable Agent-based Epidemiology.

[15] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. 2018. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems* 31 (2018).

[16] Stefan Doerr, Maciej Majewski, Adrià Pérez, Andreas Kramer, Cecilia Clementi, Frank Noe, Toni Giorgino, and Gianni De Fabritiis. 2021. Torchmd: A deep learning framework for molecular simulations. *Journal of chemical theory and computation* 17, 4 (2021), 2355–2363.

[17] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. *Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation*. http://github.com/google/brax

[18] Michael Garlick and Maria Chli. 2010. Agent-based simulation of lock-in dynamics in a duopoly. In *9th International Joint Conference on Autonomous Agents and Multiagent Systems*. 1545–1546.

[19] Voit EO Glen CM, Kemp ML. 2019. Agent-based modeling of morphogenetic systems: Advantages and challenges. *PLoS Computational Biology* (2019).

[20] Chang Gong, Oleg Milberg, Bing Wang, Paolo Vicini, Rajesh Narwal, Lorin Roskos, and Aleksander S Popel. 2017. A computational multiscale agent-based model for simulating spatio-temporal tumour immune response to PD1 and PDL1 inhibition. *Journal of the Royal Society Interface* 14, 134 (2017), 20170320.

[21] Robert Hinch, William JM Probert, Anel Nurtay, Michelle Kendall, Chris Wymant, Matthew Hall, Katrina Lythgoe, Ana Bulas Cruz, Lele Zhao, Andrea Stewart, et al. 2021. OpenABM-Covid19—An agent-based model for non-pharmaceutical interventions against COVID-19 including contact tracing. *PLoS computational biology* 17, 7 (2021), e1009146.

[22] Cars H Hommes. 2002. Modeling the stylized facts in finance through simple nonlinear adaptive systems. *Proceedings of the National Academy of Sciences* 99, suppl_3 (2002), 7221–7228.

[23] John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. 2019. Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*.

[24] Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144* (2016).

[25] Jackie Kazil, David Masad, and Andrew Crooks. 2020. Utilizing Python for Agent-Based Modeling: The Mesa Framework. In *Social, Cultural, and Behavioral Modeling*, Robert Thomson, Halil Bisgin, Christopher Dancy, Ayaz Hyder, and Muhammad Hussain (Eds.). Springer International Publishing, Cham, 308–317.

[26] Cliff C Kerr, Robyn M Stuart, Dina Mistry, Romesh G Abeysuriya, Katherine Rosenfeld, Gregory R Hart, Rafael C Núñez, Jamie A Cohen, Prashanth Selvaraj, Brittany Hagedorn, et al. 2021. Covasim: an agent-based model of COVID-19 dynamics and interventions. *PLOS Computational Biology* 17, 7 (2021), e1009149.

[27] Daniel B Larremore, Bryan Wilder, Evan Lester, Soraya Shehata, James M Burke, James A Hay, Milind Tambe, Michael J Mina, and Roy Parker. 2021. Test sensitivity is secondary to frequency and turnaround time for COVID-19 screening. *Science advances* 7, 1 (2021), eabd5393.

[28] Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. 2022. Discovered policy optimisation. *Advances in Neural Information Processing Systems* 35 (2022), 16455–16468.

[29] Sean Luke, Robert Simon, Andrew Crooks, Haoliang Wang, Ermo Wei, David Freelan, Carmine Spagnuolo, Vittorio Scarano, Gennaro Cordasco, and Claudio Cioffi-Revilla. 2019. The MASON simulation toolkit: past, present, and future. In *Multi-Agent-Based Simulation XIX: 19th International Workshop, MABS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers 19*. Springer, 75–86.

[30] Ian S Lustick et al. 2000. Agent-based modelling of collective identity: testing constructivist theory. *Journal of Artificial Societies and Social Simulation* 3, 1 (2000), 1.

[31] Kinal Mehta, Anuj Mahajan, and Pawan Kumar. 2023. marl-jax: Multi-agent Reinforcement Leaning framework for Social Generalization. *arXiv preprint arXiv:2303.13808* (2023).

[32] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. 2020. Growing Neural Cellular Automata. *Distill* (2020). https://distill.pub/2020/growing-ca/

[33] Federico Guglielmo Morelli, Michael Benzaquen, Marco Tarzia, and Jean-Philippe Bouchaud. 2020. Confidence collapse in a multihousehold, self-reflexive DSGE model. *Proceedings of the National Academy of Sciences* 117, 17 (2020), 9244–9249.

[34] Elias Najarro, Shyam Sudhakaran, Claire Glanois, and Sebastian Risi. 2022. HyperNCA: Growing developmental networks with neural cellular automata. *ICLR Workshop on Cells to Societies* (2022).

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[36] Michael Poli, Stefano Massaroli, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. 2020. Torchdyn: A neural differential equations library. *arXiv preprint arXiv:2009.09346* (2020).

[37] Arnau Quera-Bofarull, Ayush Chopra, Joseph Aylett-Bullock, Carolina Cuesta-Lazaro, Anisoara Calinescu, Ramesh Raskar, and Michael Wooldridge. 2023. Don't Simulate Twice: One-Shot Sensitivity Analyses via Automatic Differentiation. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*. 1867–1876.

[38] Arnau Quera-Bofarull, Ayush Chopra, Anisoara Calinescu, Michael Wooldridge, and Joel Dyer. 2023. Bayesian calibration of differentiable agent-based models. *ICLR Workshop on AI for Agent-based Models* (2023).

[39] Santiago Romero-Brufau, Ayush Chopra, Alex J Ryu, Esma Gel, Ramesh Raskar, Walter Kremers, Karen S Anderson, Jayakumar Subramanian, Balaji Krishnamurthy, Abhishek Singh, et al. 2021. Public health impact of delaying second dose of BNT162b2 or mRNA-1273 covid-19 vaccine: simulation agent based modeling study. *bmj* 373 (2021).

[40] Samuel Schoenholz and Ekin Dogus Cubuk. 2020. Jax md: a framework for differentiable physics. *Advances in Neural Information Processing Systems* 33 (2020), 11428–11441.

[41] Neal R Smith, James M Trauer, Manoj Gambhir, Jack S Richards, Richard J Maude, Jonathan M Keith, and Jennifer A Flegg. 2018. Agent-based models of malaria transmission: a systematic review. *Malaria journal* 17, 1 (2018), 1–16.

[42] Hyung Ju Suh, Max Simchowitz, Kaiqing Zhang, and Russ Tedrake. 2022. Do differentiable simulators give better policy gradients?. In *International Conference on Machine Learning*. PMLR, 20668–20696.

[43] Teresa Tamayo-Mendoza, Christoph Kreisbeck, Roland Lindh, and Alán Aspuru-Guzik. 2018. Automatic differentiation in quantum chemistry with applications to fully variational Hartree–Fock. *ACS central science* 4, 5 (2018), 559–566.

[44] Nina Wiedemann, Valentin Wüest, Antonio Loquercio, Matthias Müller, Dario Floreano, and Davide Scaramuzza. 2022. Training Efficient Controllers via Analytic Policy Gradient. *arXiv preprint arXiv:2209.13052* (2022).

[45] Uri Wilensky. 1999. *NetLogo*. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/