EPIPEODE: CELL FATE CHOICE ODE SIMULATION *

Faerberg, D.
QCB, Princeton
df6177@princeton.edu

Nocet-Binois, E.
CEE, Princeton
en4624@princeton.edu

Michel-Mata, S.
EEB, Princeton
sm75@princeton.edu

Marvit, H.
CS, Math, Princeton
huxley@princeton.edu

ABSTRACT

Applied mathematics has been increasingly adopted for many quantitative biological problems as the popularity of integrative sciences has risen in the past two decades. A notable approach is the use of ordinary differential equation (ODE) modeling useful especially in low-dimensional gene regulatory networks. Such modeling has proven especially insightful in more dynamical systems, where previous studies relying on terminal-collection snapshot data (e.g. fix-and-stain approaches) have provided contending findings. One crucial system that is largely studied via approaches that lack individual dynamics over time is early mammalian embryogenesis. Briefly, following fertilization mammalian embryos proliferate to ~100 cells and specify embryonic and extra-embryonic (placenta and yolk sac) precursors. While stereotyped in other model systems such as the fruit fly, this process is highly dynamic yet surprisingly robust in mammals. Recently, the dynamics of the cell fate choices in general have been modeled via Waddington landscape-like ODEs by Dr. Rand and colleagues [1]. More specifically, Drs. Siggia & Raju developed a "heteroclinic flip" model [2] and managed to capture a range of crucial features of the second cell fate choice (EPI-PE segregation) even despite biological assumptions necessary due to uncertainty in the field. However, the paper did not provide any scripts to simulate or visualize the described models. Here we present a minimal package both enabling simulations of ODE models developed in [2] and setting up an infrastructure of classes and methods for general ODE modeling and visualization. Ultimately, we present `epipeODE` - a convenient ODE modeling package for the EPI-PE segregation that is further easily extendable for most cell specification problems.

1 Background

1.1 EPI-PE segregation in early mammalian embryogenesis

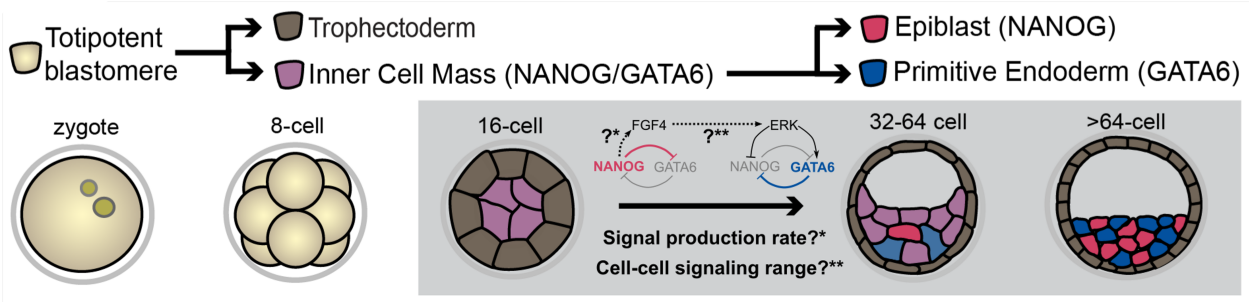


Figure 1: Graphical representation of early mammalian embryogenesis. Grey box highlights the EPI (red) - PE (blue) segregation modeled in this package. Insert circuit elaborated in Figure 2. Source: Posfai lab.

* *Corresponding Author*: **D.F. (df6177@princeton.edu)**

Following fertilization mammalian embryos undergo two sequential cell fate choices that ultimately set aside embryonic and extra-embryonic tissues (Figure 1). Briefly, first, totipotent cells differentiate into either the Trophectoderm (TE) or the Inner Cell Mass (ICM) cells. The TE cells form an outer layer and eventually form the placenta, while the ICM cells undergo a second cell fate decision specifying either the Epiblast (EPI) or the Primitive Endoderm (PE). In several well-studied invertebrate model organisms such cell fate decisions are deterministic. In contrast, early mammalian fate decisions are emergent from a dynamic, seemingly stochastic process. A prime example of this is the EPI/PE fate decision (Figure 1, grey box), where all ICM cells start with high levels of two competing transcription factors (TFs): NANOG and GATA6. Over time, one of these TFs wins over, specifying a cell's fate (Figure 2). Notably, EPI cells are known to secrete FGF4 – a signaling molecule that biases the neighbors towards the PE state. A remarkable feature of this fate decision is that EPI and PE cells emerge in reproducible ratios: 40% EPI, 60% PE. While it is clear that cell-cell communication through Fgf signaling is required for setting the ratios of the cell types, several key mechanisms of this coordination remain elusive. Specifically, the length at which cell-cell signaling occurs is disputed with claims ranging from cell-cell contact to global scales, and the rate of production of Fgf by specifying Epiblast cells is unknown.

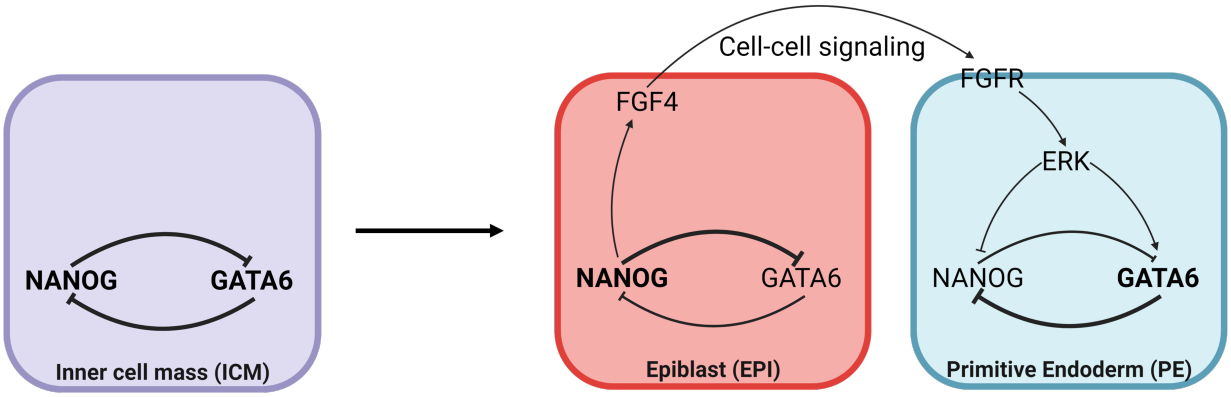


Figure 2: Antagonistic transcription factors NANOG and GATA6 control the bipotent switch between Epiblast and Primitive Endoderm fates during Inner Cell Mass specification in early mammalian embryogenesis. Made in BioRender.

1.2 Heteroclinic flip model

$$V_{hf}(x,y) = x^4 + y^4 - y^3 + 2x^2y - y^2 + f_2x + K_2y \tag{1}$$

$$f_2 = f_s\frac{1}{N}\sum_{i,x_i>0} x_i + b + f_{ex} \tag{2}$$

$$\begin{aligned} \dot{x_i} &= -\partial_{x_i}V_{hf}(x,y) + \eta_{x_i}(t) \\ \dot{y_i} &= -\partial_{y_i}V_{hf}(x,y) + \eta_{y_i}(t) \end{aligned} \tag{3}$$

The heteroclinic flip ODE model developed by Drs. Raju and Siggia [2] generates a Waddington-style landscape where initially bipotent ICM fall down a hill along the y axis eventually landing in one of the two low energy hills corresponding to the EPI and PE fates (Figure 3). The landscape is formalized over an x-y coordinate system where x defines the EPI-PE axis with -1 being 100% PE fate and +1 - 100% EPI fate, and y defines the specification axis with -1 being 100% differentiated cell (either fate) and +1 being 100% bipotent cell. The potential $V_{hf}(x,y)$ - the gravity pulling on the marbled along the slope of the landscape in the physical metaphor - is formalized in equation (1) consisting of 3 parts: a generic Rand landscape polynomial as defined in [1], the Fgf cell-cell signaling input f_2x calculated in equation (2), and a passive differentiation term K_2y that slowly and unbiasedly pulls cells away from stable bipotency. Fgf term is assumed to be spatially and temporally uniform as a global concentration for a given timepoint calculated as an average x of all cells leaning towards EPI fate ($x > 0$). Using the landscape and calculated instance Fgf term a cells displacement on the x-y plane can be computed from the potential plus some small noise (η) as shown in equation (3). The model at its basis models a progression from the ICM state at (1, 0) to either PE (-1, -1) or EPI (1, -1) fate with the landscape shifting in response to arising EPI cells to bias toward the PE fate, thus dynamically self-correcting, potentially explaining the emergent robustness of end cell fate ratios observed in real embryos.

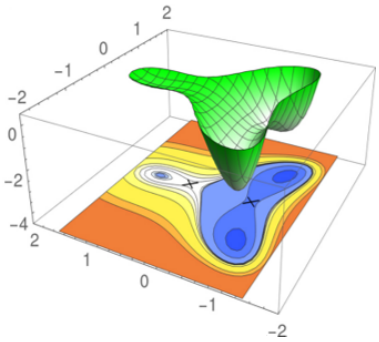


Figure 3: The heteroclinic flip model and a sample landscape [2].

This fairly simple model that abstracts away transcription factors like NANOG and GATA6, contains no spatial components of an embryo's geometry, or assumes no plasticity timing reproduces a surprisingly broad range of features observed experimentally. However, it is important to note that equation (2) is where 2 major biological assumptions are made: (1) the Fgf signal acts uniformly across the ICM without a spatial concentration component, and (2) the production of the Fgf signal is directly proportional to commitment to EPI. Currently neither of these assumptions are supported by experimental data. This package was explicitly created in the mindset that dynamic data on both (or either) key features of Fgf signaling will eventually be acquired and used to expand the ODE model.

2 Implementation

2.1 Class Structures

To allow for easy development of custom models, we designed a modular and statically typed, but modifiable environment of classes. This class structure allows for changes on each level of consideration (e.g. base model, individual cell or whole embryo levels). The classes are stored in `data_classes.py` and are as follows:

- `Point`: a spatial point class containing x, y, z coordinates in 3D space. Unused (all coordinates set to 0), but built in to allow for easy implementation of spatial components in custom models. Has a method `dist(Point)` that returns vector distance to another `Point` and a method `to_array` that converts it into a NumPy array.
- `Fate`: a point class containing x, y coordinates on the specification-fate plane described in section (1.2). Has a method `apply_noise` that adds small Gussian noise η [see equation(3)]. The noise is controlled by the optional input variable `noise_level` set to 0.05 by default.
- `Cell`: a space and fate object of a simulated cell that has 3 attributes: `loc(Point)`, `fate(Fate)` and `history(list[tuple[Point, Fate]])`. `loc` and `fate` carry the current state of the cell, while `history` saves a list of all previous states, thus keeping track of the trajectory of a given cell. Has methods for automating history tracking and a convenient string conversion (`__repr__`) for readability.
- `GeomModel`: an abstract geometrical model class controlling the simulation timestep `dt` (set to 0.001 by default) and requiring a custom model to have statically typed methods `Potential` and `Gradient` required in simulation functions.
- `Embryo`: an object of a single simulated embryo defined by a model `model(GeomModel)` and a set of cells `cells(list[Cells])`. Has a convenient string conversion method (`__repr__`) for readability.
- `History`: an object to track the state of an embryo over simulation timesteps as a list of `Embryo` variables `snapshots: list[Embryo]`. Has methods for indexing, iteration and length reading, and a convenient string conversion method (`__repr__`) for readability.

This package has 2 pre-implemented models: "dual cusp" and "heteroclinic flip" developed by Drs. Raju and Siggia [2]. These models are stored in `models.py` as class objects of their own: `DualCusp(GeomModel)` and `HeteroclinicFlip(GeomModel)`.

2.2 Simulation

The `dynamics.py` file contains the main functions for the embryo simulation, managing how cells evolve over time within a given landscape model. While we did build-in the "heteroclinic flip" and "dual cusp" models from [2], the functions and underlying classes have been designed to work with any ODE model. Note that we explicitly built in the ability to expand the existing functionalities by pre-introducing possibly desired features (e.g. our code does not use a cell's 3D coordinate part `loc` defined by a 'redundant' class `Point`, but it is included for easy introduction of spatial models). This modular design balances clarity, efficiency, and flexibility to accurately simulate biological processes while remaining adaptable for as many possible cell specification problems as we could foresee.

The central function of this module is the `update_state` function, which defines the dynamics of an individual cell. Each cell's state consists of its location (`loc`) and fate (`fate`), updated according to a potential function gradient derived from a model [see equation (3) for an example]. This behavior is abstracted through the `GeomModel` interface, allowing for different models such as `DualCusp` and `HeteroclinicFlip` to define unique potential landscapes. Following the original design in [2] avoiding local minima [see η in equation (3)] and to incorporate realistic stochasticity of biological systems, the dedicated `apply_noise` method adds small Gaussian noise directly to the cell's position on the fate-specification x-y plane. This separation of deterministic dynamics and random fluctuations makes the implementation both modular and realistic while allowing for questions such as fluctuation-to-robustness in the system.

The simulation begins with the `initialize_embryo` function, which generates cells with random initial states. These positions are centered near the origin of the geometric landscape, reflecting an assumption of initial homogeneity common in many biological systems. This controllably-random initialization introduces variability between simulations, enabling the exploration of cell trajectories relative to stochastically different conditions. Moreover, our design allows for custom initializations where an `Embryo` class can be constructed by passing a var_i = `GeomModel` and a var_j = [list of `Cell` variables] to `Embryo`(var_i, var_j). Combined, our package allows for statistical analysis of a given initial set of ICM cells producing a ratio of EPI to PE fates, relative to a desired noise level. Finally, an individual cell's dynamics towards specification - which is not temporally defined, yet measurable in this model - can be quantified and analyzed as its individual trajectory is saved in a `History` variable (a list of `Embryo` variables).

The `generate_history` function handles tracking the progress of the simulation. It saves snapshots of the embryo at specific intervals defined by the `save_interval` parameter. This feature conserves memory by limiting the amount of data stored during lengthy simulations while ensuring critical state information is preserved for analysis and visualization. Testing this functionality required careful attention to align expected and actual snapshot counts, particularly when default and custom parameters varied. For large-scale simulations, the `update_embryo_parallel` function uses Python's `ThreadPoolExecutor` to parallelize updates across cells. This significantly speeds up computations when dealing with large cell populations. However, tuning the number of threads remains a challenge, as the default setting (equal to the number of CPU cores) may not always be optimal. Adjusting the worker count dynamically or providing finer control could improve performance further.

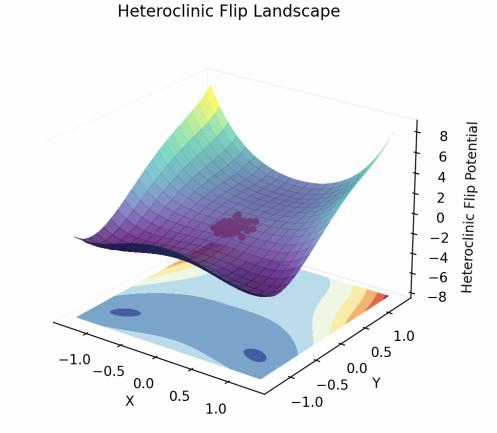


Figure 4: An example of the 3D landscape plotter output.

2.3 Visualization

Visualization is done through `matplotlib`. A user-friendly interface is created through the use of a `Visualizer` class in `visualization.py` with either of the two main methods: `plot_landscape` and `plot_trajectories`.

The plotter `plot_trajectories` creates a 3D visualization of the geometrical model being used, creating a surface plot out of the differential equation that governs how the cells specify. We generate this surface by sampling points uniformly across the X, Y grid, then finding the potential (the Z-value) at each of these points to create a 3D mesh. Bounds are dynamically calculated based on the location of the cells and the ODE provided. Then, we render this mesh (along with customized shading and such for readability), allowing users to understand the landscape that governs the cells they are studying.

The trajectories plotter creates a 2D visualization of how these cells fall into their fates over time. In order to cast the high-dimensional data of the ODE into a 2D plot, we render streamlines. (These streamlines represent the same surface as rendered above, but in 2D, and can be seen guiding the path that cells flow along). In order to render these streamlines (and do so in a readable manner across multiple different ODEs), we first calculate the gradient across all (x, y) points in our mesh. Then, to aid readability, we normalize this all, and finally render these out as streamlines. Beneath the streamlines, we also a 2D contour plot of the landscape (we found in user testing that with this added visualization component, users could better understand their landscapes). Next, we render our cells over time — we highlight the initial locations with red points, then visualize subsequent locations as they flow along the streamlines in smaller blue points.

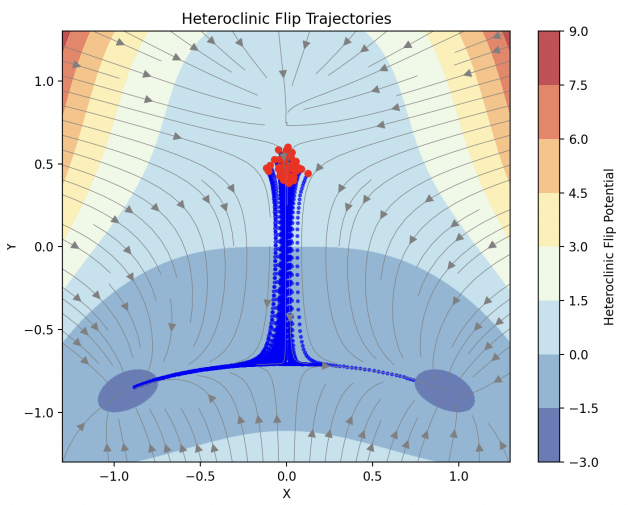


Figure 5: An example of the 2D trajectories plotter output.

We also provide functionality to save these plots as they evolve as animations, which we write to a gif using `pillow`. We do this through `matplotlib`'s `FuncAnimation`, updating the location of our cells as they move.

3 Maintenance

The next sections will provides an overview of some key aspects of the project, on the part of software design. It includes elements such as static typing, testing, packaging, and reproducibility.

3.1 Static Typing

The project employs static typing to enhance code quality and integration. The codebase leverages type annotations accessible in Python versions 3.10 and beyond such as:

- Union types expressed as `X | Y` (mostly used when including optional `None` type).
- Type hints using `list[X]`, which became available in Python 3.9.

- The use of data classes from the `dataclasses` module to declaratively define classes.

Furthermore, we integrated static validation using `mypy` to enforce type correctness. For instance, using `mypy`, we verified that functions would not receive non-iterables like `None` and try to apply operations such as `append`.

We also ensured reproducibility with the integration of pre-commit hooks to enforce coding standards and perform basic checks before any new code gets committed. This includes checks for large files, unresolved merge conflicts, and trailing whitespace, but also static type checks with `mypy`.

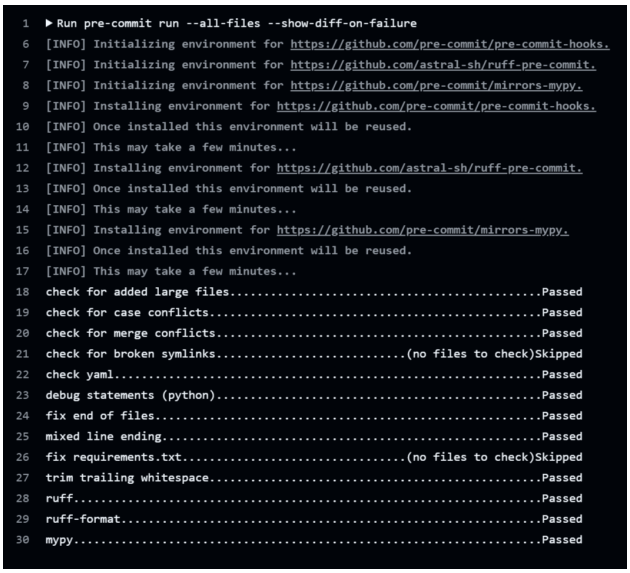


Figure 6: Pre-commits, including Ruff, that checks elements such as unused imports, incorrect formatting, and other syntactical errors, as well as Mypy which helps ensuring that variables and functions are used correctly, according to their type annotations.

3.2 Testing

On top of the static validation, we included dynamic testing, using `pytest` for writing and executing tests, which is specified as a dependency in the `pyproject.toml`. The test suite ensures that our components, such as `dynamics`, `models`, and `visualization` function correctly.

We test the core classes of our project such as Point, Fate, Cell, and Embryo which provide the foundational data structures, enabling the representation of spatial positions, dynamic states, individual cells, and collections of cells. We ensure correctness in geometric calculations (e.g., distances between points), state updates, and history tracking over time.

Higher-level functions such as `initialize_embryo`, `update_state`, and `generate_history` integrate these classes to simulate cell fate trajectories, supported by models such as `DualCusp` and `HeteroclinicFlip`. Our tests focus on the accuracy of state initialization, evolution, and time-series generation, and the geometric models are validated for their gradient, potential, and feedback mechanisms.

Lastly, we have a visualization test module that validates core graphic functionalities such as plotting dynamic landscapes, cell trajectories, or exporting simulation results as animated GIFs.

We also wrote a dedicated Nox session for running tests, ensuring that the testing environment is isolated and reproducible. The configuration in `pyproject.toml` specifies dependencies and source distribution, including tests.

The project is using `hatchling` for packaging. So far, our package has been developed mainly for Python version 3.11. This version is specified in our root directory, and is specifically targeted in the static tests and general CI. However the code should be functional with any version above 3.10.

```
▶ Run nox -s tests
nox > Running session tests
nox > Creating virtual environment (uv) using python3 in .nox/tests
nox > uv pip install '.[test]'
nox > pytest
============================= test session starts ==============================
platform linux -- Python 3.11.11, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/runner/work/embryo_ODE/embryo_ODE
configfile: pyproject.toml
collected 16 items

tests/test_basic.py ...                                                  [ 18%]
tests/test_data_classes.py .....                                        [ 50%]
tests/test_dynamics.py ...                                               [ 68%]
tests/test_models.py ..                                                  [ 81%]
tests/test_visualization.py ...                                          [100%]

============================== 16 passed in 3.85s ==============================
nox > Session tests was successful.
```

Figure 7: Executing tests on the different modules of our project.

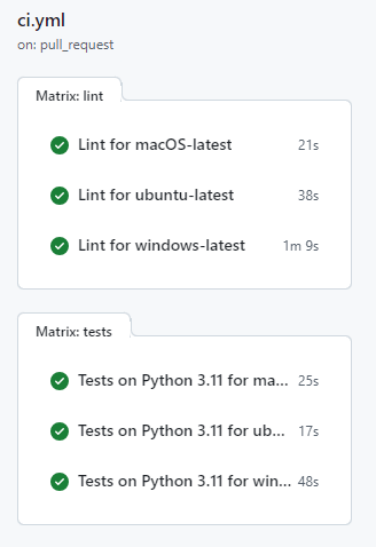


Figure 8: General CI pipeline. Our CI automates the process of linting and testing across the Ubuntu, MacOS and Windows operating systems. The workflow consists of two main jobs: lint and tests, and both make use of the `astral-sh/setup-uvv4` action to set up a virtual environment. They execute tests defined in Nox, and the pre-commit hooks.

3.3 Version Control

The embryo simulation project relied heavily on version control to maintain organization, enable collaboration, and ensure code stability throughout its iterative development. Git was chosen as the version control system, employing a feature-based branching strategy to isolate work on specific functionalities and prevent disruptions to the main codebase. For instance, the `implement_dynamics` branch focused on refining core components such as the `update_state` and `generate_history` functions. Similarly, separate branches were used for visualization improvements and bug fixes, ensuring experimental changes remained isolated until fully tested. This branching strategy minimized errors and streamlined development, allowing the team to refine features without affecting the primary code.

Commit practices were carefully managed to maintain a clear and traceable history. Each commit was descriptive, capturing incremental changes and providing context for debugging or reverting specific updates. By keeping commits small and focused, the team avoided overwhelming the project history with large, disorganized changes. One significant hurdle was resolving circular dependencies between the `data_classes` and `models` modules. These dependencies created testing and integration challenges in the project. Version control and the GitHub Actions for continuous

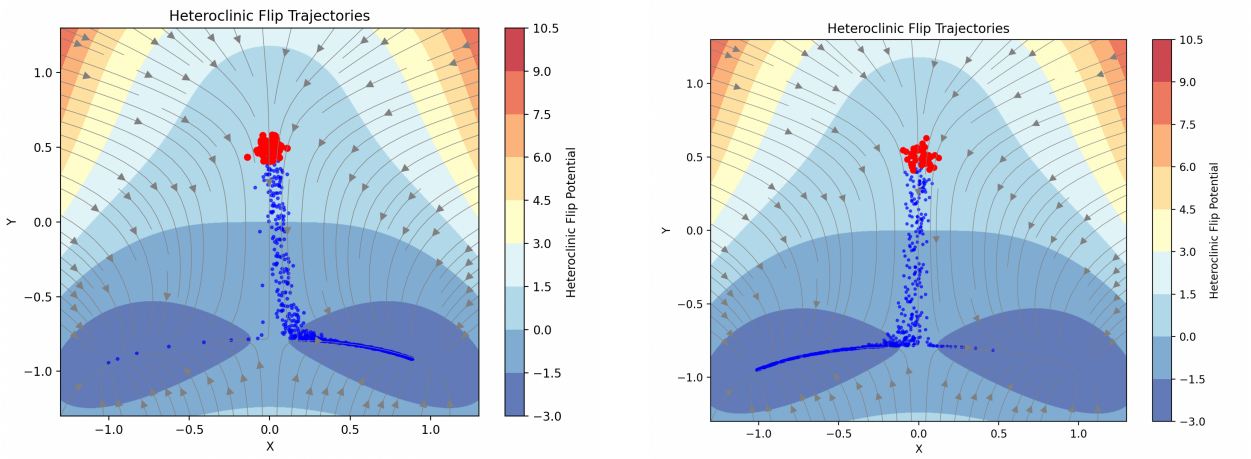
integration played a critical role here, allowing changes to be tested iteratively in dedicated branches before merging into the main codebase.

Continuous integration further enhanced the development process. Using GitHub Actions, automated testing with `pytest` ensured that all modules worked correctly. This setup identified issues such as mismatched snapshot counts during history generation and inconsistencies in visualization streamlining. Automated checks were essential for maintaining stability during feature additions and refactors. Collaboration was another key benefit of the version control strategy. Pull requests allowed team members to review code changes, fostering discussions about design choices and optimizations. For example, the decision to use `ThreadPoolExecutor` for parallel updates and the introduction of the `save_interval` parameter were refined through collaborative feedback.

Overall, the version control strategy not only structured the development process but also supported iterative refinement, balancing innovation with stability. It provided a robust framework for extending the simulation in the future while ensuring the codebase remained maintainable and scalable.

4 Results

Using this package `epipeODE`, we are able to easily and quickly run many embryogenesis simulations, which is already accelerating progress in the field (as it's in use in the Lewis-Sigler Institute for Integrative Genomics at Princeton University). With this package, we can iterate on different potential landscapes and forms of cell-seeding through simulation. This package — built for extensibility and released to the open-source community — is also poised for further use, as ongoing research in current labs are providing more models to simulate and then validate. Below are some preliminary results of simulations.

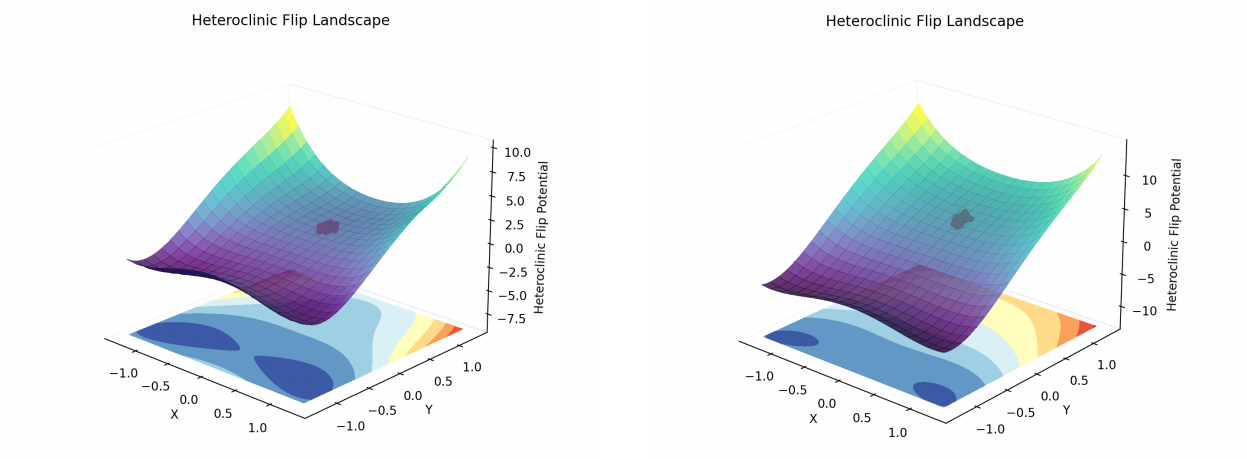


(a) Het, $K_2 = 2.2, f_s = 0.8, b = 0, f_{ex} = -0.2$

(b) Het, $K_2 = 2.2, f_s = 0.8, b = -0.13, f_{ex} = 0.2$

Figure 9: Two simulations of the Heteroclinic flip landscape, with different parameters and outcomes.

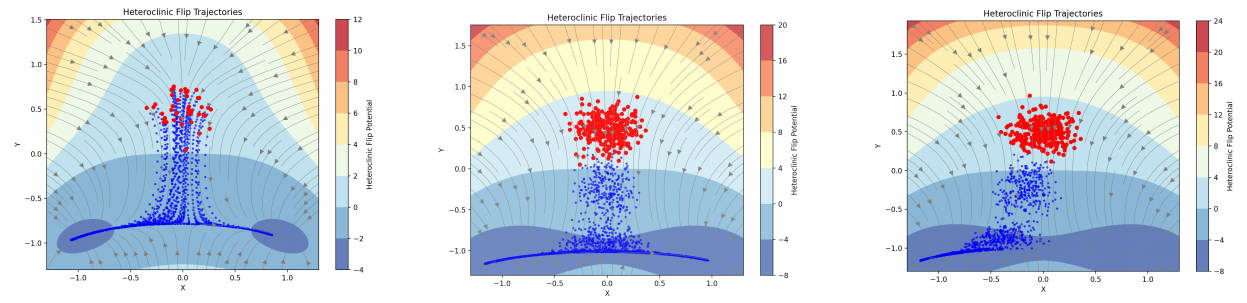
In Figure 9, we simulate a package built-in model, with two distinct sets of parameters. (Of course, in the open-source community, many distinct models would be created and passed to the package, but these simulations are run with one of the built-in example models). We see two distinct outcomes: in (a), we see the majority of our cells fall into the Fate on the right, whereas in (b) they fall into the left. (The specific paths along each timestep in history, as well as which cells fall into which fate, and the ratio between the two fates are all important data that we can extract from this simulation. For simplicity, we focus on the majority-Fate in our discussion here). The primary difference between (a) and (b) is the feedback bias term b, where in (a) $b = 0$ (the default parameter) and in (b) $b = -0.13$.



(a) Het, $K_2 = 2.2$ **(b)** Het, $K_2 = 5.4$

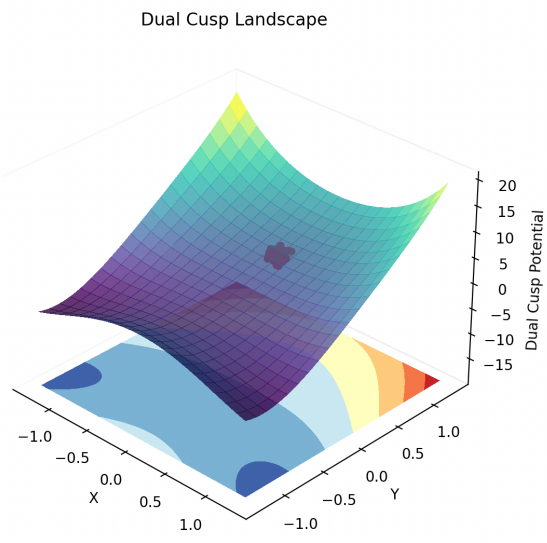
Figure 10: Two landscapes of the Heteroclinic flip model, with different parameters

In Figure 10, we can use `epipeODE` to visually tell the difference between two different parameter sets, as we render them into interactive landscapes. In this case, we are modifying the model's potential coefficiant K_2. Note the change in the contoured "shadow" that the 3D surface casts.

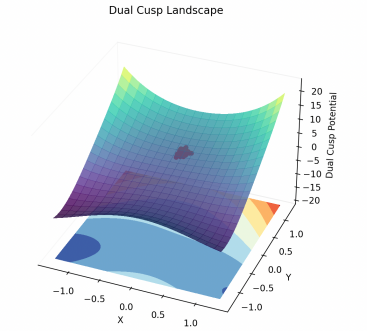


(a) Het, Spread $= 0.15$, $K_2 = 2.2$ **(b)** 300 cells, $K_2 = 5.2$ **(c)** $f_{ex} = 1.0$

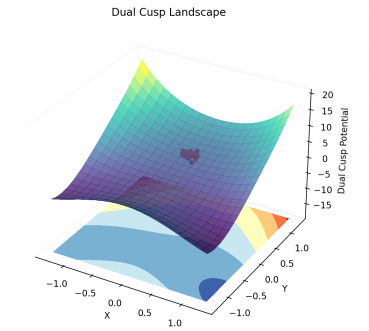
Figure 11: Simulations with both different landscape parameters and cell-seeding techniques.



(a) The base Dual Cusp landscape, $K_1 = -4.15$, $f_1 = 0.0$



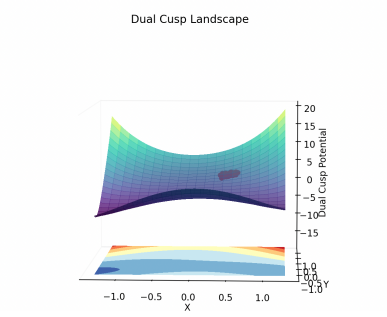
(b) $f_1 = 0.8$



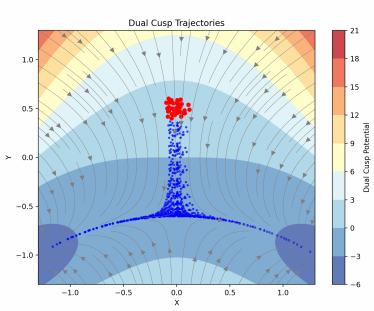
(c) $f_1 = 0.54$

Figure 12: Three instances of the Dual Cusp landscape model, with different parameters.

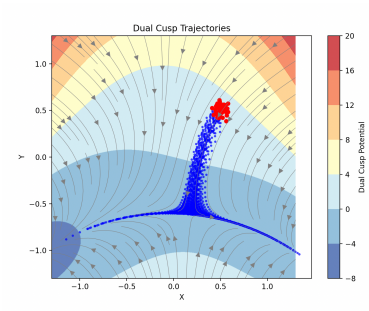
In Figure 12, we see three different landscapes generated from a new landscape model: the Dual Cusp model for embryogenesis. We see in the two parameter differences manifest in the surface "leaning" to either the left or right, as seen in (b) and (c).



(a) The landscape for (c).



(b) Base instantiation of the Dual Cusp model



(c) Modified for rightward seeding, leftward surface

Figure 13: Simulations of Dual Cusp model variants.

Figure 13 shows some intermediate steps of research on the Dual Cusp model. In (a) we see a leftward-leaning landscape, which corresponds to (c). Even while being seeded far along the x-axis, we see some cells fall into the

leftmost Fate in (c) due to this modified surface. (Please note that, in these simulations, leftward fates are set to PE while rightward fates are set to EPI). Our package enables *in silico* experiments like these.

5 Conclusion

To summarize, we have successfully established a minimal but comprehensive ecosystem to simulate and visualize cell fate choice ordinary differential equation models. As the Results section shows — and can be experienced in the test Jupyter Notebook available on GitHub — this package allows for quick and convenient testing of landscapes, parameter effects, initialized populations, etc. as all necessary functionalities are wrapped into one-line object methods. We believe this package is perfect for further development of mathematical models and allows for easy modifications necessary for more inclusive and complex modeling (e.g. introducing spatial information).

6 Contributions

D.F. edited this document, wrote the abstract, background, class structures and a parts of the Results sections of this document, and contributed to conceptualizing the package, creating dataclasses, and core design of functions / passed variables. **E.N.** wrote the static typing and testing sections of this document, and contributed to version control, setting up and maintenance of testing, manually porting in the original models from [2] to `GeomModel`, packaging and resolving merge (or any other) issues. **S.M.** wrote the simulation and version control sections, and contributed to the refactoring of the full model for readability, coding the dynamics of the model, refining visualization, and integration of the codebase. **H.M.** wrote visualization and the majority of the results sections, and contributed to creating the dataclasses, conceptualizing the architecture, creating visualization methods, setting up GitHub tools/workflows and producing panels for the Results section.

References

[1] Rand D. et al. Geometry of gene regulatory dynamics. *PNAS*, 2021.

[2] Raju A. & Siggia E. A geometrical model of cell fate specification in the mouse blastocyst. *Development*, 2024.