Engineering 448/548 (Spring 2023)

Sessions 3-4: Introduction to Multicellular Systems and Agent-Based Modeling

Paul Macklin, Ph.D.

Intelligent Systems Engineering
Indiana University

January 17 & 19, 2023

Agenda: January 17, 2023

1. Discuss journal article presentations (~10 minutes)
2. Lecture (~65 minutes)
   - Introduction to C++ based software classes
   - Starting agent-based model in C++

Journal article presentations: overview

- **Overall goals:**
  - Stimulate class discussion on multicellular systems
  - Explore potential ideas for team projects
  - Practice scientific presentation & communication skills
- **Duration:** 15-20 minutes (25 max including discussion)
- **Assessment criteria:**
  - Suitability to the class
  - Clear presentation of main question & hypotheses addressed by the problem
  - Succinct summary of main approach
  - Clear and succinct presentation of main results
  - Identify "what's multicellular about this"
  - Quick discussion of what you liked
  - Within time constraints, leaving time for discussion
  - Reasoned attempts to answer class questions

Presentation outline

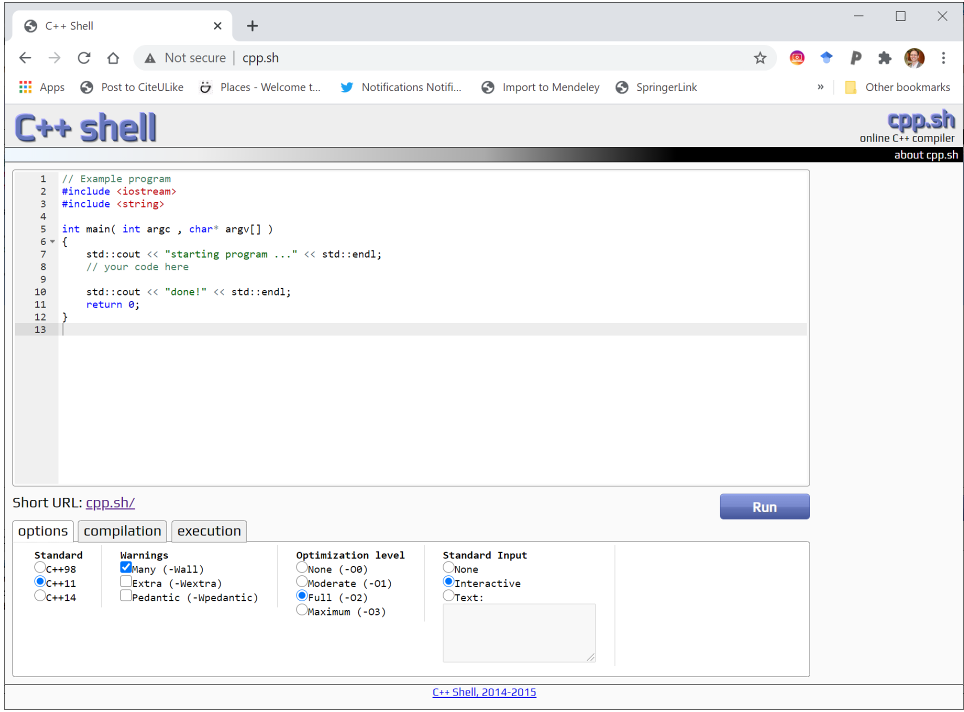
- What was the main goal of the authors? 5 min
  - What problem did they seek to investigate?
  - What was their main hypothesis?
  - Include **brief** background as needed
- What was their main approach? 3 min
- What where their main findings? 4 min
- What about this problem is multicellular? 3 min
- What did you like most about this article or problem? 3 min

C++ review: topics

- minimal program
- Classes
  - constructor
  - destructor
  - copy constructor
  - use a class
- std::vector
  - declaring
  - iterating
  - push and pop
- Pointers
- Random numbers and events
  - uniform random
  - normal random
  - random events

Testing simple C++ online

- Go to:
  https://cpp.sh
- alternative site:
  https://tio.run
- You can execute simple, single-file C++ in a web browser.
- Perfect for testing out simple functions and syntax



A minimal program (online)

```
// Example program
#include <iostream>
#include <string>

int main( int argc , char* argv[] )
{
   std::cout << "starting program ..." << std::endl;
   // your code here

   std::cout << "done!" << std::endl;
   return 0;
}
```

argc: number of arguments (including the executable name!)
argv[]: a vector of strings for all input arguments

Code: [click here]

GitHub: Session_03_04/codes/Example_01

Notice that most functions are prefaced with std::

This is because these functions "live" in the std namespace.

C++: Classes

- A **class** is a **code template** that allows us to create **objects** that bundle data and the functions that act upon them.

- Key components:
  - **member data (attributes):** variables associated with the object
  - **methods:** functions that act upon the object

- Key methods:
  - **constructor:** initializes member variables when you create an instance of the class
  - **destructor:** does cleanup (as needed) when you destroy / deallocate an instance of the class (an object)
  - **copy constructor:** safely copies an existing object to instantiate a new object

- Note that in C++, members of a class can be:
  - **private:** internal data and functions that can only be accessed within methods of the class
  - **public:** data and functions that can be accessed by anything.

Starting example

```
#include <iostream>
#include <string>

class Person
{
    private:
    public:
        std::string name;
        double age;
    Person(); // default constructor
    ~Person(); // default destructor
};

Person::Person()
{
    name = "Nobody";
    age = 0.0;
    return;
}

Person::~Person()
{ return; }


int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl;
    // your code here

    Person bob;
    std::cout << bob.name << " is " << bob.age << std::endl;

    bob.name = "Dennis";
    bob.age = 37;
    std::cout << bob.name << " is " << bob.age << std::endl;

    std::cout << "done!" << std::endl;
    return 0;
}
```

Declare the constructor and destructor in the class definition

Implement the constructor and destructor in the class definition

Instantiate a Person object. See how it uses the default constructor to initialize

Then change its (public) data and re-display

Code: [click here]

GitHub: Session_03_04/codes/Example_02

Add copy constructor & member function

```
class Person
{
    // ...
    Person( Person& p ); // copy constructor

    void display( void );
};
// ...

Person::Person( Person& p )
{
    name = p.name;
    age =p.age;
    return;
}

void Person::display( void )
{
    std::cout << name << " is " << age << " years old." << std::endl;
    return;
}

int main( int argc , char* argv[] )
{
    // ...

    Person bob;
    bob.display();

    bob.name = "Dennis";
    bob.age = 37;
    bob.display();

    Person charlie = bob;
    charlie.name = "Charlie";
    charlie.display();

    // ...
}
```

Person& p passes the object (p) to the function ***by reference.***

This means that rather than making a new copy of p for use by the function, we are referring to the original object.

This avoids the expensive costs of allocation and copying. It also allows the function to act directly upon p.

See how the new "display" function really cleans up the code and cuts down on (error-prone) repetition.

Also, if you later want to change *how* an object is displayed, you only have to change it once.

Code: [click here]

GitHub: Session_03_04/codes/Example_03

C++: Vectors

- C++ has a built-in class of vectors:
  - std::vector<some_type>
    - You must include <vector> to use it
  - Makes a vector of objects of type some_type
    - Various constructor functions let you set initial state more easily
  - These can be dynamically accessed by index
    - Use vector::size to get the current size
  - You can dynamically shrink and grow them
    - push_back(an_object) adds an_object to the end of the vector
    - pop_back() trims the last element from the end of the vector
    - vector::resize(num_objects) lets you grow/shrink
    - vector::clear() resizes to size 0

C++ Vector example

```
#include <iostream>
#include <string>
#include <vector>

int main( int argc , char* argv[] )
{
   std::cout << "starting program ..." << std::endl;
   // your code here

    std::vector<double> v( 10 ); // create a vector of doubles of size 10
    for( int i=0 ; i < v.size(); i++ )
    { std::cout << i << " : " << v[i] << std::endl; }
    std::cout << std::endl;

    // add an element to the vector
    v.push_back( 42.041 );
    for( int i=0 ; i < v.size(); i++ )
    { std::cout << i << " : " << v[i] << std::endl; }
    std::cout << std::endl;

    // trim off 3 last elements of the vector
    v.pop_back();
    v.pop_back();
    v.pop_back();
    for( int i=0 ; i < v.size(); i++ )
    { std::cout << i << " : " << v[i] << std::endl; }
    std::cout << std::endl;

   std::cout << "done!" << std::endl;
   return 0;
}
```

Code: [click here]

GitHub: Session_03_04/codes/Example_04

C++ Vector example (2) (with a function)

```
#include <iostream>
#include <string>
#include <vector>

void display_vector( std::vector<double>& v )
{
    for( int i=0; i < v.size(); i++ )
    { std::cout << i << ": " << v[i] << std::endl; }
    std::cout << std::endl;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl;
    // your code here

    std::vector<double> v( 10 ); // create a vector of doubles of size 10
    display_vector( v );

    for( int i=0 ; i < v.size(); i++ ) // act on elements of v
    { v[i] = i; }
    display_vector( v ) ;

    // add an element to the vector
    v.push_back( 42.041 );
    display_vector( v ) ;

    // trim off 3 last elements of the vector
    v.pop_back();
    v.pop_back();
    v.pop_back();
    display_vector( v ) ;

    std::cout << "done!" << std::endl;
    return 0;
}
```

Code: [click here]

GitHub: Session_03_04/codes/Example_05

C++ Vector example (3) (more fun)

```
#include <iostream>
#include <string>
#include <vector>

void display_vector( std::vector<double>& v )
{
    for( int i=0; i < v.size(); i++ )
    { std::cout << i << ": " << v[i] << std::endl; }
    std::cout << std::endl;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl;
    // your code here

    std::vector<double> v( 10 , 3.0 ); // create a vector of doubles of size 10, initialized to 3.0
    display_vector( v );

    v.resize( 3 ); // shrink or grow to size 3.
    display_vector( v );

    v.resize( 12 ); // shrink or grow to size 12.
    display_vector( v );


    for( int i=0 ; i < v.size(); i++ ) // act on elements of v
    { v[i] = i; }
    display_vector( v ) ;


    std::cout << "done!" << std::endl;
    return 0;
}
```

Code: [click here]

GitHub: Session_03_04/codes/Example_06

C++ Vector example (4) (even more fun)

```
//...
std::vector<double> operator+( std::vector<double>& v, std::vector<double>& w )
{
    unsigned int m = v.size();
    if( w.size() < m )
    { m = w.size(); }
    std::vector<double> z( m );
    for( unsigned int i=0; i < m ; i++ )
    { z[i] = v[i] + w[i]; }
    return z;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl; // your code here

    std::vector<double> v = { 0.0, 1.0 , 2.0 }; // handy way to start!
    display_vector( v );

    for( unsigned int i=0 ; i < v.size(); i++ ) // act on elements of v
    { v[i] = i; }
    display_vector( v ) ;

    std::vector<double> w = v; // copy v to w
    for( unsigned int i=0 ; i < w.size(); i++ ) // act on elements of w
    { w[i] = i*i; }
    display_vector( w ) ;

    std::vector<double> z = v + w; // add v and w
    display_vector( z );

    z.clear(); // clear z
    display_vector( z );

    std::cout << "done!" << std::endl;
    return 0;
}
```

overload an addition operator for more intuitive work with vectors.

Code: [click here]

GitHub: Session_03_04/codes/Example_07

C++: Pointers

- A pointer lets you refer to (and access) an object directly.

```
Person* pPerson;              // pPerson is a pointer to an instance of Person
pPerson->data_member;         // access a data member
(*pPerson).data_member;       // another way to access the data member
```

- A pointer also lets you declare new objects in global memory and let them persist.

```
Person* pPerson = NULL;
pPerson = new Person();       // make a new person
```

Pointer examples with a class

```
#include <iostream>
#include <string>

class Person
{
    private:
    public:
        std::string name;
        double age;
    Person(); // default constructor
    ~Person(); // default destructor
    Person( Person& p ); // copy constructor

    void display( void );
};

Person::Person()
{
    name = "Nobody";
    age = 0.0;
    return;
}

Person::~Person()
{ return; }
```

```
Person::Person( Person& p )
{
    name = p.name;
    age =p.age;
    return;
}

void Person::display( void )
{
    std::cout << name << " is " << age << " years old." << std::endl;
    return;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl; // your code here

    Person* pPerson = NULL; // un-initiated pointer to a person
    pPerson = new Person; // create a new person
    pPerson->display(); // display this new person

    pPerson->name = "Pointy"; // operate on the pointer
    (*pPerson).display();

    std::cout << "done!" << std::endl;
    return 0;
}
```

Code: [click here]

GitHub: Session_03_04/codes/Example_08

Combining pointers and vectors

- You can use a vector of pointers to easily and efficiently keep track of objects, particularly if they are created dynamically.

- First, we'll make a few Person objects and manually add them to a list of all Person objects.

C++ vector of object pointers

```
#include <iostream>
#include <string>
#include <vector>

class Person
{
    private:
    public:
        std::string name;
        double age;
    Person(); // default constructor
    ~Person(); // default destructor
    Person( Person& p ); // copy constructor
    void display( void );
};

Person::Person()
{
    name = "Nobody";
    age = 0.0;
    return;
}

Person::~Person()
{ return; }

Person::Person( Person& p )
{
    name = p.name;
    age = p.age;
    return;
}
```

```
void Person::display( void )
{
    std::cout << name << " is " << age << " years old." << std::endl;
    return;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl; // your code here

    Person* pPerson = NULL; // un-initiated pointer to a person

    std::vector<Person*> all_persons_list; // vector of all Person objects

    // make three objects and add them to this list
    pPerson = new Person;
    pPerson->name = "Alice";
    all_persons_list.push_back( pPerson );

    pPerson = new Person;
    pPerson->name = "Bob";
    all_persons_list.push_back( pPerson );

    pPerson = new Person;
    pPerson->name = "Charlie";
    all_persons_list.push_back( pPerson );

    // iterate through all of the Person objects
    for( unsigned int n = 0; n < all_persons_list.size() ; n++ )
    { all_persons_list[n]->display(); }

    std::cout << "done!" << std::endl;
    return 0;
```

Code: [click here]

GitHub: Session_03_04/codes/Example_09

Now, let's automate!

- Here's where constructors and destructors come in handy: we can automate adding / removing from this global list.
- Let's show how!

```cpp
#include <iostream>
#include <string>
#include <vector>
class Person
{
        private:
        public:
            std::string name;
            double age;
            Person();     // default constructor
        ~Person();        // default destructor
        Person( Person& p );   // copy constructor

        void display( void );
};

std::vector<Person*> all_persons_list;
Person::Person()
{
        name = "Nobody";
        age = 0.0;
        all_persons_list.push_back( this ); // add this object to the list
        return;
}

Person::~Person()
{
    // find and remove "me" in the list of all persons
    int n = 0;
    bool done = false;
    while( done == false )
    { // search for "me"
        if( all_persons_list[n] == this )
        {
            // put last item on the list in my place
            all_persons_list[n] = all_persons_list.back();
            // shrink list
            all_persons_list.pop_back();
            done = true;
        }
        n++;
    }
    return;
}
```

```cpp
void Person::display( void )
{
    std::cout << name << " is " << age << " years old." << std::endl;
    return;
}
Person::Person( Person& p )
{
    name = p.name;
    age = p.age;
    all_persons_list.push_back( this ); // add this object to the list
    return;
}

int main( int argc , char* argv[] )
{
    std::cout << "starting program ..." << std::endl; // your code here

    Person* pPerson = NULL; // un-initiated pointer to a person

    // make three objects and add them to this list
    pPerson = new Person;
    pPerson->name = "Alice";

    pPerson = new Person;
    pPerson->name = "Bob";

    pPerson = new Person;
    pPerson->name = "Charlie";

    // iterate through all of the Person objects
    std::cout << std::endl;
    for( unsigned int n = 0; n < all_persons_list.size() ; n++ )
    { all_persons_list[n]->display(); }

    // delete Bob
    delete all_persons_list[1];

    // iterate through all of the Person objects
    std::cout << std::endl;
    for( unsigned int n = 0; n < all_persons_list.size() ; n++ )
    { all_persons_list[n]->display(); }

    std::cout << "done!" << std::endl;
    return 0;
```

Code: [click here]

GitHub: Session_03_04/codes/Example_10

C++: randomness

- C++ has a built-in pseudorandom number generators.
  - Always use these instead of rand()!

- Create a random number generator (64-bit Mersenne Twister)
  - Then use this to generate random numbers with correct distributions:

  - Uniform random number generator U(a,b);
    - uniformly distributed numbers from a to b. Usually a = 0, b = 1.
  - Normal number generator N(a,b)
    - Gaussian distribution with mean a, standard deviation b. Usually a = 0, b = 1.

- How to evaluate an event.
  - Suppose event X happens with probability *p*
  - First, get a uniform number *u* from U(0,1).
    - If $u \leq p$, then X happens. Otherwise, it does not.

```cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include <random>

int main( int argc , char* argv[] )
{
   std::cout << "starting program ..." << std::endl;
   // your code here

   std::mt19937_64 generator( 0 );
   // initialize 64-bit Mersenne twister with seed 0

   std::uniform_real_distribution<> uniform(0.0,1.0);
   std::cout << "uniform:" << std::endl;
   for (int n = 0; n < 10; n++)
   {
      // use uniform() to transform a random number
      // from generator to a number in U(0,1).
      std::cout << uniform(generator) << ' ';
   }
   std::cout << std::endl << std::endl;

   std::normal_distribution<> normal(0.0,1.0);
   std::cout << "normal:" << std::endl;
   for (int n = 0; n < 10; n++ )
   {
      // use uniform() to transform a random number
      // from generator to a number in N(0,1).
      std::cout << normal(generator) << ' ';
   }
   std::cout << std::endl << std::endl;

   double probability = 0.15;
   int number_of_tests = 100000;
   int number_of_events = 0;

   std::cout << "getting random events:" << std::endl;
   for (int n = 0; n < number_of_tests; n++)
   {
      if( uniform(generator) <= probability )
      { number_of_events++; }
   }

   double fraction = (double) number_of_events
      / (double) number_of_tests;
   std::cout << "fraction of events: " << fraction
      << " vs " << probability << std::endl << std::endl;

   std::cout << "done!" << std::endl;
   return 0;
}
```

Code: [click here]

GitHub: Session_03_04/codes/Example_11

Let's use this to make a starting model with a basic cell class. (v1)

Code design (version 1)

- This code is available at:
  - GitHub: Session_03_04/codes/ABM/
- Overall structure:
  - main.cpp — overall program loop
  - Cell.cpp and Cell.h — define cell agents
  - Makefile — tells the compiler how to operate (compiling and linking…)
- I will put different versions in different directories. To choose a version, overwrite the source code, and compile:
  - make version v=VERSION_NUMBER
    - e.g., `make version V=1` copies everything from the directory v1

main structure (main.cpp)

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <random>
#include <cmath>

#include "Cell.h"
std::string version = "v1";
int main( int argc, char* argv[] )
{
   std::cout << "starting version " << version
      << " ..." << std::endl;
   // parse settings

   // create cell types

   // create environment

   // place cells

   long double t = 0;
   double max_time = 5 * 24 * 60 ;
   double output_interval = 720;
   double next_output_time = 0.0;
   double dt = 0.1;

   while( t < max_time + 0.01*dt )
   {
      // output?
      if( fabs(t-next_output_time) < 0.1*dt )
      {
         std::cout << t << " of "
            << max_time << " (min)" << std::endl;
         next_output_time += output_interval;
      }

      // update phenotypes

      // birth and death

      // update positions

      t += dt;
   }

   std::cout << "done!"  << std::endl;

   return 0;
}
```

Makefile

define a variable

conditional

rule "**all**" depends upon main.o and Cell.o

get the value of the output variable

```
GCC := g++

ifdef PHYSICELL_CPP
   GCC := $(PHYSICELL_CPP)
endif

CFLAGS := -std=c++11 –m64
CC := $(GCC) $(CFLAGS)

output := model

all: main.o Cell.o
   $(CC) -o $(output) main.o Cell.o

main.o: main.cpp
   $(CC) -c main.cpp

Cell.o: Cell.cpp
   $(CC) -c Cell.cpp

clean:
   rm -f *.o $(output)

version:
   cp v$(V)/* .
   make
```

Cell class design (for now)

- Cell class has:
  - member data:
    - 2-D position
    - radius
    - live or dead state
    - birth rate, death rate
    - max interaction distance
    - adhesion strength
  - methods
    - constructor, destructor
    - division
    - death
    - movement

Declaring the class (in Cell.h)

```
#ifndef __Cell_h__
#define __Cell_h__

#include <vector>

class Cell
{
   private:
   public:
      std::vector<double> position;
      double radius;
      double birth_rate;
      double death_rate;

      Cell();
      Cell( Cell& copy_me );
      ~Cell();
      bool division( void );
      bool death( void );
      bool movement( double dt );
};

extern std::vector<Cell*> all_cells;

#endif
```

Constructors and Destructor (Cell.cpp)

```
#include "Cell.h"
std::vector<Cell*> all_cells;
Cell::Cell()
{
	position = {0.0, 0.0};
	radius = 5.0;
	birth_rate = 0.001;
	death_rate = 0.00001;

	all_cells.push_back( this );
	return;
}

Cell::Cell( Cell& copy_me )
{
	position = copy_me.position;
	radius = copy_me.radius;
	birth_rate = copy_me.birth_rate;
	death_rate = copy_me.death_rate;

	all_cells.push_back( this );
	return;
}
```

```
Cell::~Cell()
{
	// find and remove self
	int n = 0;
	bool done = false;
	while( done == false )
	{
		if( all_cells[n] == this )
		{
			all_cells[n] = all_cells.back();
			all_cells.pop_back();
			done = true;
		}
		n++;
	}
	return;
}
```

Let's try it! (v1)

```
make version V=1
make && ./model
```

Command Prompt

```
C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>make version V=1
cp v1/* .
make
make[1]: Entering directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'
g++  -c main.cpp
g++  -c Cell.cpp
g++  -o model  main.o Cell.o
make[1]: Leaving directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>model
starting version v1 ...
0 of 7200 (min)
720 of 7200 (min)
1440 of 7200 (min)
2160 of 7200 (min)
2880 of 7200 (min)
3600 of 7200 (min)
4320 of 7200 (min)
5040 of 7200 (min)
5760 of 7200 (min)
6480 of 7200 (min)
7200 of 7200 (min)
done!

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>
```

Next time

- Continue building the agent model:
  - v2: add cell birth and death
  - v3: cell-cell mechanics
  - v4: saving model state
- reading output in python
- visualization in python

Agenda: January 19, 2023

1. Discuss journal article presentations (~10 minutes)
2. Lecture (~65 minutes)
   - Introduction to C++ based software classes
   - Starting agent-based model in C++

Last time

- We reviewed C++ elements to help us with agent-based modeling
- We built a basic Cell class
- We built a v1 simulator that has a (very empty!) main loop

- This code is available on GitHub
  - See Session_03_04/codes/ABM

- To build version 1:

```
make version V=1
```

Now let's move to v2

version 2 updates

- First, let's implement division and death
  - division: make a new cell (via copy constructor), and place it to the right (for now)
  - death: call the destructor by deleting the cell

- We'll need a uniform random number function

- Let's loop over all cells to "decide" to look for birth/death events.
  - If the birth rate is *b,* then the probability of a birth event in $[t,t+\Delta t]$ is $b\,\Delta t$
    - ♦ add it to a list of cells that need to do a division event, then process them all
  - If the death rate is *d,* then the probability of a birth event in $[t,t+\Delta t]$ is $d\,\Delta t$
    - ♦ add it to a list of cells that need to do a death event, then process them all

- We'll make a function that automates this.

division and death

```
bool Cell::division( void )
{
   // create a new cell
   Cell* pNewCell;
   // use the copy constructor
   pNewCell = new Cell( *this );

   // place it to the right
   pNewCell->position[0] += 2.0 * (pNewCell->radius);

   return true;
}

bool Cell::death( void )
{
   delete this;
   return true;
}
```

Uniform random numbers

```
// in Cell.h

long double uniform_random( void );

// in Cell.cpp

long double uniform_random()
{
   // create 64-bit Mersenne twister, seed zero
   // static so it persists between function calls
   static std::mt19937_64 generator(0);

   // create uniform distribution, static to persist
   // between function calls
   static std::uniform_real_distribution<> uniform(0.0,1.0);
   return uniform(generator);
}
```

birth / death function

```
// in Cell.h

bool check_for_birth_and_death( double dt );

// in Cell.cpp

bool check_for_birth_and_death( double dt )
{
    Cell* pCell = NULL;

    std::vector<Cell*> birth_list;
    std::vector<Cell*> death_list;
    for( int n=0; n < all_cells.size() ; n++ )
    {
        pCell = all_cells[n];
        // birth event?
        if( uniform_random() <= pCell->birth_rate * dt )
        { birth_list.push_back(pCell); }

        // death event?
        if( uniform_random() <= pCell->death_rate * dt )
        { death_list.push_back(pCell); }
        n++;
    }
    // process births
    for( int n=0; n < birth_list.size(); n++ )
    { birth_list[n]->division(); }

    // process deaths
    for( int n=0; n < death_list.size(); n++ )
    { death_list[n]->death(); }

    return true;
}
```

Flag for later processing so that all_cells does not change size as we're working through it.

Updating the main simulation loop (1)

```
// place cells
Cell* pCell = NULL;
pCell = new Cell;
pCell->position[1] = -10;

pCell = new Cell;
pCell->position[1] = 10;

pCell = new Cell;
pCell->position[0] = -10;

pCell = new Cell;
pCell->position[0] = 10;

// ...
```

Updating the main simulation loop (2)

```
while( t < max_time + 0.01*dt )
{
   // output?
   if( fabs(t-next_output_time) < 0.01*dt )
   {
      std::cout << t << " of " << max_time << " (min)" << std::endl;
      std::cout << "\tCell count: " << all_cells.size() << std::endl;
      next_output_time += output_interval;
   }

   // update phenotypes

   // birth and death
   check_for_birth_and_death( dt );

   // update positions

   t += dt;
}
```

Let's try it! (v2)

```
make version V=2
make && ./model
```

Command Prompt

```
C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>make version V=2
cp v2/* .
make
make[1]: Entering directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'
g++  -c main.cpp
g++  -c Cell.cpp
g++  -o model  main.o Cell.o
make[1]: Leaving directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>model
starting version v2 ...
0 of 7200 (min)
        Cell count: 4
720 of 7200 (min)
        Cell count: 4
1440 of 7200 (min)
        Cell count: 6
2160 of 7200 (min)
        Cell count: 13
2880 of 7200 (min)
        Cell count: 19
3600 of 7200 (min)
        Cell count: 25
4320 of 7200 (min)
        Cell count: 36
5040 of 7200 (min)
        Cell count: 58
5760 of 7200 (min)
        Cell count: 92
6480 of 7200 (min)
        Cell count: 126
7200 of 7200 (min)
        Cell count: 181
done!

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>
```

Now let's move to v3

version 3 updates

- Let's add some cell-cell mechanics:
  - Cell *i* searches within an interaction distance d_{max}
    - If cell *j* is within that distance, it uses a spring-like force to update its velocity
- Let's update the main loop to calculate all cell velocities
- Let's update the main loop to calculate all cell positions
- Let's also save the output to a text file

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:

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

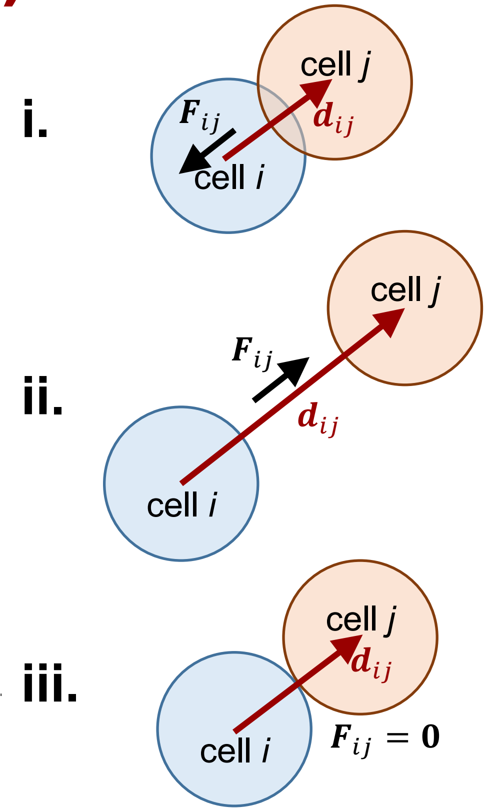
(Otherwise, $\boldsymbol{F}_{ij} = 0$.)

- 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.



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:

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

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

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

Cell velocities: code plan

- First, we'll write a member function to add the contribution of cell *j* to the velocity of cell *i.*
- Then, we'll write a member function that calculates cell *i* velocity by looking at every cell *j.*
- Then, we'll write a function that calculates these velocities for all cells, and then updates their positions.

Cell velocities: function declarations (Cell.h)

```
#ifndef __Cell_h__
#define __Cell_h__

// ...

class Cell
{
   private:
   public:
      // ...
      std::vector<double> velocity;
      double mechanics_strength;
      double max_interaction_distance;
      void mechanics_interaction( Cell* pCell );
      void mechanics_interactions( void );
      // ...
};

// ...
bool update_mechanics( double dt );
// ...

#endif
```

Cell velocities: (Cell.cpp) (part 1)

```
void Cell::mechanics_interaction( Cell* pCell )
{
   if( pCell == this )  // don't interact with yourself!
   { return; }

   // calculate displacement
   double DisplacementX = pCell->position[0] - position[0];
   double DisplacementY = pCell->position[1] - position[1];

   // calculate distance
   double distance = sqrt( DisplacementX*DisplacementX + DisplacementY*DisplacementY );

   if( distance > max_interaction_distance ) // are we in range?
   { return; }

   // normalize displacement (don't divide by zero)
   distance += 1e-16;
   DisplacementX /= distance;
   DisplacementY /= distance;

   // calculate equilibrium spacing
   double spacing = radius + pCell->radius;

   // contribute to velocity
   double coefficient = mechanics_strength * ( distance-spacing );
   velocity[0] += coefficient * DisplacementX;
   velocity[1] += coefficient * DisplacementY;
   return;
}
```

Cell velocities: (Cell.cpp) (part 2)

```
void Cell::mechanics_interactions( void )
{
   velocity[0] = 0;
   velocity[1] = 0;

   for( int n=0 ; n < all_cells.size(); n++ )
   { mechanics_interaction( all_cells[n] ); }

   return;
}


bool update_mechanics( double dt )
{
   // update velocities
   for( int n=0 ; n < all_cells.size(); n++ )
   { all_cells[n]->mechanics_interactions(); }

   // update positions
   for( int n=0 ; n < all_cells.size(); n++ )
   {
      all_cells[n]->position[0] += dt* all_cells[n]->velocity[0];
      all_cells[n]->position[1] += dt* all_cells[n]->velocity[1];
   }

   return true;
}
```

Update constructors (Cell.cpp)

```
Cell::Cell()
{
   position = {0.0, 0.0};
   radius = 5.0;
   birth_rate = 0.001;
   death_rate = 0.00001;

   mechanics_strength = 0.01;
   max_interaction_distance = 1.25 * (2*radius);
   velocity = {0,0};

   all_cells.push_back( this );
   return;
}

Cell::Cell( Cell& copy_me )
{
   position = copy_me.position;
   radius = copy_me.radius;
   birth_rate = copy_me.birth_rate;
   death_rate = copy_me.death_rate;

   mechanics_strength = copy_me.mechanics_strength;
   max_interaction_distance = copy_me.max_interaction_distance;
   velocity = {0,0};

   all_cells.push_back( this );
   return;
}
```

Update main loop (main.cpp)

```
while( t < max_time + 0.01*dt )
{
   // output?
   if( fabs(t-next_output_time) < 0.01*dt )
   {
      std::cout << t << " of " << max_time << " (min)" << std::endl;
      std::cout << "\tCell count: " << all_cells.size() << std::endl;

      next_output_time += output_interval;
   }

   // update phenotypes

   // birth and death
   check_for_birth_and_death( dt );

   // update positions
   update_mechanics( dt );

   t += dt;
}
```

Let's try it! (v3)

```
make version V=3
make && ./model
```

Command Prompt

```
C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>make version V=3
cp v3/* .
make
make[1]: Entering directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'
g++  -c main.cpp
g++  -c Cell.cpp
g++  -o model  main.o Cell.o
make[1]: Leaving directory `/c/GitHub/PhysiCell-Training/CPP_ABM_crashcourse'

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>model
starting version v3 ...
0 of 7200 (min)
        Cell count: 4
720 of 7200 (min)
        Cell count: 4
1440 of 7200 (min)
        Cell count: 6
2160 of 7200 (min)
        Cell count: 13
2880 of 7200 (min)
        Cell count: 19
3600 of 7200 (min)
        Cell count: 25
4320 of 7200 (min)
        Cell count: 36
5040 of 7200 (min)
        Cell count: 58
5760 of 7200 (min)
        Cell count: 92
6480 of 7200 (min)
        Cell count: 126
7200 of 7200 (min)
        Cell count: 181
done!

C:\GitHub\PhysiCell-Training\CPP_ABM_crashcourse>
```

Now let's move to v4

version 4 updates

- Save data to a comma-separated text file (csv)
- Load and visualize data in Python

Writing data: code plan

- Each row (line) of CSV file is a cell
  - x, y
  - radius
  - velocity_x
  - velocity_y
  - birth rate
  - death rate
  - mechanics strength
  - (add more columns as needed)

- Make a new save.h and save.cpp (with changes to Makefile)
  - Write a function to save to a file
  - Write a function to choose a file name
  - Write a function that automates this.

- Put the save function in the main loop

Writing data: save.h

```
#ifndef __save_h_
#define __save_h_

#include <vector>
#include <random>
#include <ctime>

#include <fstream>
#include <string>
#include <iostream>

#include "Cell.h"

std::string output_filename( void );
bool output( std::string filename );
bool save_data( void );

#endif
```

Saving data requires knowledge of **Cell**.

Writing data: save.cpp (part 1)

```
bool output( std::string filename )
{
   std::cout << "\tSaving data to " << filename << " ... ";

   std::ofstream out( filename , std::ofstream::out );

   // x,y,radius,vx,vy,birth,death,mechanics,

   for( int n=0 ; n < all_cells.size() ; n++ )
   {
      Cell* pC = all_cells[n];
      out << pC->position[0] << "," << pC->position[1] << ","
         << pC->radius << ","
         << pC->velocity[0] << "," << pC->velocity[1] << ","
         << pC->birth_rate << "," << pC->death_rate << ","
         << pC->mechanics_strength << std::endl;
   }

   out.close();

   std::cout << " done!" << std::endl;

   return true;
}
```

ofstream writes to ***filename***

iterate through all cells

Writing data: save.cpp (part 2)

```
std::string output_filename( void )
{
   static int output_index = 0;
   static std::string output_directory = "./data";

   char temp [1024];
   sprintf( temp , "%s/output%08u.csv" , output_directory.c_str() , output_index );
   output_index++;

   std::string output = temp;
   return output;
}

bool save_data( void )
{
   std::string filename = output_filename();
   return output( filename );
}
```

Automatically keep track of how many files we have written.

Good practice to *not* save to root directory!

8-digits, padded with zeros

sprintf requires an old-fashioned C string. Grab it from a std::string like this

Update main.cpp

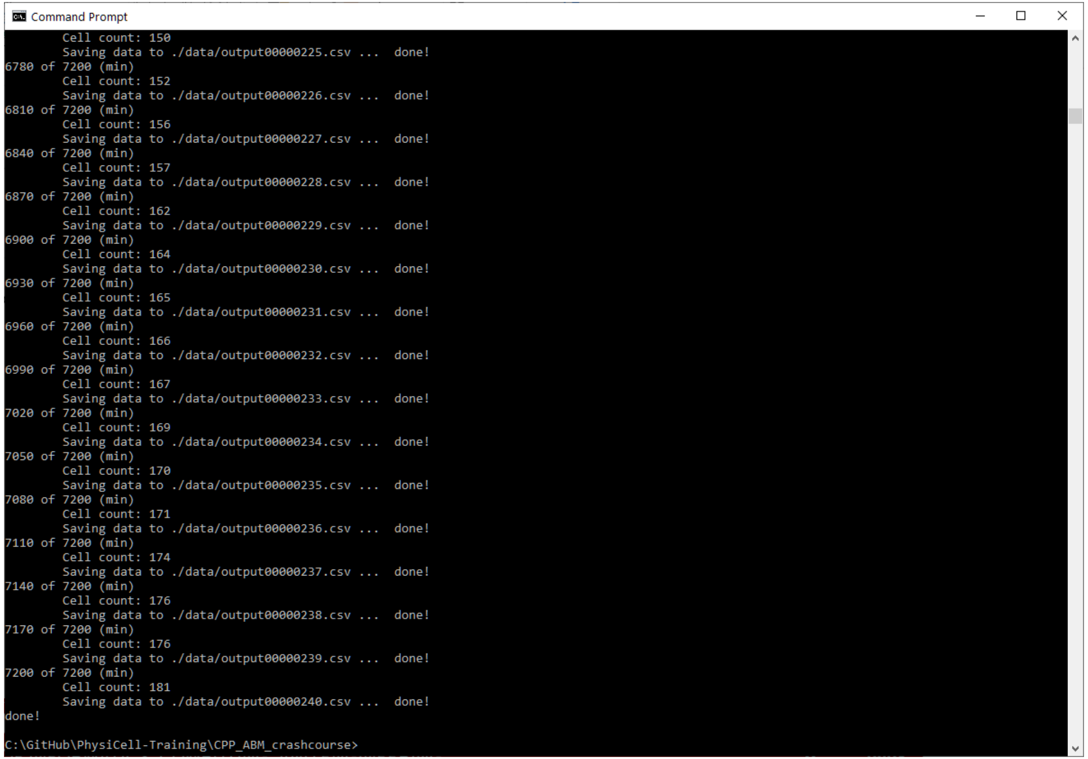
```
// ...
#include "Cell.h"
#include "save.h"
// ...

while( t < max_time + 0.01*dt )
   {
      // output?
      if( fabs(t-next_output_time) < 0.01*dt )
      {
         std::cout << t << " of " << max_time << " (min)" << std::endl;
         std::cout << "\tCell count: " << all_cells.size() << std::endl;

         save_data();

         next_output_time += output_interval;
      }

      // update phenotypes

      // birth and death
      check_for_birth_and_death( dt );

      // update positions
      update_mechanics( dt );

      t += dt;
   }
```

Update Makefile

```
GCC := g++ # macOS users need their compiler here

output := model

all: main.o Cell.o save.o
	$(GCC) -o $(output) main.o Cell.o save.o

main.o: main.cpp
	$(GCC) -c main.cpp

Cell.o: Cell.cpp
	$(GCC) -c Cell.cpp

save.o: save.cpp
	$(GCC) -c save.cpp

clean:
	rm -f *.o $(output)

version:
	cp v$(V)/* .
	make
```

Let's try it! (v4)

```
make version V=4
make && ./model
```



Data visualization in Jupyter

Create / open a notebook in data directory

- probably easiest to:
  - launch qtconsole (from anaconda)
  - navigate to your data directory
  - launch Jupyter lab from within qtconsole:

```
!jupyter lab
```

- This will now have you in Jupyter lab in the right directory. The "qt" means you'll have extra plotting and windowing options, such as plots in an external window.

Code plan

- Load CSV to an array
- Process array to get:
  - Positions
  - Radii
  - Velocities
  - birth_rates
  - death_rates
  - mechanics_strengths
- create a function to plot all cells
- script this to "animate" the visualization

Loading and processing the CSV

```
import numpy as np
import matplotlib.pyplot as plt

# set plot options
plt.rcParams['figure.figsize'] = (15,15) # Pick something here, bigger than (6.0,4.0)
plt.rcParams['font.size'] = 25           # pick something bigger than 10
plt.rcParams['lines.markersize'] = 7     # bigger markers

def load_data( index ):
    filename = "output" + "%08u" % index + ".csv"
    print( "Loading data from " + filename + " ...")

    file = open(filename)
    data = np.double( np.loadtxt(file, delimiter=",") )

    num_cells,num_cols = data.shape
    positions = np.double( data[:,0:2])
    radii =  np.double( data[:,2])
    velocities = np.double( data[:,3:5])
    birth_rates = np.double( data[:,5])
    death_rates = np.double( data[:,6])
    mechanics_strengths = np.double( data[:,7])

    return positions,radii,velocities,birth_rates,death_rates,mechanics_strengths
```

plot all cells

```
def plot_cells( positions, radii, velocities ):
    plt.scatter( positions[:,0] , positions[:,1] , s=radii*10 )
    plt.quiver( np.transpose(positions[:,0]) , np.transpose(positions[:,1]),
        np.transpose(velocities[:,0]) , np.transpose(velocities[:,1]) ,
        scale=1, headwidth=4,headlength=3,headaxislength=2,width=.003)
    scale = 75
    m,n = positions.shape
    center = [ np.mean( positions[:,0]) , np.mean( positions[:,1]) ]
    width = 150;
    axes = [center[0]-0.5*width,center[0]+0.5*width,
        center[1]-0.5*width,center[1]+0.5*width]
    plt.axis(axes)
    ax = plt.gca()
    ax.set_aspect('equal')
```

get the aspect ratio right

Let's make a function to automate / animate

```
def animate( interval , last_index ):
    %matplotlib qt

    plt.figure(1)
    plt.pause(5)

    n = 0
    while( n < last_index+1 ):
        plt.figure(1)
        plt.clf()
        positions,radii,velocities,birth_rates,death_rates,
            mechanics_strengths=load_data(n)
        plot_cells( positions, radii, velocities )
        plt.title( n );
        plt.pause(0.2)
        n += interval
```

Select for plotting in external / popup window

give us a few seconds to select the window before the main plot loop

a slight pause is enough to get the updated plot to render as an "animation" while we loop

Let's try it!

```
# plot every other frame (interval = 2), up to frame 240:

animate( 2 , 240 )
```

Let's try one more thing

```
// in main.cpp, change max_time to 10*24*60

// permanently turn off proliferation and increase death after 7200 minutes:

void change_phenotypes( double t )
{
   static bool change_made = false;
   if( change_made == true )
   { return; }

   if( t > 7200 )
   {
      for( int n = 0; n < all_cells.size() ; n++ )
      {
         all_cells[n]->birth_rate = 0.0;
         all_cells[n]->death_rate *= 50.0;
      }

      change_made = true;
   }

   return;
}

// add change_phenotypes(t); to the main loop, at "update phenotypes"
```

Use static variable (persists in the scope of the function between function calls) to detect if this code has been executed.

Only execute it once.

I put this code in main_v4alt.cpp in v4