User Documentation

Valentin Jacot-Descombes and Peter Ashcroft

December 20, 2019

This document serves as a software documentation for a numerical platform to implement the continuous mathematical model of cell differentiation and its comparison to other types of models.

Contents

1 Introduction

The platform was developed in Python. Though it functions as a simple Python framework, and can be used using an interpreter only, the framework was developed using a Jupyter notebook and IPython. This allows the user to make change on the fly, have a lot of visualisation tools, use interactive widgets and eases the use of the platform for research purposes. To solve the numerical problems the frameworks rely on the finite-element library FEniCS.

2 Installation

The framework was developed on Mac OS 10.14.

2.1 Anaconda

> The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X.
>
> anaconda.com

To use the FEniCS library, the simplest solution is to use Anaconda's Python interpreter and install the library using Miniconda. Alternatively this guide is very useful.

To install Anaconda, you can download the open source version from anaconda.com. Miniconda can be downloaded from this repository.

2.2 Installation of FEniCS

1. Activate the root element: `source /miniconda3/bin/activate root`
2. Create an environment for the FEniCS distribution: `conda create --name fenics`
3. Activate the environment: `conda activate fenics`
4. Update conda: `conda update -n base -c defaults conda`
5. Create a link to the preferred installation channel for FEniCS: `echo "channels:\n - conda-forge\n - defaults\n" > miniconda3/envs/fenics/.condarc`
6. Install FEniCS, as well as other useful python packages: `conda install numpy scipy matplotlib jupyter fenics mshr`

7. Test if the installation succeeded by running: `python -c "import fenics"`

You should now be able to interpret the framework from this environment, which you can load with this command line: `source /miniconda3/bin/activate fenics`

2.3 Jupyter

2.3.1 From terminal

Install Jupyter for Anaconda with: `conda install -c anaconda jupyter`. The notebooks can then be launched with: `jupyter-notebook source.ipynb`. You may need to activate the FEniCS environment first with the command `source /miniconda3/bin/activate fenics`.

2.3.2 From Anaconda

From Anaconda one can install Jupyter and then launch a Jupyter Notebook. From this, we can check if the FEniCS library is available: `from python import *`. If an error occurs, try the following:

```
!conda config -add channels conda-forge
!conda install fenics -y
```

2.4 PyCharm

Once FEniCS is installed from Anaconda, you only need to change the interpreter in PyCharm to use it. The process is similar for different IDEs (see Fig. 1).

3 Continuous Model

The continuous model implements different versions of the advection diffusion equation, as used in different studies. See: Mathematical modeling with single-cell sequencing data, (Cho & Rockne, 2019) [1].

Let $\vec{\theta}$ describe the location in phenotypic space, then $u(\vec{\theta}, t)$ is the density of cells of that phenotype at time t. This cell density is modified by three terms: i) cell differentiation; ii) population growth; and iii) noise. The more general functions are in the form of:

$$\partial_t u(\vec{\theta}, t) = \underbrace{-\vec{\nabla} \cdot \left[\vec{V}(\vec{\theta}, t) u(\vec{\theta}, t)\right]}_{\text{cell differentiation}} + \underbrace{R(\vec{\theta}, u(\vec{\theta}, t))}_{\text{population growth}} + \underbrace{\vec{\nabla} \cdot \left[D(\vec{\theta}) \vec{\nabla} u(\vec{\theta}, t)\right]}_{\text{noise}}, \tag{1}$$

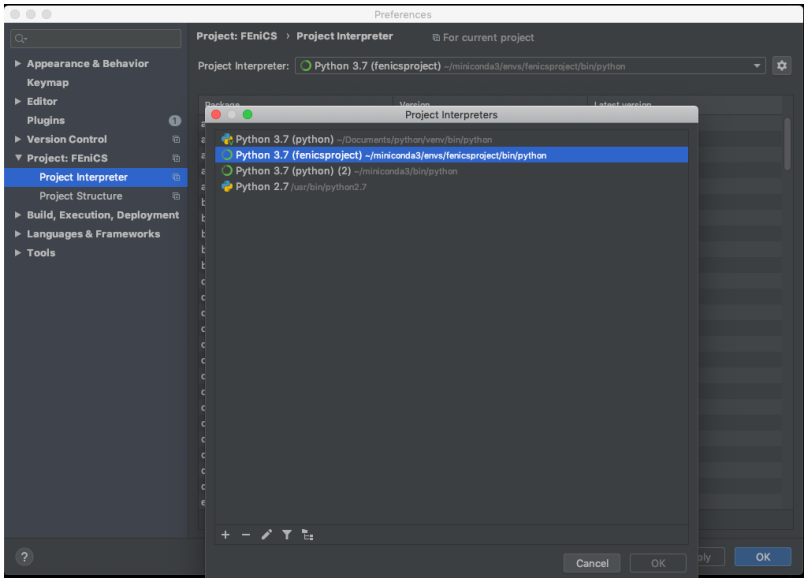


Fig. 1: Setting the interpreter of PyCharm to use the FEniCS library.

with zero Dirichlet boundary conditions. The cell differentiation term is is split in two parts: the first is the homeostatic potential, while the second is active differentiation. We therefore have

$$\vec{V}(\vec{\theta},t) = -\nu\vec{\nabla}\left[-\log\big(u^*(\vec{\theta})\big)\right] + \vec{c}(\vec{\theta})\left[2\big(1-a(\vec{\theta})\big)r(\vec{\theta})\right]s_v(t), \tag{2}$$

where $u^*(\vec{\theta})$ is the homeostatic cell density, $\vec{c}(\vec{\theta})$ is the direction and magnitude of differentiation (what fraction of differentiating cells move in a given direction), $a(\vec{\theta})$ is the self-renewal probability, $r(\vec{\theta})$ is the rate of cell division, and $s_v(t)$ is a signalling mechanism.

The different Jupyter files are divided in cells, which are indexed by integers representing the following subsections. The implementation of this chapter is the file `Jupyter/Framework_continuous_model.ipynb`.

3.1 Imports

These cells hold the necessary Python imports for the framework.

3.2 Constants

These cells define global constants for the framework. The grid size N is the dimension of the grid used by FEniCS to solve the PDE (Default $N = 120$). The grid size M is the

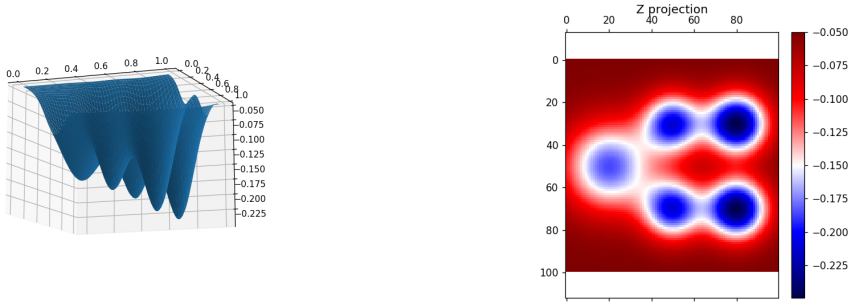
dimension of the grid used to define the matrix coefficients (Default $M = 100$).

3.3 Functions

Theses cells define global functions used to generate the artificial landscape, that corresponds to the homeostatic cell density.

3.4 Definition of a Landscape

These cells create the artificial landscape. The default landscape is made of five Gaussian "wells". These are located as described in Fig. 2.



(a) 3-dimensional landscape.

(b) 2-dimensional projection of landscape.

Fig. 2: Landscape.

In some of the more advanced applications, the wells with a lower y-position are less deep, which breaks the symmetry and allows for a better understanding of the underlying dynamics, but puts a heavy load on the performance of the solver. The grids X and Y are also defined here.

3.5 Function Space and Mesh

In these cells the function space $\hat{V}$ and the mesh for the solver are defined.

3.6 Coefficients of the PDE

Most coefficients are extracted from a simple 2-dimensional function written in Python. It is applied to the grid, which yields a matrix. This matrix is then given as an argument to the instance of an object `functionFactory` (see Table 1 for the ULM of the class). The usage

is very simple. The object is instantiated as such: `factory = functionFactory(Vhat, N, M)`, where `Vhat` is the function space, `N` and `M` are the respective grid sizes. The instance is global for the whole notebook, and only needs to be instantiated once (as long as the problem size does not change). The matrix is then passed to the instance as such: `newFunction = factory(matrix)`. The function is returned and can be used in the solver.

functionFacotry
M N $\hat{V}$
interpMat2(n,m,mat) plot(mat_ , title)

Table 1: ULM of the functionFactory class, implementing the factory pattern.

The coefficients that are non-scalar are defined for each dimension. This is the case of the direction of the differentiation $\vec{g}$, which is defined and integrated by it x and y components, `v2_x` and `v2_y`.

3.7 PDE solver

The solver in this section takes an initial distribution as defined previously and the different coefficients as functions. If the coefficient is dependent on the value of the unknown, as is the case of the reaction term, it can be updated through direct assignment at each loop iteration. The solver takes a variational approach. The equation Eq. (1) is multiplied by a test function and then integrated by parts, in order to reduce the order of differentiation. In the case of a problem with zero boundary condition this process is fairly simple. This yield a system of equations that is solved by the FEniCS solver. The choice of the time step and grid size are a little more complex, and the time-dependent part of the equation has to be estimated by a Euler iterations. In the framework, there is the possibility to save the solution as `.vtk` files, which can the be visualised with the open-source solver Paraview. The corresponding code lines are commented out. These files can be surprisingly large, especially if the number of steps is large.

3.8 Selection Tool

A simple selection tool is used to define an area. The coordinates of the area can be used to extract data from the solution, e.g. integrate the solution over this area.

4 Continuous to Discrete

The calculation of the different coefficients for the discrete approximation of the differentiation problem is done by defining and applying bins on the solution of the continuous problem. These can be easily made by using the `Bin` class described in the next section. Many examples are provided in the file `Framework_Continuous_to_discrete.ipynb`. The coefficients extracted from the bins can then be used in the linear ODEs, and thus be compared to the results of the continuous method.

4.1 Bin class

`class bin(x,y,rad,n,state)`

This class builds an object that represents a square mesh around the coordinate `(x,y)` (the center of the mesh) and of side length $2r$ and consisting of $2n$ triangles. This helps to calculate the needed parameters for the next steps.

The inner state of the object `Bin` can be set to `'c'` and the integration will be performed on a circle mesh based on the coordinate `(x,y)` (the center of the mesh) and of radius r and consisting of $2n$ triangles. Remark: the circle shape is less precise, but with a higher number of degree of freedom, the correctness stands.

The ULM diagram of this class are depicted in the Fig. 3. Example of the instantiation of the class: `Bin1 = Bin(0.2, 0.5, 0.1, 100, 'S')`.

The methods of this class are the following:

1. `integrate(u)`: integrates a function `u` over the mesh covering the instance of the bin, and returns the value of the integral. Example: `u1 = Bin1.integrate(u)`.
2. `normedIntegral(arg, u)`: returns the value of the integral of arg over the mesh divided by the value of the integral of the argument `u`. Useful to calculate rates.
3. `setState(state)`: sets the state of the instance and hence the form of the mesh to either a square if state is equal to `'s'`, or a circle if state is equal to `'c'`. The state pattern is used to allow the user to change the state on the fly and limit the code redundancy. By default the state is set to square.
4. `maxValue(u)` and `minValue(u)`: returns the maximum and minimum values respectively of the function `u` in the bin.
5. `outflow(u)`: returns the value of the outflow of the function `u` on the boundary of the bin's mesh.

6. `upperHalfOutflow(u)` and `lowerHalfOutflow(u)`: returns the value of the outflow of the function `u` on the boundary of the bin's upper and lower halves of the mesh respectively.

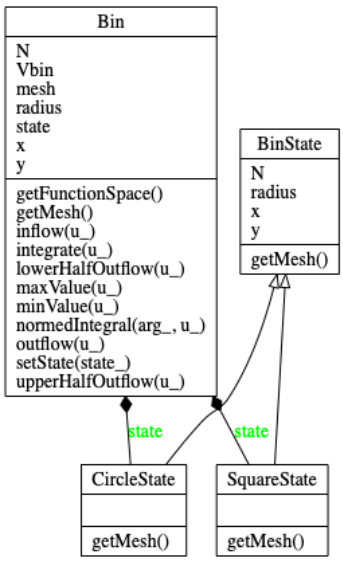


Fig. 3: ULM of the Bin class, with the state Pattern implemented.

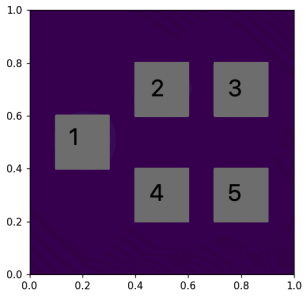


Fig. 4: Default position of the bins.

4.2 Further test and implementations

In the file `Jupyter/Archive/Framework_Continuous_to_discrete_noadvec.ipynb`, there are some further test and attempts to understand the problem, including the isolation of the bins in order to extract coefficients.

5 Discrete to Continuous

The implementation of this chapter is the file `Jupyter/Framework_Discrete_to_continuous.ipynb`.

5.1 Discrete model

Let u_i be the number of cells of type i, where $i = 1$ are the stem cells and $i = n_k$ for $k \in \{1, 2, \ldots, K\}$ are the terminally differentiated states. We then have parameters r_i (birth rate), d_i (death rate), g_i (differentiation rate), and c_{ij} (branching probability). Our discrete equations are then given by

$$\dot{u}_i = \underbrace{r_i u_i}_{\text{reproduction}} + \underbrace{\sum_j c_{ji} g_j u_j}_{\text{diff. in}} - \underbrace{g_i u_i}_{\text{diff. out}} - \underbrace{d_i u_i}_{\text{death}}. \tag{3}$$

In terms of boundary conditions for the parameters we have: $c_{j1} = 0 \, \forall j$ (no differentiation into stem cells); $g_{n_k} = 0$ (no differentiation out of terminal states); and $r_{n_k} = 0$ (no proliferation in terminal states). Finally, for homeostasis to occur, we require $r_1 - g_1 - d_1 = 0$.

5.2 Stationary solution

The stationary solution for this problem satisfies

$$0 = (r - d)u^* + \nu \nabla^2 u^* - \vec{\nabla} \cdot (\vec{g} u^*). \tag{4}$$

If u^* is the known stationary solution (i.e. observed distribution from data), and we know the birth and death rates $r(x, y)$ and $d(x, y)$, then we want to solve Eq. (4) to find $\vec{g}(x, y)$. Writing $\vec{F} = \vec{g} u$, we want to solve the equation

$$\vec{\nabla} \cdot \vec{F} = (r - d)u^* + \nu \nabla^2 u^*, \tag{5}$$

for $\vec{F}$ and then the differentiation field is given $\vec{g} = \vec{F}/u^*$. If $\vec{F}$ is curl-free, then we can write it as a scalar potential $\vec{F} = \vec{\nabla}\phi$, and then solve Poisson's equation

$$\nabla^2 \phi = (r - d)u^* + \nu \nabla^2 u^*. \tag{6}$$

Note that the field $\vec{F}$ can only be determined up to a constant, and we need some additional information from somewhere to completely specify g.

5.3 Implementation

From the steady state solution we compute $\vec{g}$ from Eq. (6) and $\vec{g} = \vec{F}/u^*$. This vector field is shown in Fig. 5. We can then use their vector components in the general equation of the framework to get back to the continuous solution.

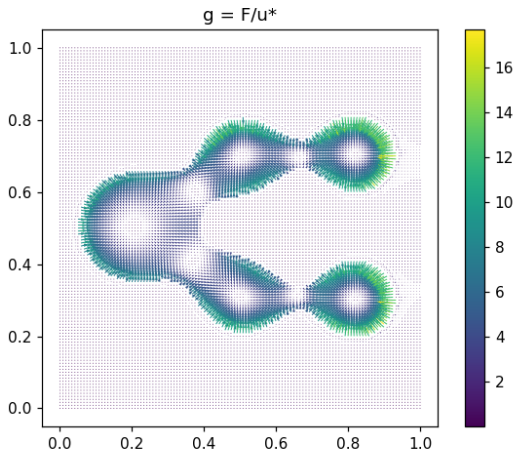


Fig. 5: $\vec{g} = \vec{F}/u^*$

6 Framework with biological data

In the folder `Jupyter/WithData` there are two examples biological data integrated into the framework. The data come from Paul *et al.* [2] and Nestorowa *et al.* [3]. The method of dimension reduction is inspired by Cho & Rockne [1], and the Matlab code that they published. The following steps are executed on the Matlab data:

1. Diffusion map: it is a method of dimensionality reduction;
2. Projection: projects the data onto the first 10 axes;
3. Normalisation: normalisation of the data;
4. Homeostasis: computation of homeostasis cell distribution;

The homeostasis distribution is finally used in the continuous model as the homeostatic cell density (previously simulated by the landscape).

7 Acknowledgement and links

The framework was developed in the context of the ETH Zürich Career Seed Grant SEED26 19-1. All the files are available at https://github.com/valentinjacot/differentiation_framework.

References

[1] Cho H, Rockne R. *Mathematical modeling with single-cell sequencing data.* bioRxiv (2019). doi:10.1101/710640.

[2] Paul F, Arkin Y, Giladi A, Jaitin DA, Kenigsberg E, Keren-Shaul H, Winter D, Lara-Astiaso D, Gury M, Weiner A, David E, Cohen N, Lauridsen FKB, Haas S, Andreas S, Mildner A, Ginhoux F, Jung S, Trumpp A, Porse BT, Tanay A, Amit I. *Transcriptional heterogeneity and lineage commitment in myeloid progenitors.* Cell 163, 1663–1677 (2015). doi:10.1016/j.cell.2015.11.013.

[3] Nestorowa S, Hamey FK, Pijuan Sala B, Diamanti E, Shepherd M, Laurenti E, Wilson NK, Kent DG, Göttgens B. *A single-cell resolution map of mouse hematopoietic stem and progenitor cell differentiation.* Blood 128, e20–31 (2016). doi:10.1182/blood-2016-05-716480.