User Documentation

Valentin Jacot-Descombes and Peter Ashcroft

December 18, 2019

This document serves as a software documentation for a numerical platform to implement the continuous mathematical model of cell differentiation.

Contents

1 Introduction

The platform was developed in Python. Though it functions in a stand alone way, 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 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." (source: `https://www.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 was very useful:

1. Install Anaconda:
   You can download the open source version from: `www.anaconda.com`
2. Install Miniconda:
   Download miniconda using this link or the appropriate one from: `https://repo.continuum.io/miniconda`
   You may need to install is using the terminal command: `./bash/sh/Downloads/Miniconda3-latest-Mac`

2.2 Installation of FEniCS

1. Activate the root element: `source /miniconda3/bin/activate root`
2. Create an environment called fenics: `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 fenics scipy matplotlib jupyter 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`

2.3.2 From Anaconda

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

```
!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 IDE.

3 Continuous Model

The continuous model implement different versions of the advection diffusion equation, from different papers. See: Mathematical modeling with single-cell sequencing data, (Cho & Rockne) [**?**]. 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:

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

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.

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 simulate the homeostatic cell density.

3.4 Definition of a Landscape

These cells create an artificial landscape that simulate a homeostatic cell density.
The default landscape is made of 5 2-dimensional gaussian "wells". These are located as in figure 1 and 2.

In some of the more advanced frameworks, 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 $\hat{\mathrm{V}}$ and the mesh for the solver are defined.

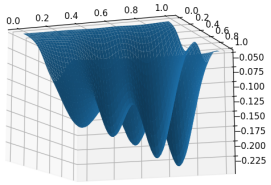

Figure 1: 3-dimensional landscape

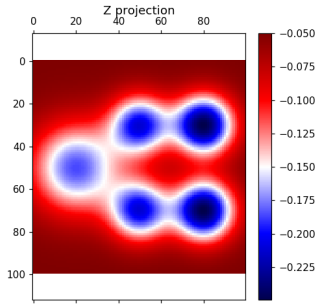

Figure 2: 2-dimensional projection

3.6 Coefficients of the PDE

Most coefficients are extracted from a simple python 2-dimensional function. 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 figure 3 for the ULM of the class). The usage is very simple. The object is instantiated as such: `factory = functionFactory(V̂, N, M)`, where $\hat{V}$ is the function space, N and M are the respective grid sizes. The matrix is then passed to the instance as such: `newFunction = factory(matrix)`. The function is returned and can be used in the solver.

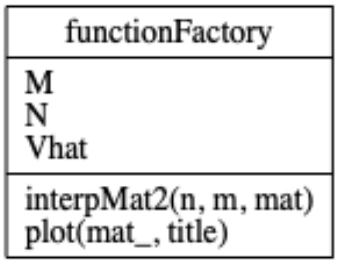

Figure 3: ULM of the functionFactory class, implementing the factory pattern

The coefficients that are non-scalar are defined for each dimension. This is the case of the direction of the differentiation, which is defined and integrated by it x and y components.

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 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 library. 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, but this is fairly basic math.

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 ordinary differential equations, 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) and of side length 2r and 2n triangles. This helps to calculate the needed parameters for the next steps.
The state of the object Bin.shape can be set to 'c' and the integration will be performed on a circle mesh based on the coordinate (x,y) and of radius r and consisting of 2*n triangles. Remark: the circle shape is less precise, but with a higher number of degree of freedom, the correctness stands
The method of this class are depicted in the figure 4
The methods of this class:

1. `integrate(u)`: integrates a function u over the mesh covering the instance of the bin, and returns the value of the integral.

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.

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.

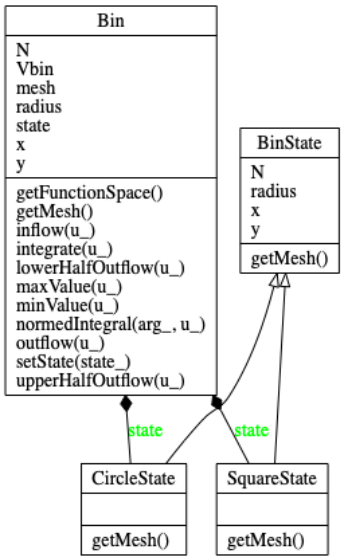

Figure 4: ULM of the Bin class, with the state Pattern implemented

5 Discrete to Continuous

6 Framework with biological data

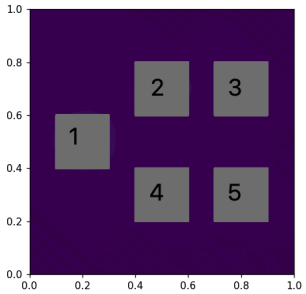

Figure 5: Standard postition of the bins as it stands in the local