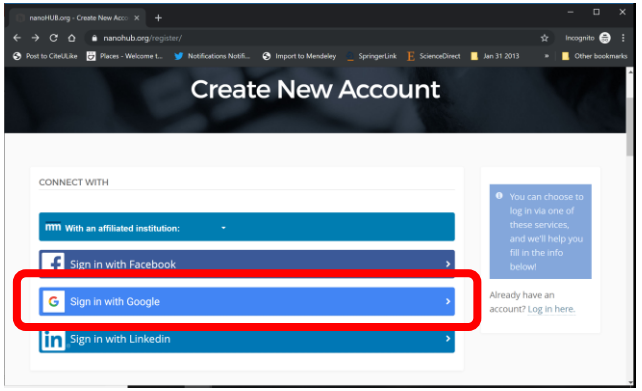
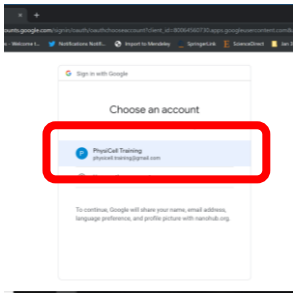
nanoHUB Account

- This talk's online PhysiCell models are cloud-hosted on nanoHUB.org.
- nanoHUB is **free**, but it requires a one-time registration.
- **Steps**:
  1. Visit https://nanohub.org/register
  2. Choose "Sign in with Google"
  3. Choose a Google account
  4. Click "No" (so it doesn't try to associate with some other nanoHIB account)
  5. Finish filling in details, and you're done!
  6. Use your google account to sign in in the future.

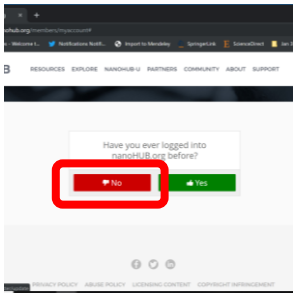
2



3



4



Introduction to agent-based modeling in biology: Part 1 (Python-based)

Lecture materials and code are available at:

[link]

Paul Macklin, Ph.D

Intelligent Systems Engineering
Indiana University

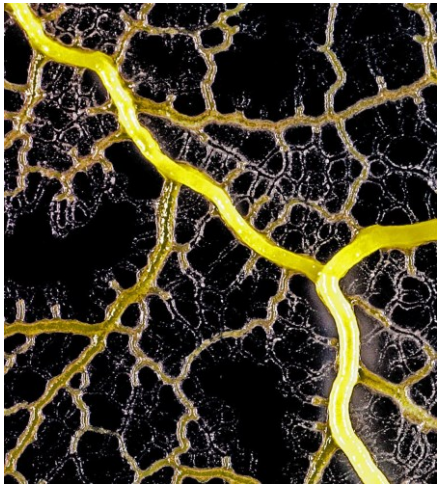
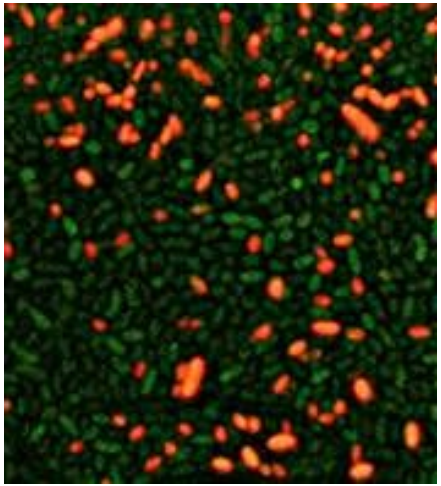
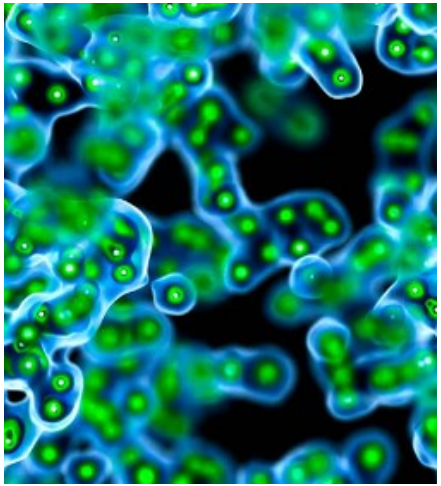
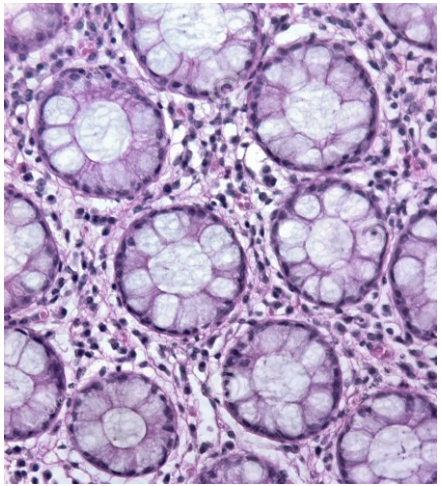
February 16, 2021

Simple single-cell behaviors …

- Growth
- Division
- Death
- Adhesion
- Mechanics
- Motility
- Secretion
- Uptake
- Sampling
- Predation
- Differentiation
- …

Give rise to complex systems

- ***Multicellular systems***—composed of multiple cells of multiple types—can exhibit remarkable diversity, with complex emergent behaviors.

How do these systems self-organize and sustain themselves?

How do we understand these multiscale systems?

Interconnected systems and processes:

- Single-cell behaviors
- Cell-cell communication
- Physics-imposed constraints (e.g., diffusion)
- Systems of systems (e.g., immune system)

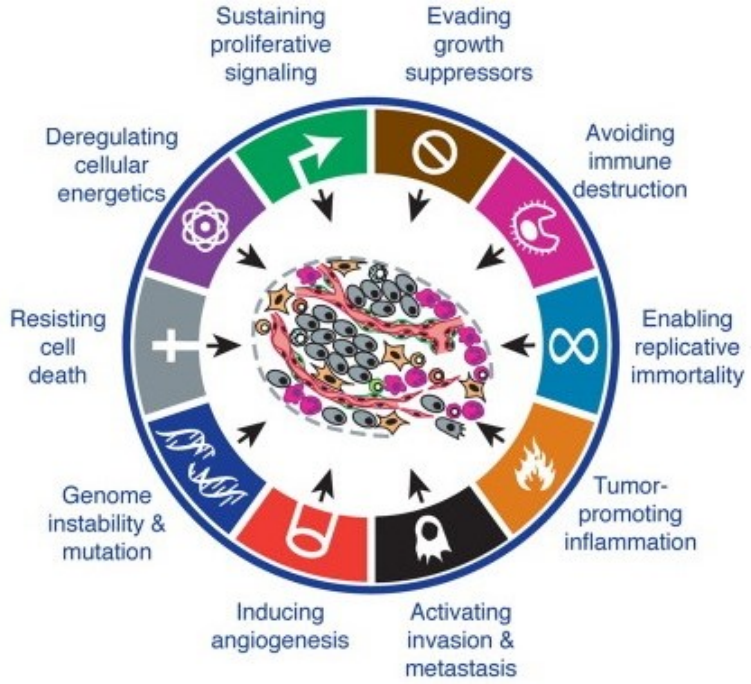
In diseases, these systems become dysregulated.

Treatments target *parts* of these systems.

Health is a **complex system:**
changing one part can have **surprising effects**!

Modeling can help **understand** this system.
This is **multicellular systems biology.**

If we can **control** these systems, we've arrived at **multicellular systems engineering**.



Source: Hanahan & Weinberg (2011)
DOI: 10.1016/j.cell.2011.02.013

Scientists use [models*] to detangle complex systems.

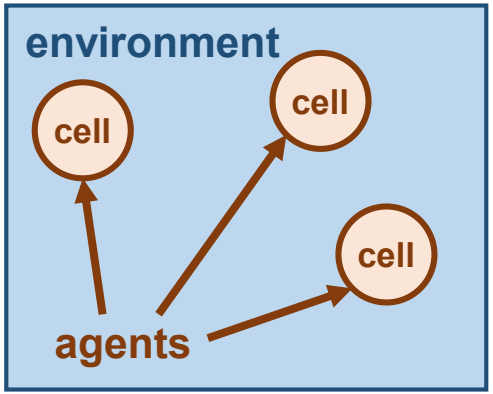
* animal, *in vitro*, engineered, mathematical, …

Key parts of a multicellular virtual laboratory

- **Model multiple diffusing chemical factors**
  - Growth substrates and metabolites
  - Signaling factors
  - Drugs
- **Model many cells in these chemical environments**
  - Environment-dependent behavior (including molecular-scale "logic")
  - Mechanical interactions
  - Heterogeneity:
    - individual states
    - individual parameter values
    - individual model rules
- **Run many copies of the model in high throughput**
  - Discover the rules that best match observations.
  - Identify and exploit weaknesses that can restore control

What is an agent-based model?

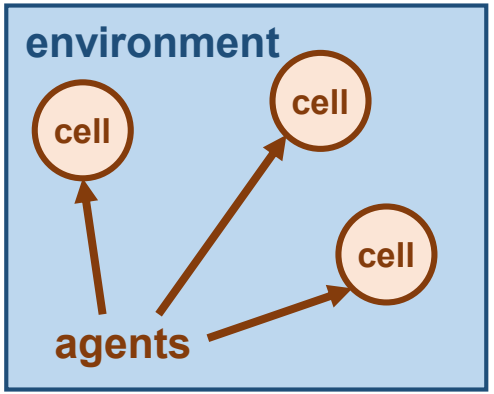
- Each cell is modeled as a separate software object (an **agent**) with:
  - **member data:** internal state variables
    - Position, Size, Cycle State, molecular variables, ….
  - **methods:** cellular processes
    - Cycling, Death, Motility, Growth, Adhesion, …



- Virtual cells move a virtual **(micro)environment**
  - Usually liquid (e.g., water or interstitial fluid)
  - Chemical movement (oxygen, glucose, signaling factors)
    - Typically diffusion: solve partial differential equations (PDEs)
    - May also require advection for environments with flow
  - May include mechanical components like extracellular matrix (ECM)
    - Finite element methods or related methods

What's the connection to biology and physics?

- The cell **agents encode our biological knowledge and hypotheses:**
  - Cell variables (member data) are selected to record important biological quantities
    - Volume, cell cycle state, energy, …
  - Cell rules (methods) encode biological hypotheses
    - Increase motility in low oxygen, down-regulate cycling under compression, …
  - Cell rules are often written at mathematical models
    - Potential functions for mechanics, systems of ODEs for metabolism, …

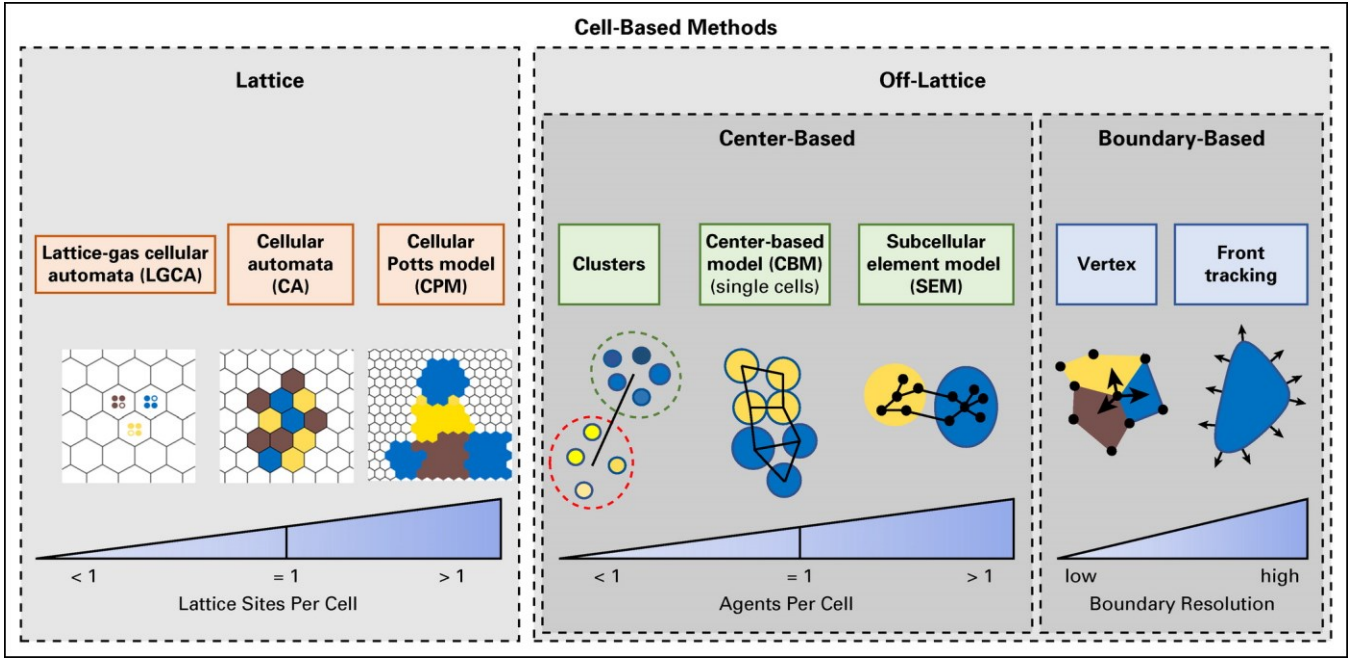


- The **microenvironment encodes physical constraints:**
  - *Chemical transport:* diffusion and advection equations (PDEs)
  - *Tissue mechanics:* viscoelastic, plastoelastic or other solid mechanics

- Most agent-based models combine **discrete** cell agents and **continuum** microenvironment processes. This is a **hybrid continuum-discrete approach.**

Types of cell-based models

- **lattice-bound**
  - resolution:
    - ♦ < 1 site / cell:
      - » lattice gas
    - ♦ 1 site / cell
      - » cellular automaton
    - ♦ many sites / cell
      - » cellular Potts
- **off-lattice**
  - **center-based**
  - boundary-based

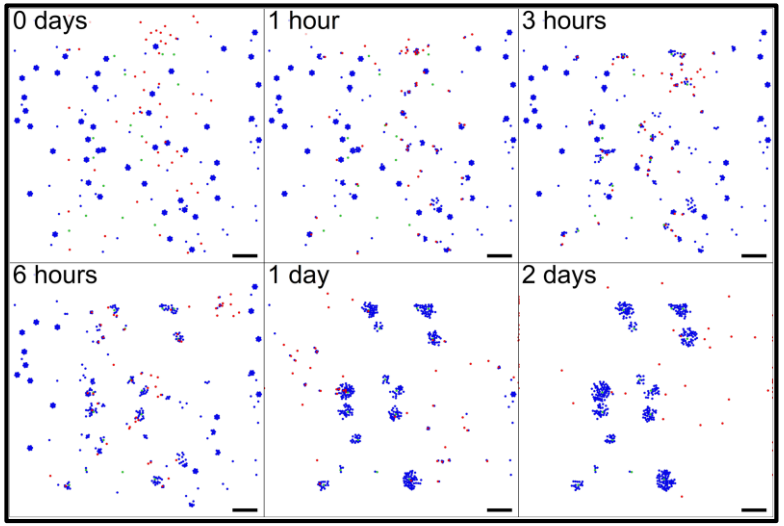


J. Metzcar, Y. Wang, R. Heiland, and P. Macklin. A review of cell-based computational modeling in cancer biology. *JCO Clinical Cancer Informatics* 3:1-13, 2019 (invited review). DOI: 10.1200/CCI.18.00069.

Example: biological cargo delivery system

- **Chemical environment:**
  - two diffusing chemical signals

- **Cell types and rules:**
  - **directors (green):**
    - secrete director signal to attract workers
  - **cargo (blue):**
    - **undocked**: secrete cargo signal to attract workers
    - **docked**: turn off signal
  - **workers (red):**
    - **undocked**: seek cargo via chemotaxis
    - **docked:** seek directors via chemotaxis, release cargo in high signal areas



Try this model yourself!

https://nanohub.org/tools/pc4biorobots

Typical program flow

- Read parameters
- Set up microenvironment
  - Create meshes, initialize chemical substrates, diffusion solvers, etc.
- Set up cell agents
  - Define all cell types
  - Instantiate cells
- For each time:
  - Update microenvironment
    - Solve reaction-diffusion equations (as needed)
    - Solve tissue mechanics (as needed)
  - Update each cell's state
    - Sample environment
    - Run signaling model (as needed)
    - Update behavioral parameters based on signaling model and sampled environment
    - Run cell process models (growth, cycling, death, …)
  - Calculate cell velocities
  - Update cell positions
  - Advance time

Let’s build an agent-based model in Python

Classes

- A **class** is a template for creating a software object, including:
  - member data
  - initial values for its state (member variables)
  - member functions (or methods)

- A class will generally have one or more **constructors** that are called when the class is instantiated.
  - Set default values for member data

A class declaration (Python)

```
class Agent:
    def __init__( self ): # default constructor
        self.hidden_variable = False; # no such thing as private in Python
        self.position = [0,0];
        self.hunger = 0.0;
        self.ID = 0;
        return;

    def move_me( self , dx, dy ): # member function
        self.position[0] += dx ;
        self.position[1] += dy;
        return;

    def display( self ):
        print( str(self.ID) + ' at ' + str(self.position) + ' has hunger level '
            + str(self.hunger) )

Bob = Agent();
Bob.ID = 1;
Bob.move_me( 1,-1);
Bob.display();
```

Go to the Lecture notebook (Section 1) to execute this code.

Loading key libraries

```
# Let's load libraries

import numpy as np
import matplotlib.pyplot as plt

# set matplotlib to do plots inline
%matplotlib inline
```

Go to the Lecture notebook (Section 2) to execute this code.

Creating an environment

- We'll create an environment with one chemical substrate *c* that diffuses and decays in the environment.

- We'll need a mesh on a domain $[a, b] \times [c, d]$
  - X coordinates: *m* nodes, $x_i = a + i\Delta x$
  - Y coordinates: *n* nodes, $y_j = c + j\Delta y$
  - An array to store the chemical substrate: *m* × *n* nodes

  - We'll store the diffusion coefficient (*D*) and decay rate (λ)

- By convention, $c_{i,j} = c(x_i, y_j)$ stored at the present simulation time.

Environment class (v1)

```
# Declare environment class
class Environment:
    def __init__( self , shape=[-100,-100,100,100],  m=2 , n=2 ):
        self.a = shape[0]
        self.b = shape[2]
        self.c = shape[1]
        self.d = shape[3]
        self.m = m;
        self.n = n;
        self.dx = (self.b-self.a)/(m-1);
        self.dy = (self.d-self.c)/(n-1);
        self.X = np.linspace( self.a, self.b , m )
        self.Y = np.linspace( self.c, self.d, n )
        self.C = np.zeros((m,n));
        return;
    def setup( self , D=1000, decay=0.1 , initial=1.0, boundary=1.0 ):
        self.D = D;
        self.decay = decay; # In Python you can evidently declare more class elements in methods
        # set boundary values
        self.C = boundary * np.ones((self.m,self.n))
        # set interior values to initial value
        for j in range(1,self.n-1):
            for i in range(1,self.m-1):
                self.C[i,j] = initial ;
        print( 'Length scale: ' + str( np.sqrt(self.D/self.decay) ) )
        return;
```

Go to the Lecture notebook (Section 3) to execute this code.

Testing …

```
E = Environment( [-100,0,200,400] , 31,41 )

E.setup( 1000, 0.1 , 0.5 , 1 )

plt.contourf( E.X, E.Y, np.transpose( E.C ) )
plt.axis('image')
plt.colorbar()
```

Go to the Lecture notebook (Section 3) to execute this code.

Adding diffusion and plotting

- Now, we want to solve the diffusion-decay equation:

$$\frac{\partial c}{\partial t} = D\nabla^2 c - \lambda c$$

- We approximate the partial derivatives with finite differences:

$$\frac{c_{i,j}^{n+1} - c_{i,j}^{n}}{\Delta t} = D\left(\frac{c_{i+1,j}^{n} - 2c_{i,j}^{n} + c_{i-1,j}^{n}}{\Delta x^2} + \frac{c_{i,j+1}^{n} - 2c_{i,j}^{n} + c_{i,j-1}^{n}}{\Delta y^2}\right) - \lambda c_{i,j}^{n}$$

- Then, algebraically solve for $c_{i,j}^{n+1}$

Environment class (v2)

```
# Declare environment class
class Environment:
    # __init__ and setup as before

    def update( self , dt=0.001 ):
        # define constants for simplicity
        A = self.D * dt / ( self.dx**2 )
        B = self.D * dt / ( self.dy**2 )
        C = self.decay * dt
        # copy the prior solution.
        old = self.C.copy(); # make sure this is a real copy and not a reference.
        # finite differences for decay-diffusion
        for i in range(1,self.m-1):
            for j in range(1,self.n-1):
                self.C[i,j] = (1-2*A-2*B-C)*old[i,j] + A*(old[i-1,j]+old[i+1,j])
                    + B*( old[i,j-1]+old[i,j+1] );
        return;

    def plot( self ):
        plt.clf()
        plt.contourf( self.X, self.Y, np.transpose( self.C ) )
        plt.axis('image')
        plt.colorbar()
        plt.xlabel('x')
        plt.ylabel('y')
```

Go to the Lecture notebook (Section 4) to execute this code.

A word on stability

- For an explicit finite difference method, the time step size must be chosen to maintain numerical stability.
- For *n*-dimensional diffusion, the criterion is

$$\Delta t \leq \frac{\Delta x^2}{2nD}$$

(Theory: Information cannot spread across your mesh faster than the diffusion process.)

Testing …

```
# declare an environment
E = Environment( [-250,-250,250,250] , 26,26 )
# set up
E.setup( 10000, 1 , 0.5 , 1 )

dt = E.dx**2 / (2 * 2 * E.D )

print(dt)

# main test loop
for n in range(100):
    if( n % 20  == 0 ):
        E.plot()
        plt.show()
    E.update( dt )

E.plot()
```

Go to the Lecture notebook (Section 4) to execute this code.

Creating a Cell class (v1)

- Let's iteratively create and refine a Cell class.

- The first version will add a constructor and basic cell-cell mechanics
  - Each cell will interact with nearby cells
  - We'll use a spring-like force to model the combined effects of adhesion and repulsion
  - We'll assume dissipative (drag-like) forces lead to fast equilibration, allowing us to solve for the cell velocity.

Cell velocities: math (1)

- Suppose cell *i* and cell *j* are connected by a spring with constant k_{ij}.
- Suppose they have radii R_i and R_j, and the equilibrium spacing is $s_{ij} = R_i + R_j$.
- Suppose the maximum cell-cell interaction distance is $R_{max} > s_{ij}$.

- Let's calculate:
  - The displacement (directed from *i* to *j*) is:

$$\boldsymbol{d}_{ij} = \boldsymbol{x}_j - \boldsymbol{x}_i$$

  - The distance between *i* and *j* is:

$$d_{ij} = |\boldsymbol{d}_{ij}|$$

  - The normal unit vector from *i* to *j* is:

$$\boldsymbol{n}_{ij} = \frac{\boldsymbol{d}_{ij}}{d_{ij}}$$

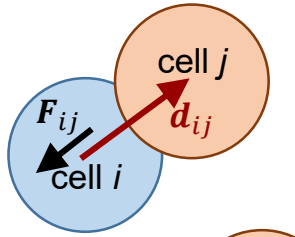
  - If $d_{ij} < R_{\max}$, then the spring-like force acting upon cell *i* is:

$$\boldsymbol{F}_{ij} = k_{ij}(d_{ij} - s_{ij})\boldsymbol{n}_{ij}$$

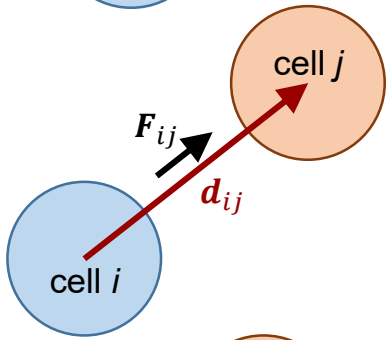
(Otherwise, $\boldsymbol{F}_{ij} = 0$.)

- Sanity check:
  i. If $d_{ij} < s_{ij}$, then the cells are too close together. The force is directed along $-\boldsymbol{n}_{ij}$ away from cell *j* to increase displacement
  ii. If $d_{ij} > s_{ij}$, then the cells are too far apart. The force is directed along $\boldsymbol{n}_{ij}$ towards cell *i* to decrease displacement.
  iii. If $d_{ij} = s_{ij}$, then the cells are at the desired separation, and the net force is zero.

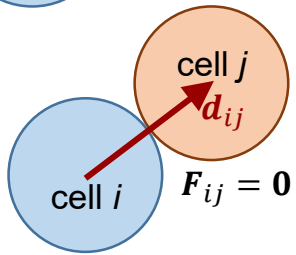
i.



ii.



iii.



Cell velocities: math (2)

- As a very basic force law, let's total up all the forces acting upon cell *i:*

$$\boldsymbol{F}_i = \sum_{j \neq i} \boldsymbol{F}_{ij} - \nu \boldsymbol{v}_i$$

- If forces equilibrate quickly, then we arrive at an inertialess form:

$$\boldsymbol{v}_i = \sum_{j \neq i} \frac{k_{ij}}{\nu} (d_{ij} - s_{ij}) \boldsymbol{n}_{ij}$$

- From here out, let's scale the "spring" constant k_{ij} by the drag coefficient ν and rewrite with the "mechanics strength" $\alpha_{ij} = k_{ij}/\nu$

$$\boldsymbol{v}_i = \sum_{j \neq i} \alpha_{ij} (d_{ij} - s_{ij}) \boldsymbol{n}_{ij}$$

Creating a Cell class (v1)

```
all_cells = list();

class Cell:
    def __init__( self ):
        self.position = np.array( [0.0,0.0] );
        self.velocity = np.array( [0,0] );
        self.mechanics_distance = 30;
        self.equilibrium_spacing = 20;
        self.mechanics_contant = 1;

    def update_velocity( self, dt , env , all_cells ):
        self.velocity = np.array([0,0])
        # spring-like interactions with cells
        for c in all_cells:
            displacement = c.position - self.position
            dist = np.linalg.norm( displacement )
            if( dist < self.mechanics_distance and dist > 1e-16 ):
                dv = dist - self.equilibrium_spacing
                displacement = displacement / dist;
                displacement = displacement * dv;
                displacement = displacement * self.mechanics_contant;
                self.velocity = self.velocity + displacement
        angle = 6.28318530718 * np.random.uniform();
        perturbation_size = 0.1 * self.mechanics_contant;
        self.velocity[0] += perturbation_size * np.cos(angle)
        self.velocity[1] += perturbation_size * np.sin(angle)

    def update_position( self, dt , env, all_cells ):
        self.position = self.position + dt*self.velocity;
```

Go to the Lecture notebook (Section 5) to execute this code.

Testing (part 1)

```
all_cells = list()

# make a plotting function
def plot_cells( env, all_cells ):
    plt.clf()
    for c in all_cells:
        plt.plot( c.position[0] , c.position[1] , 'ko')
    plt.axis('image')
    # plt.axis( [env.a, env.b, env.c, env.d])
    return;

# now make 10 cells with random positions
number_of_cells = 10;
center = [ 0.5*(E.a+E.b) , 0.5*(E.c+E.d)]
for n in range(number_of_cells):
    c = Cell(); # create cell.
    r = 5 * np.random.normal();
    angle = 6.28318530718 * np.random.uniform();
    c.position[0] = center[0] + r * np.cos(angle)
    c.position[1] = center[1] + r * np.sin(angle)
    all_cells.append(c)

plot_cells( E, all_cells )
```

Go to the Lecture notebook (Section 5) to execute this code.

Testing (part 2)

```
dt = 0.1

for n in range( 1000 ):
    for c in all_cells:
        c.update_velocity( dt , E, all_cells )
    for c in all_cells:
        c.update_position( dt , E, all_cells )
    if( n % 100 == 0 ):
        plot_cells( E, all_cells )
        plt.show()
```

Go to the Lecture notebook (Section 5) to execute this code.

Cell class (v2)

- Now, we add division and death methods.
- Division will make a copy of the cell and add it to a list of all cells
- Death will remove the cell from the list of all cells and delete it

Cell class (v2)

```
class Cell:
    # __init__, update_velocity, and update_position as before

    def division( self, all_cells ):
        # make a brand new cell
        c = copy.deepcopy( self );
        # append it to the data structure
        all_cells.append( c );
        # set its position to be near its parent cell
        r = self.equilibrium_spacing;
        angle = np.random.uniform()
        c.position[0] = r*np.cos(angle)
        c.position[1] = r*np.sin(angle)
        return;

    def death( self, all_cells ):
        all_cells.remove( self )
        del self;
```

Go to the Lecture notebook (Section 6) to execute this code.

Testing …

```
all_cells.clear()

# now make 10 cells with random positions
number_of_cells = 10;
center = [ 0.5*(E.a+E.b) , 0.5*(E.c+E.d)]
for n in range(number_of_cells):
    c = Cell(); # create cell.
    r = 5 * np.random.normal();
    angle = 6.28318530718 * np.random.uniform();
    c.position[0] = center[0] + r * np.cos(angle)
    c.position[1] = center[1] + r * np.sin(angle)
    all_cells.append(c)

plot_cells( E, all_cells )
dt = 0.1;

for n in range( 1000 ):
    for c in all_cells:
        c.update_velocity( dt , E, all_cells )
    for c in all_cells:
        c.update_position( dt , E, all_cells )
    if( n % 50 == 0 ):
        n_cells = len( all_cells )
        n_divide = np.random.randint( n_cells );
        all_cells[n_divide].division( all_cells )
    if( n % 200 == 0 ):
        n_cells = len( all_cells )
        n_die = np.random.randint( n_cells );
        all_cells[n_die].death( all_cells )
    if( n % 100 == 0 ):
        plot_cells( E, all_cells )
        plt.show()
```

Go to the Lecture notebook (Section 6) to execute this code.

Cell class (v3)

- Now, we add a update function that uses a cell's birth and death rates to decide when to divide or die (stochastically).

- If an event X happens at rate *r*, then the probability of that rate happening between t and $t + \Delta t$ is:

$$\text{Probability}(X) = r\Delta t$$

- Numerically, if event *X has* probability *p:*
  - Evaluate a uniform random number generator to get $0 \leq u \leq 1$
  - If $u \leq p$, then the event *X* happens. Otherwise it does not.

Cell class (v3)

```
class Cell:
    def __init__( self ):
        #mostly same, but add these
        self.birth_rate = 0.01;
        self.death_rate = 0.005;

    # udpate_velocity, update_position, division, and death as before
    def update( self, dt, env, all_cells ):
        prob_birth = self.birth_rate * dt;
        prob_death = self.death_rate * dt;
        if( np.random.uniform() <= prob_birth ):
            self.division( all_cells );
            return;
        if( np.random.uniform() <= prob_death ):
            self.death( all_cells );
            return;
        return;
```

Go to the Lecture notebook (Section 7) to execute this code.

Testing …

```
all_cells.clear()

# now make 10 cells with random positions
number_of_cells = 10;
center = [ 0.5*(E.a+E.b) , 0.5*(E.c+E.d)]
for n in range(number_of_cells):
    c = Cell(); # create cell.
    r = 5 * np.random.normal();
    angle = 6.28318530718 * np.random.uniform();
    c.position[0] = center[0] + r * np.cos(angle)
    c.position[1] = center[1] + r * np.sin(angle)
    all_cells.append(c)

plot_cells( E, all_cells )
dt = 0.1;

for n in range( 1000 ):
    for c in all_cells:
        c.update( dt , E, all_cells );
    for c in all_cells:
        c.update_velocity( dt , E, all_cells )
    for c in all_cells:
        c.update_position( dt , E, all_cells )
    if( n % 100 == 0 ):
        plot_cells( E, all_cells )
        plt.show()
```

Go to the Lecture notebook (Section 7) to execute this code.

Cell class (v4)

- Now, we make the cell interact with the environment:
  - Find the mesh point nearest to the cell at position (x, y):

$$i = \text{round}\left(\frac{x-a}{\Delta x}\right), \quad j = \text{round}\left(\frac{y-c}{\Delta y}\right)$$

  - Cell consumes resource *c* at rate r_C by the backwards Euler discretization:

$$\frac{c_{i,j}^{n+1} - c_{i,j}^{n}}{\Delta t} = -r_C \, c_{i,j}^{n+1}$$

  - If $c \leq c_{\text{death}}$, then immediate death

  - If $c > c_{\text{death}}$, then we scale the birth rate by:

$$\frac{c - c_{\text{death}}}{1 - c_{\text{death}}}$$

Cell class (v4)

```
class Cell:
    def __init__( self ):
        # add these
        self.death_threshold = 0.3;
        self.consumption_rate = 1;

    # update_velocity and update_position unchanged
    def update( self, dt, env, all_cells ):
        # find my coordinates in the environment
        i = np.int( np.round( (self.position[0]-env.a)/env.dx ) );
        j = np.int( np.round( (self.position[1]-env.c)/env.dy ) );
        # consume substrate (backwards Euler)
        constant = 1.0 + dt*self.consumption_rate;
        env.C[i,j] /= constant;
        # update birth and death rates based on C
        substrate = env.C[i,j]
        birth_rate = self.birth_rate * (substrate - self.death_threshold)/(1-self.death_threshold);
        death_rate = self.death_rate;
        if( birth_rate < 0 ):
            birth_rate = 0.0;
        if( env.C[i,j] < self.death_threshold ):
            self.death(all_cells)
            return;
        prob_birth = birth_rate * dt;
        prob_death = death_rate * dt;
        if( np.random.uniform() <= prob_birth ):
            self.division( all_cells );
            return;
        if( np.random.uniform() <= prob_death ):
            self.death( all_cells );
            return;
        return;
```

Go to the Lecture notebook (Section 8) to execute this code.

New plotting function

```
# make a plotting function
def plot( env, all_cells ):
    plt.figure(1 , figsize=(10,10) )
    plt.clf()
    env.plot()
    for c in all_cells:
        plt.plot( c.position[0] , c.position[1] , 'ko' , markersize=10 ,
            markerfacecolor='w' , alpha=0.5)
    plt.axis('image')
    plt.axis( [env.a, env.b, env.c, env.d])
    plt.title('number of cells: ' + str(len(all_cells)) )
    return;
```

Go to the Lecture notebook (Section 8) to execute this code.

Testing …

```
all_cells.clear()
E.setup( 10000, 0.001 , 1 , 1 )

# now make 25 cells with random positions
number_of_cells = 25;
center = [ 0.5*(E.a+E.b) , 0.5*(E.c+E.d)]
for n in range(number_of_cells):
    c = Cell(); # create cell.
    r = 50 * np.random.normal();
    angle = 6.28318530718 * np.random.uniform();
    c.position[0] = center[0] + r * np.cos(angle)
    c.position[1] = center[1] + r * np.sin(angle)
    all_cells.append(c)

plot( E, all_cells )
dt = 0.1;

for n in range( 2000 ):
    for nn in range(10):
        E.update( 0.1*dt )
    for c in all_cells:
        c.update( dt , E, all_cells );
    for c in all_cells:
        c.update_velocity( dt , E, all_cells )
    for c in all_cells:
        c.update_position( dt , E, all_cells )
    if( n % 200 == 0 ):
        plot( E, all_cells )
        plt.show()
```

Go to the Lecture notebook (Section 8) to execute this code.

Next time:

- Introduce PhysiCell as a more developed agent-based platform.
- Show examples of agent-based PhysiCell modeling in research.
- Explore cloud-hosted models.

Further reading (1)

- **BioFVM method paper (3-D diffusion)**
  A. Ghaffarizadeh, S.H. Friedman, and P. Macklin. BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations. *Bioinformatics* 32(8):1256-8, 2016. DOI: 10.1093/bioinformatics/btv730.

- **PhysiCell method paper (agent-based model)**
  A. Ghaffarizadeh, R. Heiland, S.H. Friedman, S.M. Mumenthaler, and P. Macklin. PhysiCell: an open source physics-based cell simulator for 3-D multicellular systems. *PLoS Comput. Biol.* 14(2):e1005991, 2018. DOI: 10.1371/journal.pcbi.1005991.

- **PhysiBoSS (PhysiCell + MaBoSS for Boolean networks)**
  G. Letort, A. Montagud, G. Stoll, R. Heiland, E. Barillot, P. Macklin, A. Zinovyev, and L. Calzone. PhysiBoSS: a multi-scale agent based modelling framework integrating physical dimension and cell signalling. *Bioinformatics* 35(7):1188-96, 2019. DOI: 10.1093/bioinformatics/bty766.

- **xml2jupyter paper (create GUIs for cloud-hosted models)**
  R. Heiland, D. Mishler, T. Zhang, E. Bower, and P. Macklin. xml2jupyter: Mapping parameters between XML and Jupyter widgets. *Journal of Open Source Software* 4(39):1408, 2019. DOI: 10.21105/joss.01408.

- **PhysiCell+EMEWS (high-throughput 3D PhysiCell investigation)**
  J. Ozik, N. Collier, J. Wozniak, C. Macal, C. Cockrell, S.H. Friedman, A. Ghaffarizadeh, R. Heiland, G. An, and P. Macklin. High-throughput cancer hypothesis testing with an integrated PhysiCell-EMEWS workflow. *BMC Bioinformatics* 19:483, 2018. DOI: 10.1186/s12859-018-2510-x.

- **PhysiCell+EMEWS 2 (HPC accelerated by machine learning)**
  J. Ozik, N. Collier, R. Heiland, G. An, and P. Macklin. Learning-accelerated Discovery of Immune-Tumour Interactions. *Molec. Syst. Design Eng.* 4:747-60, 2019. DOI: 10.1039/c9me00036d.

Further reading (2)

- **A review of cell-based modeling (in cancer):**

  J. Metzcar, Y. Wang, R. Heiland, and P. Macklin. A review of cell-based computational modeling in cancer biology. *JCO Clinical Cancer Informatics* 3:1-13, 2019 (invited review). DOI: 10.1200/CCI.18.00069.

- **Progress on multicellular systems biology:**

  P. Macklin, H.B. Frieboes, J.L. Sparks, A. Ghaffarizadeh, S.H. Friedman, E.F. Juarez, E. Jockheere, and S.M. Mumenthaler. "Progress Towards Computational 3-D Multicellular Systems Biology". In: . Rejniak (ed.), *Systems Biology of Tumor Microenvironment*, chap. 12, pp. 225-46, Springer, 2016. ISBN: 978-3-319-42021-9. (invited author: P. Macklin). DOI: 10.1007/978-3-319-42023-3_12.

- **Challenges for data-driven multicellular systems biology**

  P. Macklin. Key challenges facing data-driven multicellular systems biology. *GigaScience* 8(10):giz127, 2019. DOI: 10.1093/gigascience/giz127

Some models to explore

On nanoHUB:

- **pc4heterogen:** heterogeneous cancer growth (https://nanohub.org/tools/pc4heterogen)
- **pc4cancerbots:** use the "biorobots" as a cell-based cancer therapy (https://nanohub.org/tools/pc4cancerbots)
- **pc4cancerimmune:** basic cancer immunotherapy model (https://nanohub.org/tools/pc4cancerimmune)
- **trmotility**: learn about biased random cell migration (https://nanohub.org/tools/trmotility)
- **pcisa:** learn about an adversarial ecosystem: invader cells are fueled by resource providers, but scout cells seek invaders to recruit attackers, who poison invaders. (https://nanohub.org/tools/pcisa)
- **pc4thanos:** *Avengers Endgame* battle using cell rules (https://nanohub.org/tools/pc4thanos)
- **pc4covid19:** COVID-19 simulation model (https://nanohub.org/tools/pc4covid19)
- **pc4livermedium:** tumor-stroma biomechanical feedbacks (https://nanohub.org/tools/pc4livermedium)