Application of C++ in Computational Cancer Modeling

RUIBO ZHANG

Cppcon
The C++ Conference

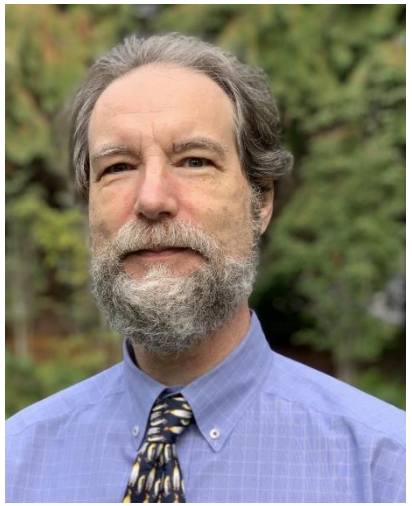
2024
September 15 - 20

Application of C++ in Cancer Modeling

- The speaker (Ruibo)
- University of Washington, department of applied mathematics
- Cancer Modeling Group: Evolution of cancer initiation
- Group leader:

Ivana Bozic
Associate Professor
UW AMATH
Fred Hutchinson Cancer Research center

Personal programming advisor:



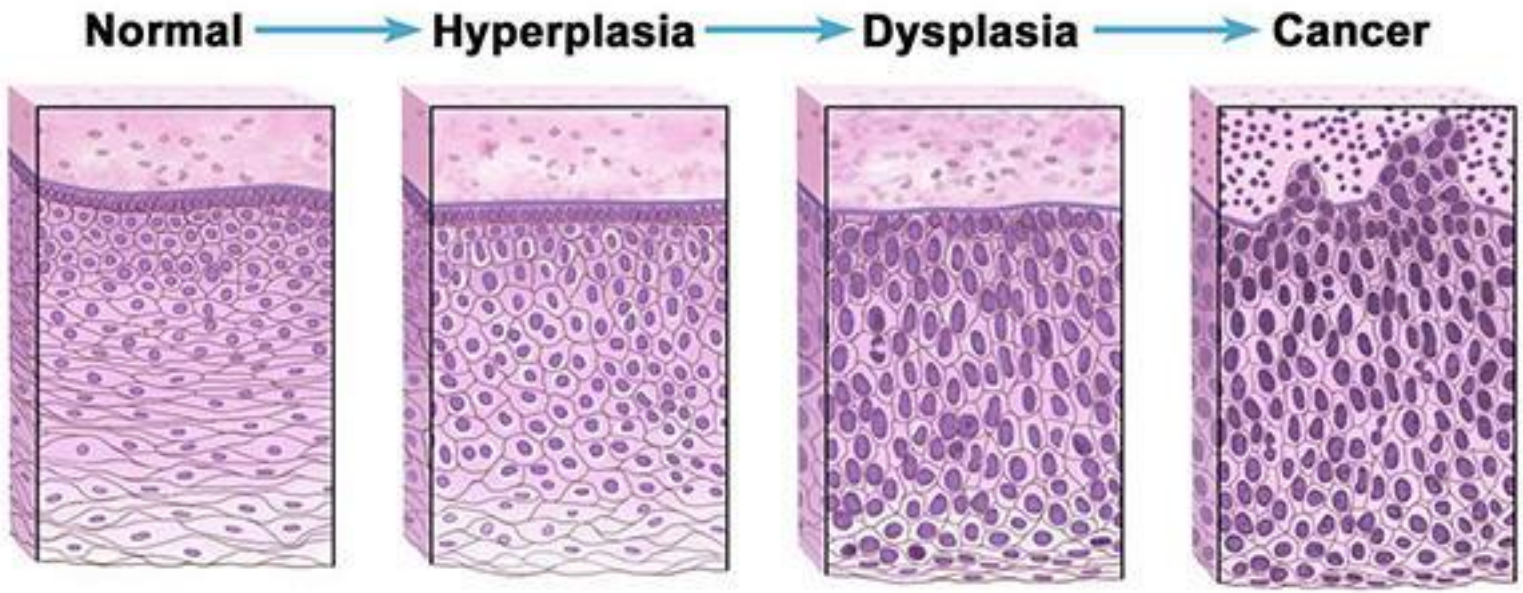
Daniel Hanson
CppCon, Student Program Chair
SG20

Outline

- Main Topic: use C++ to simulate the process of cancer initiation
- The mathematical model and simulation study
  - Generate a single tumor (A single step of evolution)
  - Generate multiple tumors (Tasked Based Concurrency)
  - Obtain statistical properties of the tumors (Parallel STL algorithms)
- Eigen (Array Class)
  - Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
- Modern C++:
  - <random>: Pseudo-random number generation
  - <future>: Task-Based Concurrency
  - <numeric>: Parallel versions of certain STL algorithms

Cancer and early detection

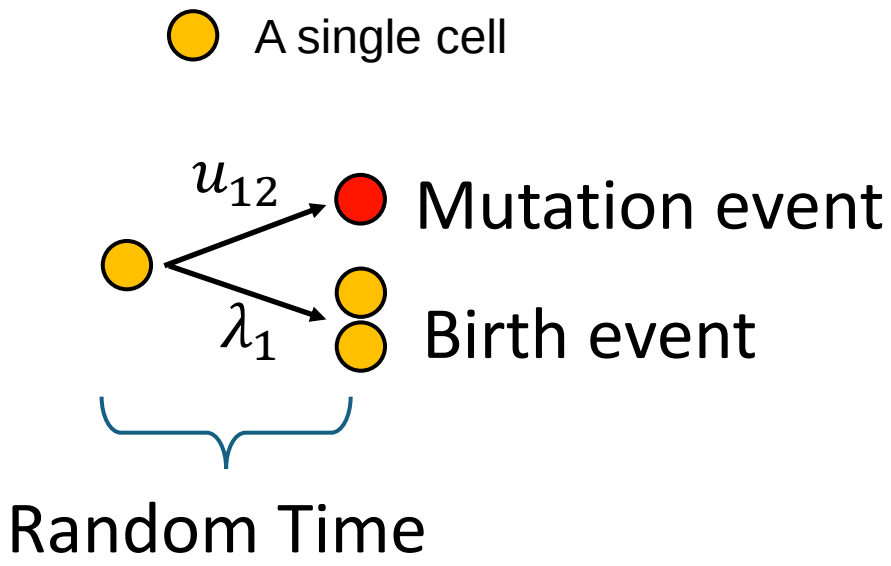
- Cancer is a disease characterized by the uncontrolled division of abnormal cells caused by changes in genes (DNA).
- When cancer is found early, it may be easier to treat or cure
- Mathematical models help understand the evolution quantitatively. (e.g. predict the window of opportunity for screening)



National Cancer Institute – What is Cancer

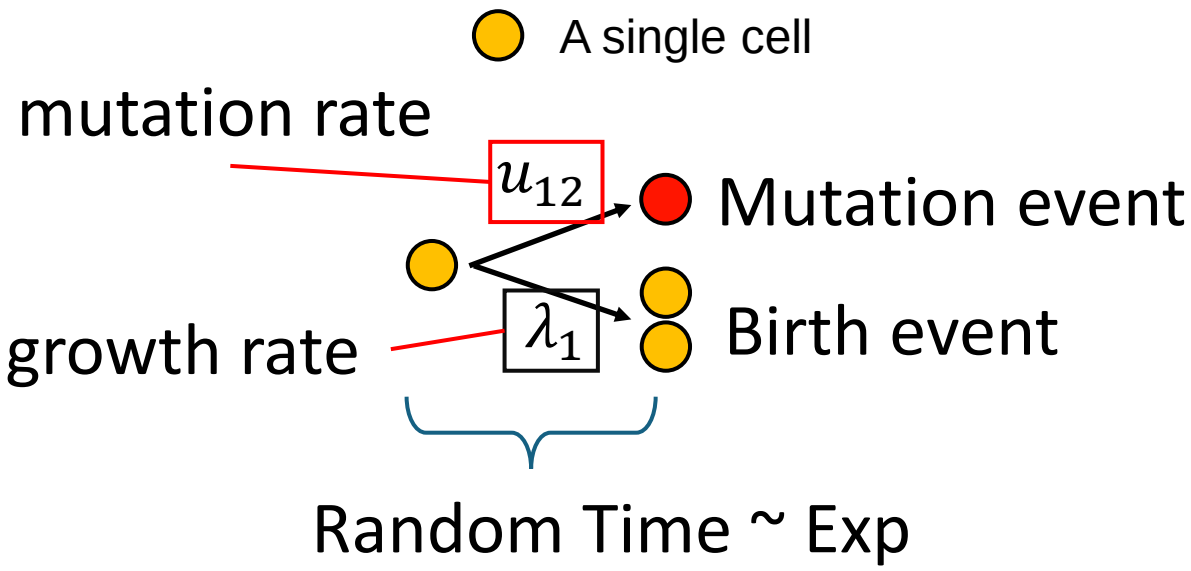
A computational model for cancer initiation

- Consider a bunch of ‘pseudo’ cells.
- Cells have different types that determines their behavior.
- Each cell can do two things: alter its type or divide into two cells of the same type.
- Formally, this model is a stochastic process (Markov process).



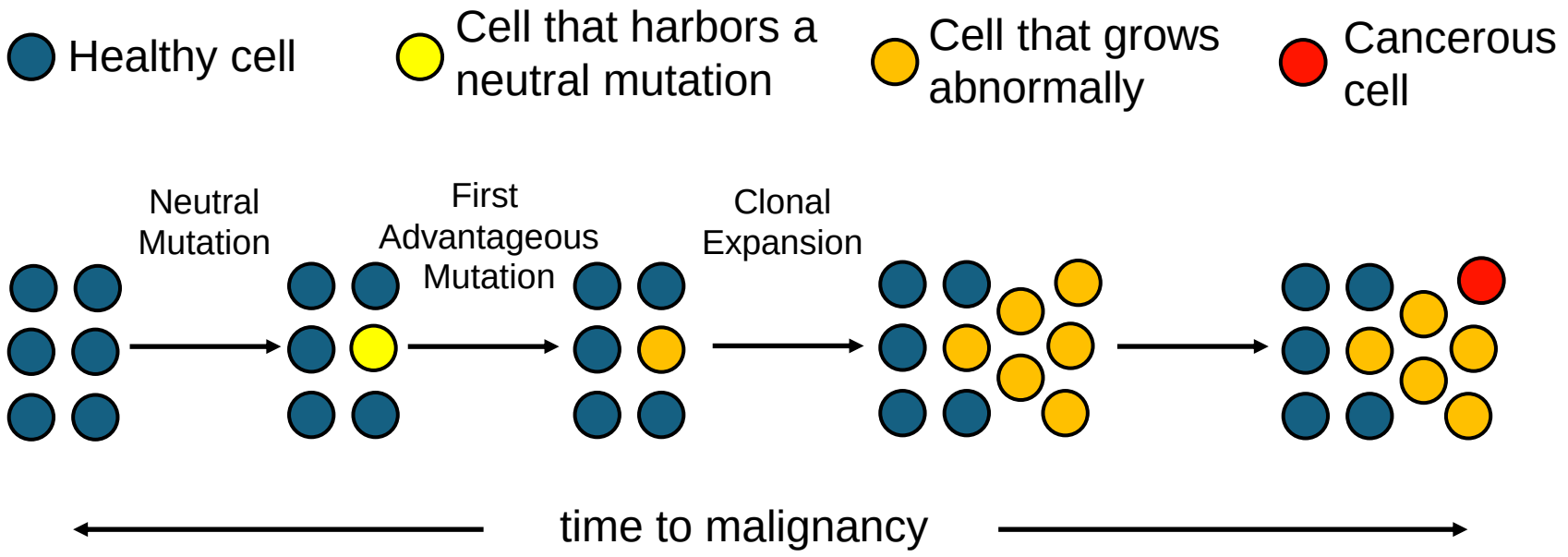
A computational model for cancer initiation

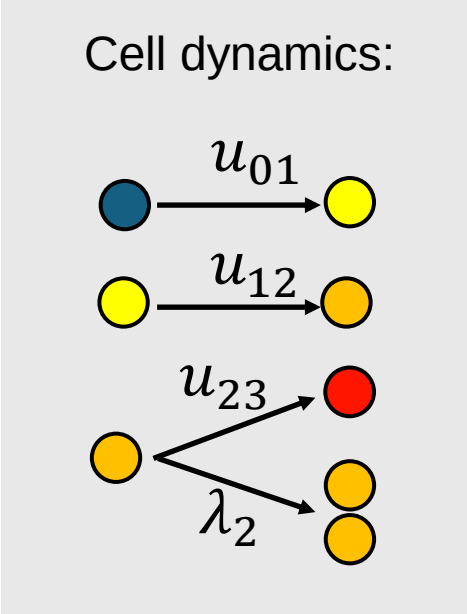
- Consider a bunch of ‘pseudo’ cells.
- Cells have different types that determines their behavior.
- Each cell can do two things: alter its type or divide into two cells of the same type.
- Formally, this model is a stochastic process (Markov process).




```
#include <random>
std::mt19937_64 rnd_generator;
std::exponential_distribution<> exp{rate};
double time = exp(rnd_generator);
```

A computational model for cancer initiation



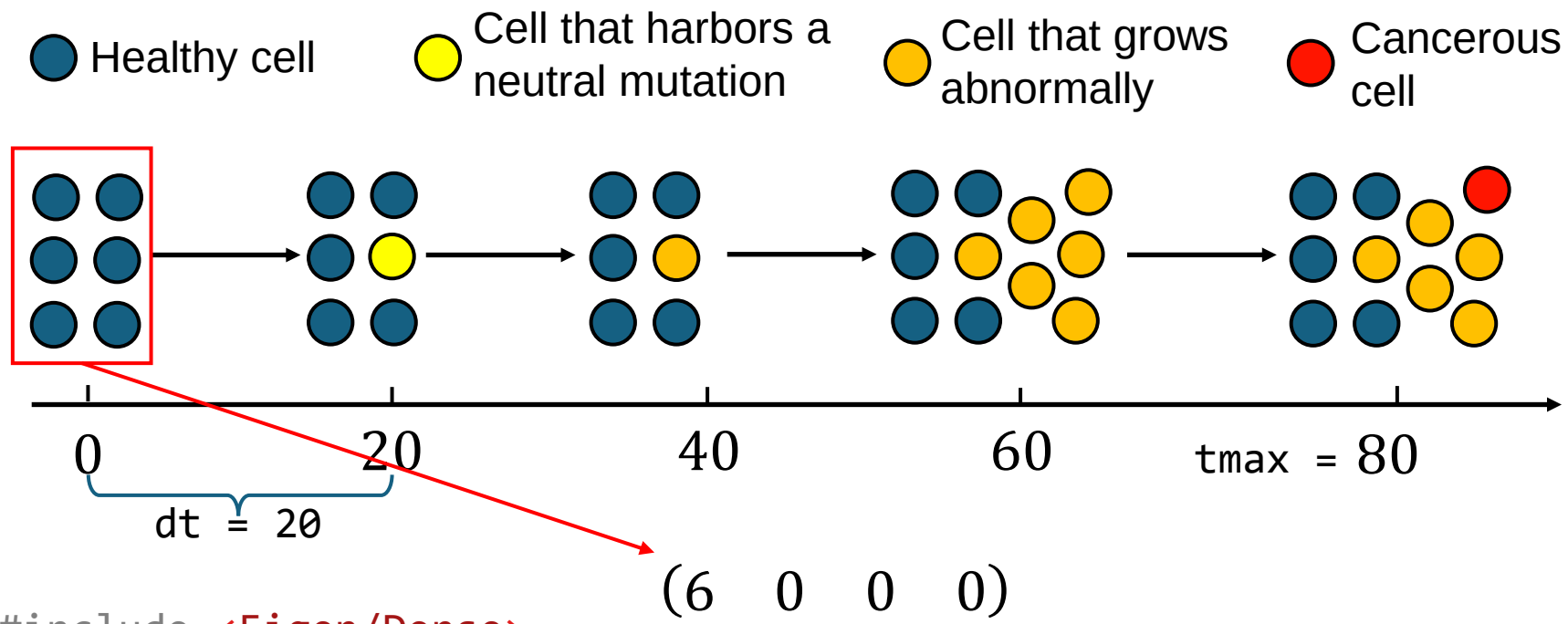


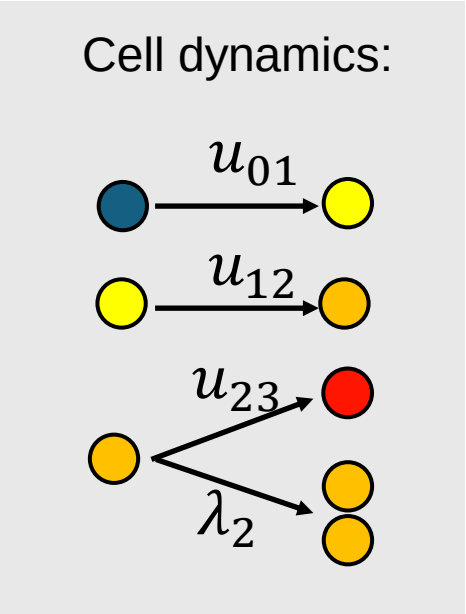
Input: Parameters.

Output: Population at desired times: $\mathrm{N}_i(t_1), \mathrm{N}_i(t_2), \ldots$

- Need many simulated tumors for understanding their statistical properties.
  - The waiting time to each type: τ_i = the first time such that $\mathrm{N}_i(t) > 0$
  - The distribution of population size for cell of each type.

Parameters for simulating a single tumor





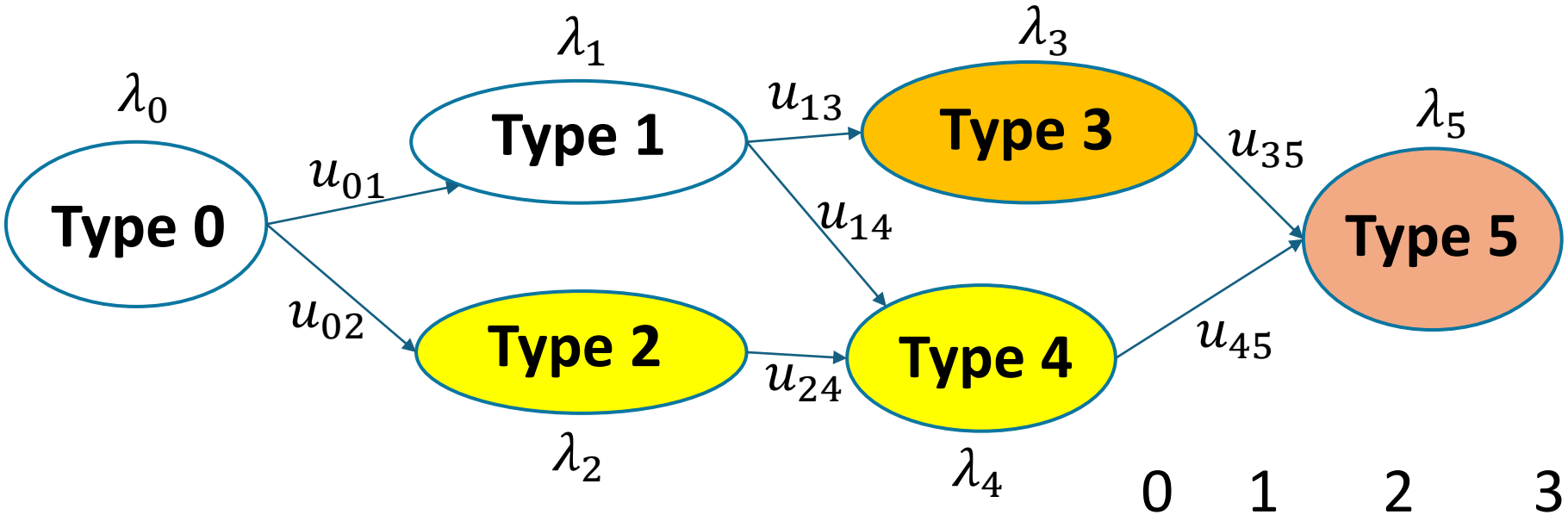

```
#include <Eigen/Dense>
class TumorGenerator {
Eigen::ArrayXd initial_population;
double tmax;
double dt; //...
```

The **Array** class template: Array<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>

```
typedef Eigen::Array<double, Eigen::Dynamic,1> Eigen::ArrayXd;
```

cppreference.com

Transition of types forms a network

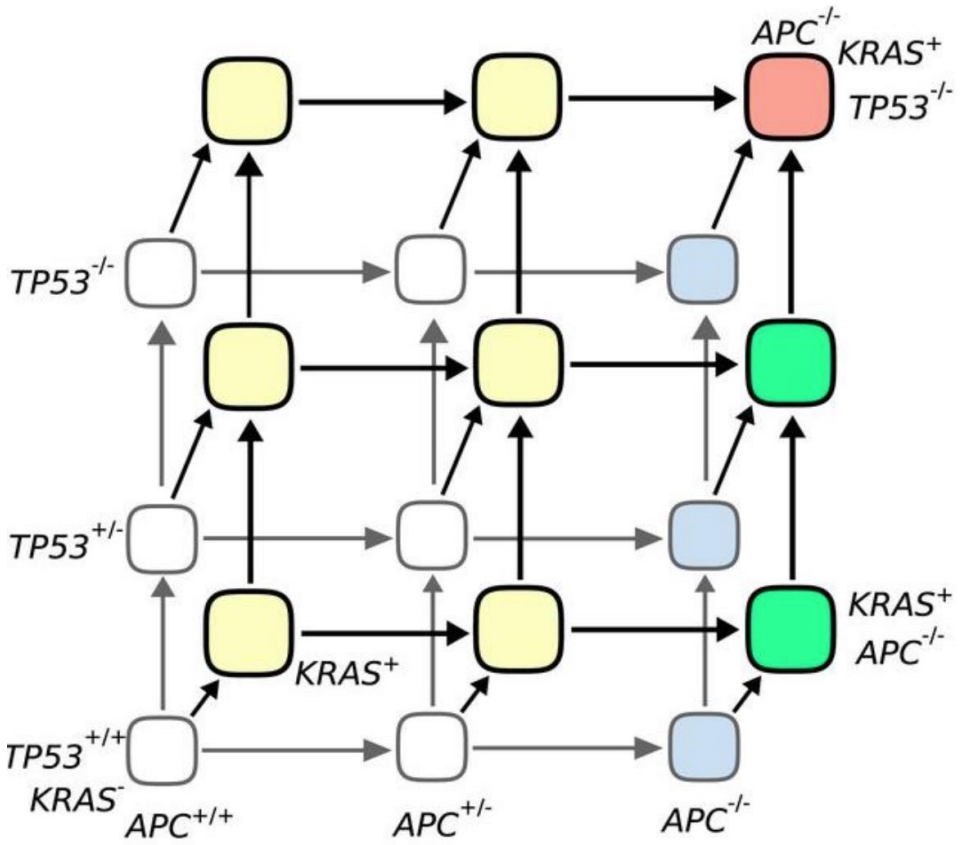


- A weighted graph for the transition of types
- Parameters form a weighted adjacent matrix.

```
int ntype;
Eigen::ArrayXXd transition_rates;
```

$$
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{pmatrix}
\lambda_0 & u_{01} & u_{02} & & & \\
 & \lambda_1 & & u_{13} & u_{14} & \\
 & & \lambda_2 & & u_{24} & \\
 & & & \lambda_3 & & u_{35} \\
 & & & & \lambda_4 & u_{45} \\
 & & & & & \lambda_5
\end{pmatrix}
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\end{array}
$$

Example: Colorectal Cancer with 5 mutations

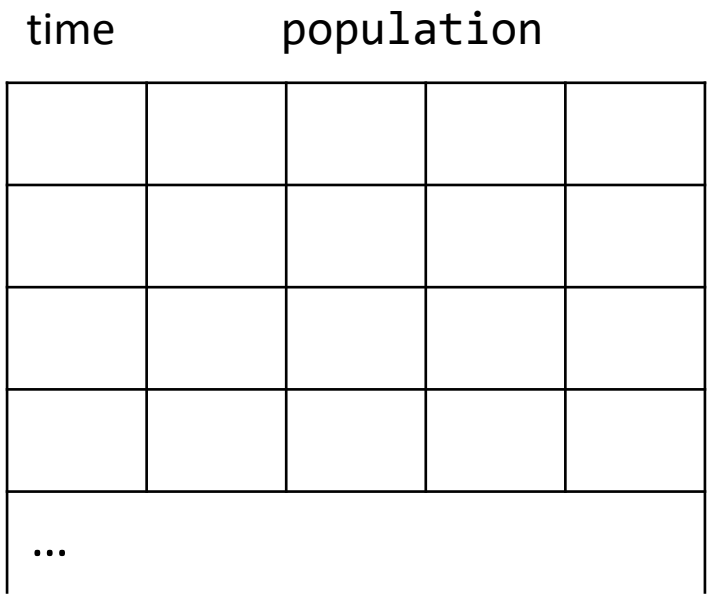


5 mutations can occur in any order.

Paterson et al. PNAS 2020

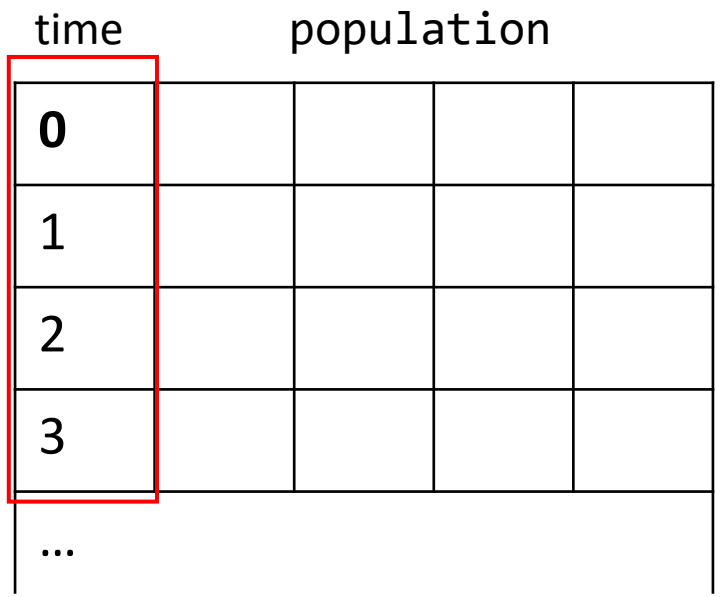
Track a single tumor by a 2d array

```
ArrayXXd TumorGenerator::single_tumor(unsigned seed)
{
    rnd_generator.seed(seed); // set the seed of the random number generator
    const int datalen = (int)(tmax / dt) + 1;
    Eigen::ArrayXXd time_population(datalen, ntype + 1);

    //...
```



Track a single tumor by a 2d array

```
ArrayXXd TumorGenerator::single_tumor(unsigned seed)
{
    rnd_generator.seed(seed); // set the seed of the random number generator
    const int datalen = (int)(tmax / dt) + 1;
    Eigen::ArrayXXd time_population(datalen, ntype + 1);

    const Eigen::VectorXd record_time = Eigen::VectorXd::LinSpaced(datalen, 0, tmax);
    time_population.block(0, 0, datalen, 1) = record_time;
    //...
```

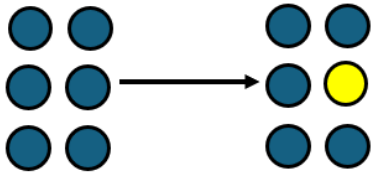


Block operation	Version constructing a dynamic-size block expression
Block of size (p,q), starting at (i,j)	`matrix.block(i,j,p,q);`

Track a single tumor by a 2d array

```
ArrayXXd TumorGenerator::single_tumor(unsigned seed)
{
    //...
    int data_index = 0;
    while(data_index < datalen){
        //...
        time_population.block(data_index, 1, 1, ntype) = population_old.transpose();
        //...
        evolve_step();
    }
    return time_population;
}
```

A single step of evolution



time	population			
0	6	0	0	0
1	5	1	0	0
2				
3				
…				

Block operation	Version constructing a dynamic-size block expression
Block of size (p,q), starting at (i,j)	`matrix.block(i,j,p,q);`

A single step of evolution

```
double TumorGenerator::lifespan(const Eigen::ArrayXXd& weights);
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd& weights);

void TumorGenerator::evolve_step()
{
    Eigen::ArrayXXd weights = transition_rates.colwise() * population; // 1
    time += lifespan(weights); // 2

    auto [decre_type, incre_type] = increment_type(weights); // 3
    population(incre_type)++;
    if (decre_type != -1) population(decre_type)--;
}
```

1. Compute the weights matrix.
2. Determine the arrival time by sampling an exponential distribution.
3. Normalize the weights to get the probabilities for possible events. Sample a random variable to decide which event happens.

Compute the weights matrix

```
void TumorGenerator::evolve_step()
{
    Eigen::ArrayXXd weights = transition_rates.colwise() * population; // 1

    //...
}
```

transition_rates

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

population

$$\begin{pmatrix} 0 \\ 0.2 \\ 2 \\ 0 \end{pmatrix} \qquad \begin{pmatrix} 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

weights

$$\begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

element-wise multiplication

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

$$\text{weights} \quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}$$

```
template< class InputIt >
discrete_distribution( InputIt first, InputIt last );
```

Constructs the distribution on integers with weights in the range [first, last)

cppreference.com

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

reshape() returns a ***view*** on the input expression.

0, 0, 0, 0, 0.5, 0, 0, 0, 0, 0, 2, 0, 0, … 0

$$\text{weights} \quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}$$

```
template< class InputIt >
discrete_distribution( InputIt first, InputIt last );
```

Constructs the distribution on integers with weights in the range [first, last)

cppreference.com

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

0, 0, 0, 0; 0.5, 0, 0, 0; 0, 0, 2, 0; 0, … 0

$$\text{weights}\quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}$$

Eigen: STL iterators are intrinsically designed to iterate over 1D structures. This is why begin()/end() methods are disabled for 2D expressions.

```
template< class InputIt >
discrete_distribution( InputIt first, InputIt last );
```

Constructs the distribution on integers with weights in the range [first, last)

cppreference.com

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

weights.reshaped()	0,	0,	0,	0;	0.5,	0,	0,	0;	0,	0,	2,	0,	…	0	
Integers:	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	…,	15	
Probabilities:	0,	0,	0,	0,	1/6,	0,	0,	0,	0,	0,	2/3,	0,	…,	0	

$$\text{weights} \quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}$$

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

Select an integer according to the probabilities

weights.reshaped() 0, 0, 0, 0; 0.5, 0, 0, 0; 0, 0, 2, 0, … 0

Integers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, …,15

Probabilities: 0, 0, 0, 0, 1/6, 0, 0, 0, 0, 0, 2/3, 0, …,0

$$\text{weights} \quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}$$

Sample a position in a matrix randomly

```
std::pair<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    //...
}
```

Select an integer according to the probabilities

weights.reshaped() 0, 0, 0, 0; 0.5, 0, 0, 0; 0, 0, 2, 0, … 0

Integers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, …,15

Probabilities: 0, 0, 0, 0, 1/6, 0, 0, 0, 0, 0, 2/3, 0, …,0

$$\text{weights} \quad \begin{array}{c} \begin{array}{cccc} 0 & 1 & \boxed{2} & 3 \end{array} \\ \begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} 0 \\ 1 \\ \boxed{2} \\ 3 \end{array}$$

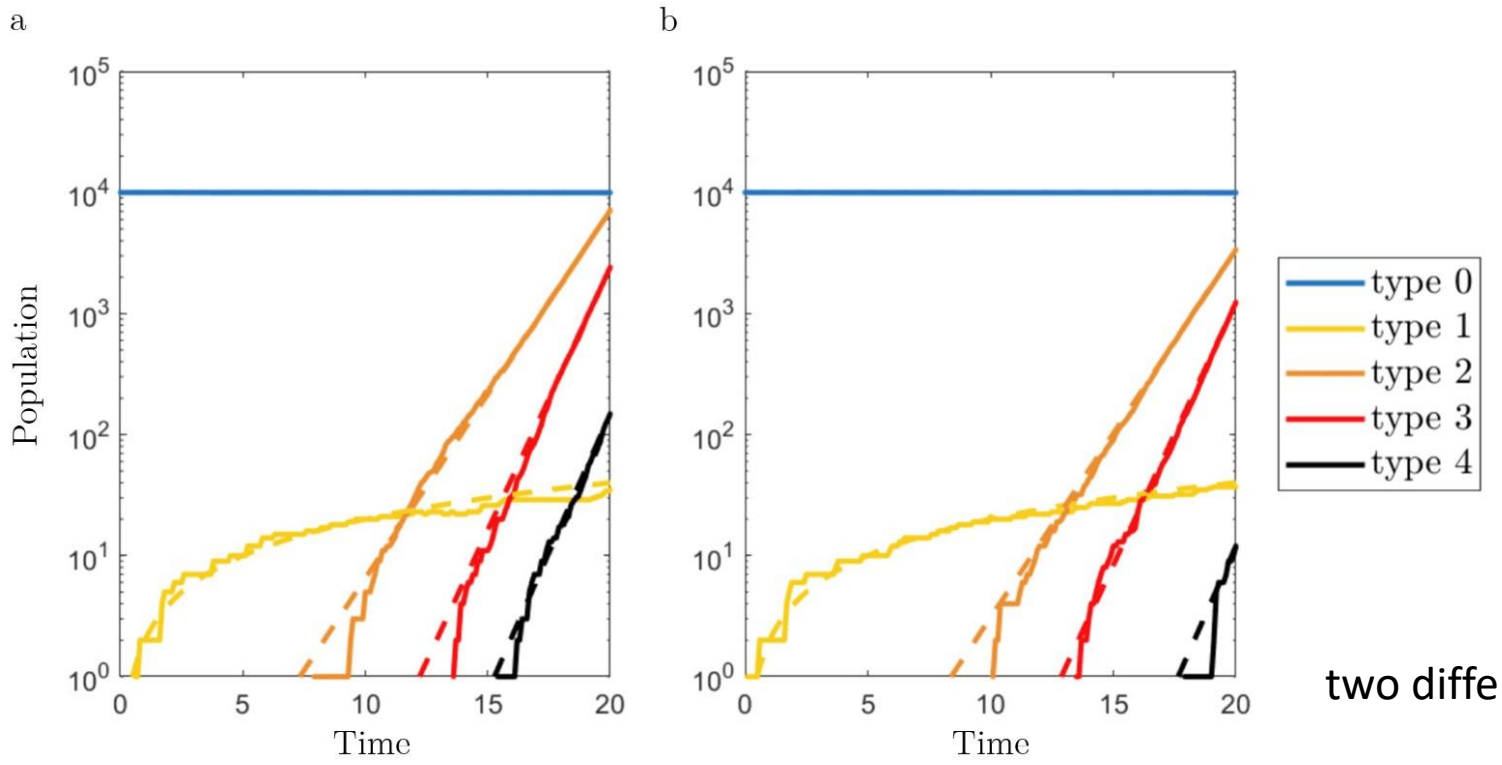
10 = 2 x 4 + 2 → 10 == weights(2,2)

Task-based concurrency

- Each tumor is a parallel task:

```
std::vector<std::future<Eigen::ArrayXXd>>
```

- For each future object, call get() to obtain a 2d array.



two different tumors

Task-based concurrency

```
Eigen::ArrayXXd tumor_par(const unsigned& seed, const double& tmax, const double& dt,
    const Eigen::ArrayXd& initial_population, const Eigen::ArrayXXd& transition_rates);

int main(){
    //...
    std::vector<std::future<Eigen::ArrayXXd>> sample_tumor_par;
    for (auto k : seeds)
        sample_tumor_par.emplace_back(std::async(std::launch::async, tumor_par, k, tmax,
    dt, initial_population, transition_rates));

    // store results (a vector of future Eigen arrays)
    std::vector<Eigen::ArrayXXd> population_result;
    population_result.reserve(sample_tumor_par.size());

    // get results (a vector of future Eigen arrays)
    std::ranges::transform(sample_tumor_par, std::back_inserter(population_result),
        [](std::future<Eigen::ArrayXXd> &fut){
            return fut.get();
        });
```

Task-based concurrency

```
Eigen::ArrayXXd tumor_par(const unsigned& seed, const double& tmax, const double& dt,
    const Eigen::ArrayXd& initial_population, const Eigen::ArrayXXd& transition_rates);

int main(){
    //...
    std::vector<std::future<Eigen::ArrayXXd>> sample_tumor_par;
    for (auto k : seeds)
        sample_tumor_par.emplace_back(std::async(std::launch::async, tumor_par, k, tmax,
    dt, initial_population, transition_rates));

    // store results (a vector of future Eigen arrays)
    std::vector<Eigen::ArrayXXd> population_result;
    population_result.reserve(sample_tumor_par.size());

    // get results (a vector of future Eigen arrays)
    std::ranges::transform(sample_tumor_par, std::back_inserter(population_result),
        [](std::future<Eigen::ArrayXXd> &fut){
            return fut.get();
        });
```

Task-based concurrency

```
Eigen::ArrayXXd tumor_par(const unsigned& seed, const double& tmax, const double& dt,
    const Eigen::ArrayXd& initial_population, const Eigen::ArrayXXd& transition_rates);

int main(){
    //...
    std::vector<std::future<Eigen::ArrayXXd>> sample_tumor_par;
    for (auto k : seeds)
        sample_tumor_par.emplace_back(std::async(std::launch::async, tumor_par, k, tmax,
    dt, initial_population, transition_rates));

    // store results (a vector of future Eigen arrays)
    std::vector<Eigen::ArrayXXd> population_result;
    population_result.reserve(sample_tumor_par.size());

    // get results (a vector of future Eigen arrays)
    std::ranges::transform(sample_tumor_par, std::back_inserter(population_result),
        [](std::future<Eigen::ArrayXXd> &fut){
            return fut.get();
        });
```

Launch Policy

Task-based concurrency

```
Eigen::ArrayXXd tumor_par(const unsigned& seed, const double& tmax, const double& dt,
    const Eigen::ArrayXd& initial_population, const Eigen::ArrayXXd& transition_rates);

int main(){
    //...
    std::vector<std::future<Eigen::ArrayXXd>> sample_tumor_par;
    for (auto k : seeds)
        sample_tumor_par.emplace_back(std::async(std::launch::async, tumor_par, k, tmax,
    dt, initial_population, transition_rates));

    // store results (a vector of future Eigen arrays)
    std::vector<Eigen::ArrayXXd> population_result;
    population_result.reserve(sample_tumor_par.size());

    // get results (a vector of future Eigen arrays)
    std::ranges::transform(sample_tumor_par, std::back_inserter(population_result),
        [](std::future<Eigen::ArrayXXd> &fut){
            return fut.get();
        });
```

Launch Policy

Each tumor has a unique seed.

Parameters shared by all the tumors

Task-based concurrency

```
Eigen::ArrayXXd tumor_par(const unsigned& seed, const double& tmax, const double& dt,
    const Eigen::ArrayXd& initial_population, const Eigen::ArrayXXd& transition_rates);

int main(){
    ...
    std::vector<std::future<Eigen::ArrayXXd>> sample_tumor_par;
    for (auto k : seeds)
        sample_tumor_par.emplace_back(std::async(std::launch::async, tumor_par, k, tmax,
    dt, initial_population, transition_rates));

    // store results (a vector of future Eigen arrays)
    std::vector<Eigen::ArrayXXd> population_result;
    population_result.reserve(sample_tumor_par.size());

    // get results (a vector of future Eigen arrays)
    std::ranges::transform(sample_tumor_par, std::back_inserter(population_result),
        [](std::future<Eigen::ArrayXXd> &fut){
            return fut.get();
        });
```

Launch Policy

Each tumor has a unique seed.

Parameters shared by all the tumors

call .get() to obtain population arrays.

Parallel STL algorithm (average population)

```
// compute average population
    Eigen::ArrayXXd initial_array = Eigen::ArrayXXd::Zero(population_result[0].rows(),
population_result[0].cols());

    Eigen::ArrayXXd average_population = 1.0 / runs *
        std::reduce(std::execution::par, population_result.begin(), population_result.end(),
initial_array);
```

The generalized sum of init and the elements of [first, last) over std::plus<>().

```
template< class ExecutionPolicy, class ForwardIt, class T >
T reduce( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init );
```

cppreference.com

Parallel STL algorithm (average population)

```
// compute average population
    Eigen::ArrayXXd initial_array = Eigen::ArrayXXd::Zero(population_result[0].rows(),
population_result[0].cols());

    Eigen::ArrayXXd average_population = 1.0 / runs *
        std::reduce(std::execution::par, population_result.begin(), population_result.end(),
initial_array);
```

Specify the execution policy of parallel algorithms

The generalized sum of init and the elements of [first, last) over std::plus<>().

```
template< class ExecutionPolicy, class ForwardIt, class T >
T reduce( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init );
```

cppreference.com

Parallel STL algorithm (waiting time)

```
// computing waiting time distribution
Eigen::ArrayXXd waitingtime_dist = 1.0 / runs *
    std::transform_reduce(std::execution::par,population_result.begin(),
    population_result.end(),
    initial_array,
    std::plus<Eigen::ArrayXXd>(), // binary operation
    [](Eigen::ArrayXXd& array)->Eigen::ArrayXXd {return (array > 0).cast<double>();});
```

unary operation

Applies transform to each element in the range [first, last) and std::reduces the results along with the initial value init over reduce.

Performance

Generate a required number of tumors:

ntype	Initial cells	tmax	Number of tumors	Time-Serial (sec)	Time-Parallel (sec)
3	10^5	85	10^3	0.4914	0.0223
3	10^5	85	10^5	42.30	1.86
6	10^8	70	1	68.24	69.29
6	10^8	70	10	733.6	137.6

Use a STL algorithm to get waiting time distributions:

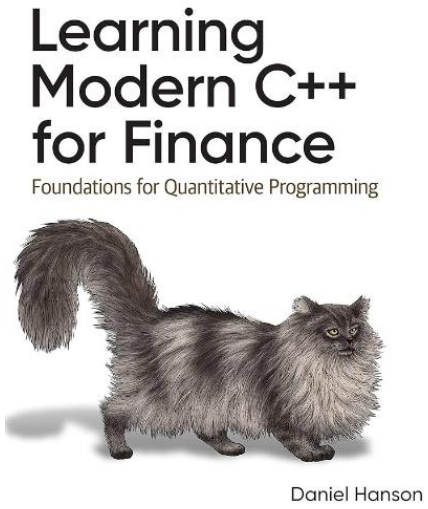
ntype	Initial cells	tmax	Number of tumors	Time-Serial (sec)	Time-Parallel (sec)
3	10^5	85	10^3	0.0175	0.0015
3	10^5	85	10^5	1.8663	0.0897

std::chrono

Concluding remarks

- Documentation of Eigen is pretty good.
- std::future/std::async are easy to pick up.
- Computational Biology is a rapidly growth field. (sequencing data)
- Major reference:

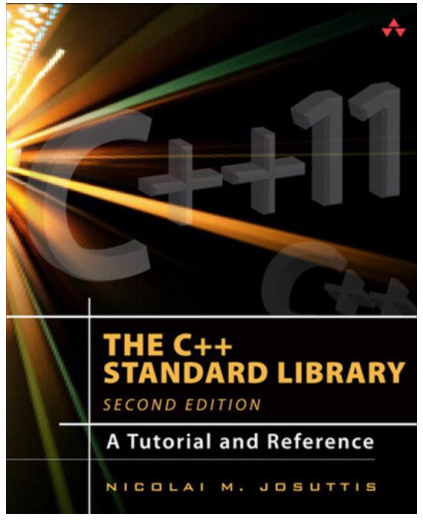
Daniel Hanson, Learning Modern C++ for Finance, O'Reilly

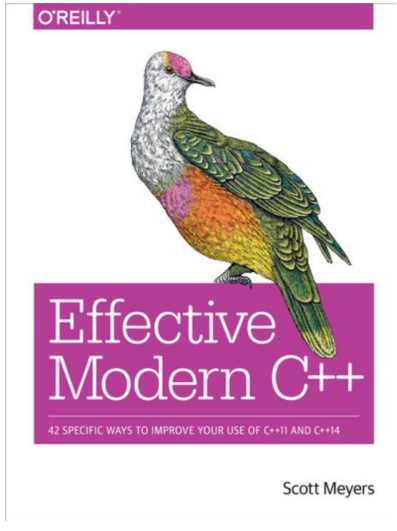


Thank you for attending!

- Sample Code available on Github: https://github.com/RuiboZhang98/CppGillespie
- Contact information: rzhang98 (at) uw.edu

Additional References:
C++ Standard Library, The: A Tutorial and Reference, 2nd Edition, Addison Wesley
Effective Modern C++, Scott Meyers, O'Reilly
cppreference.com



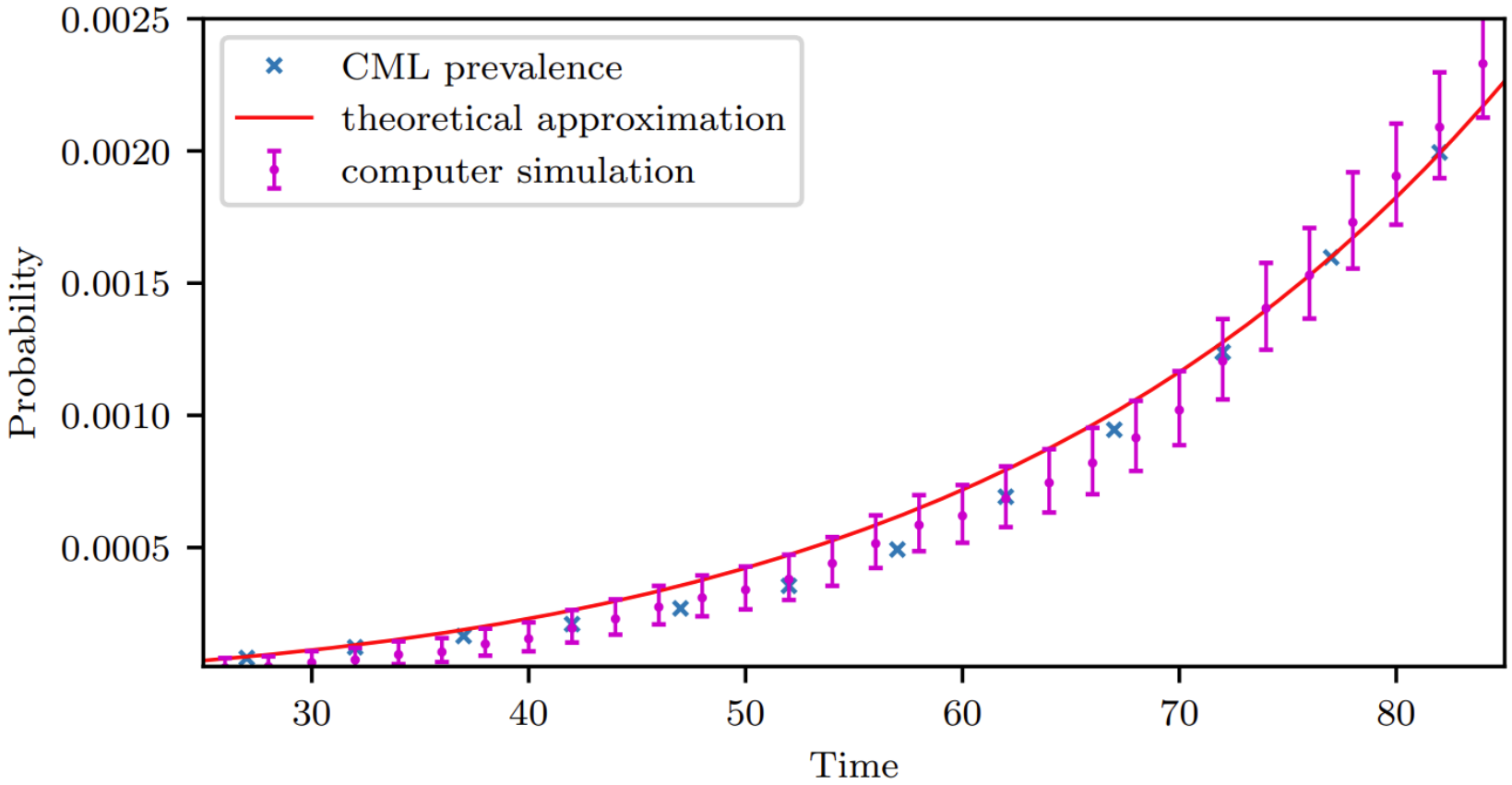


Appendices

Chronic Myeloid Leukemia

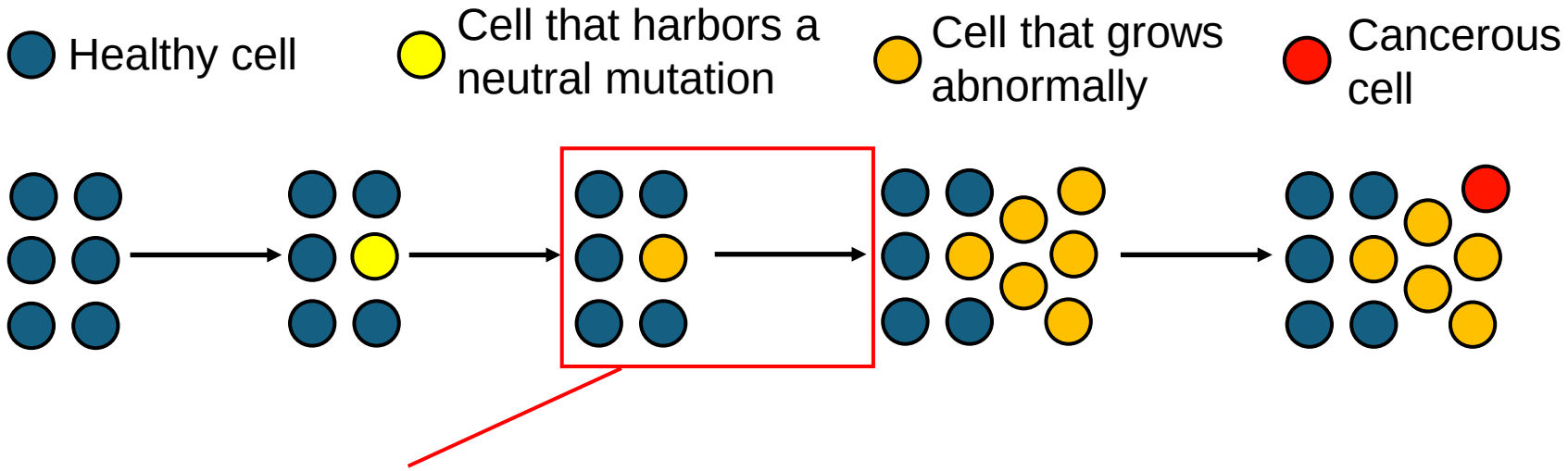
- One genetic alteration (BCR-ABL) can cause CML.
- A simple model with three types:

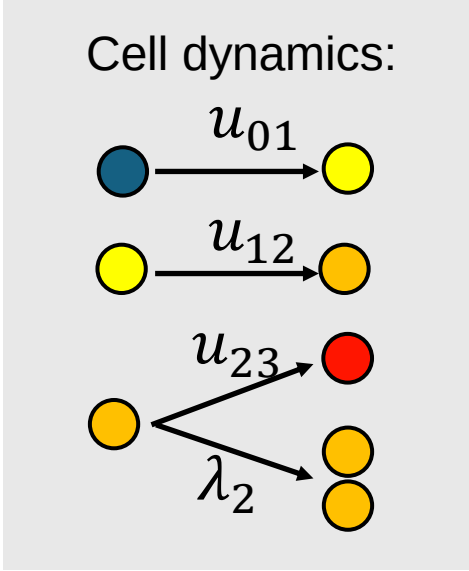
Heathy cells –> cancerous cells –> detectable cancer



CML prevalence data points come from the SEER Incidence Data, National Cancer institute.
SEER (Surveillance, Epidemiology, and End Results Program)

A single step of evolution





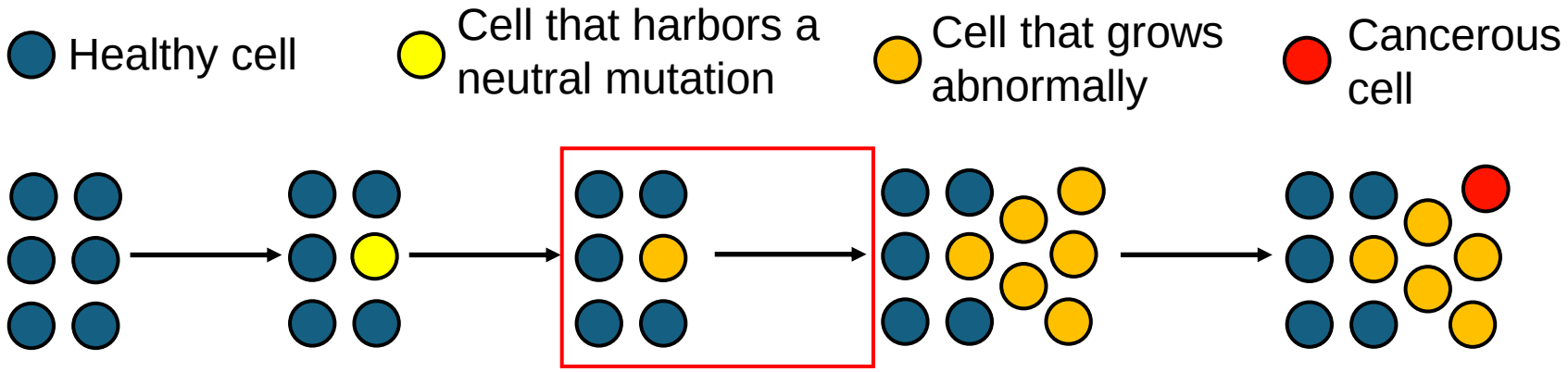
population $\begin{pmatrix} 5 & 0 & 1 & 0 \end{pmatrix}$

transition_rates

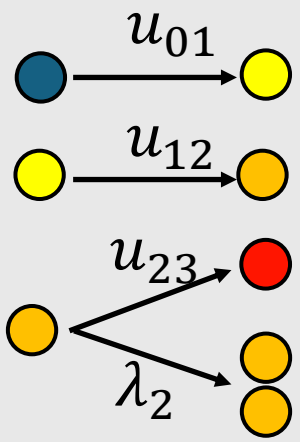
$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

```
// tumor population at a certain time
Eigen::ArrayXd population;
double time;
```

A single step of evolution



Cell dynamics:

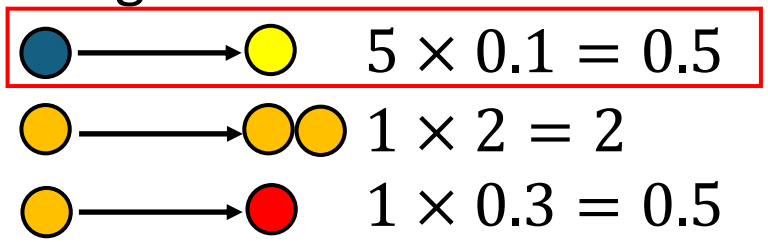


population $\begin{pmatrix} \boxed{5} & 0 & 1 & 0 \end{pmatrix}$

transition_rates

$$\begin{pmatrix} 0 & \boxed{0.1} & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Weights for the next event:



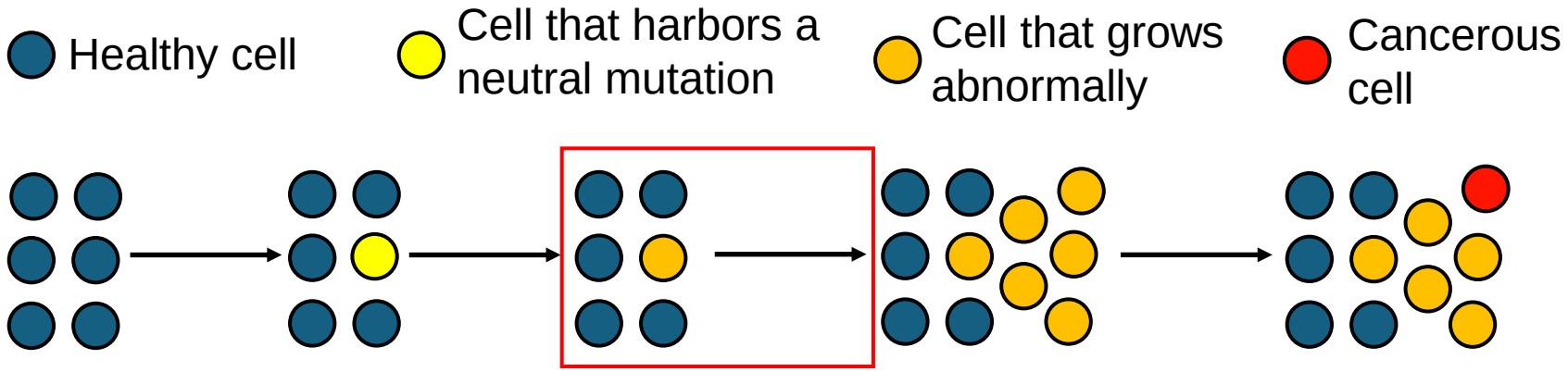
$5 \times 0.1 = 0.5$

$1 \times 2 = 2$

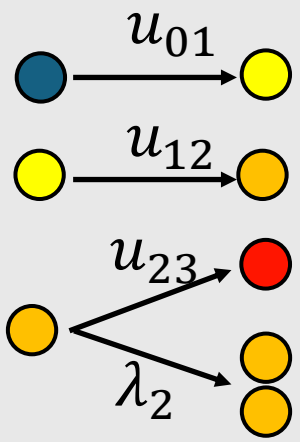
$1 \times 0.3 = 0.5$

Sum: $0.5 + 2 + 0.5 = 3$

A single step of evolution



Cell dynamics:



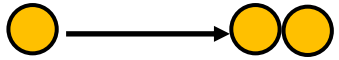
`population` $\begin{pmatrix} 5 & 0 & \boxed{1} & 0 \end{pmatrix}$

`transition_rates`

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & \boxed{2} & \boxed{0.5} \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Weights for the next event:

$5 \times 0.1 = 0.5$

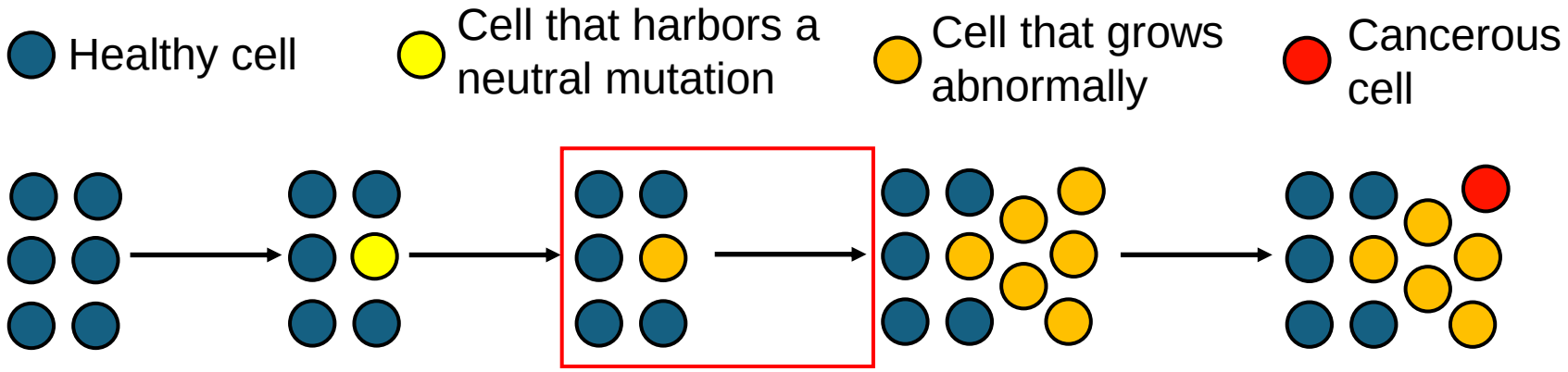


$1 \times 2 = 2$

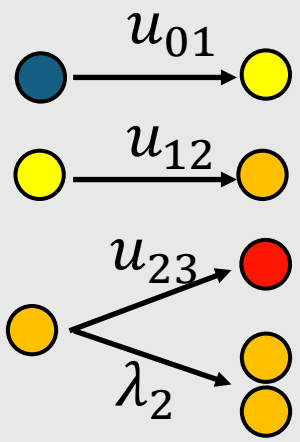
$1 \times 0.3 = 0.5$

Sum: $0.5 + 2 + 0.5 = 3$

A single step of evolution



Cell dynamics:

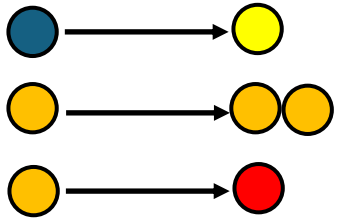


`population` $\begin{pmatrix} 5 & 0 & 1 & 0 \end{pmatrix}$

`transition_rates`

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Weights for the next event:



$5 \times 0.1 = 0.5$

$1 \times 2 = 2$

$1 \times 0.3 = 0.5$

Sum: $0.5 + 2 + 0.5 = 3$

Probabilities:

0.5/3 = 1/6

2/3 = 2/3

0.5/3 = 1/6

Time for the next event follows an exponential distribution $\sim$ Exp(3)

Appendix A: Compute the weight matrix

```
void TumorGenerator::evolve_step()
{
1.  Eigen::ArrayXXd weights = transition_rates.colwise() * population;

    ...
}
```

transition_rates

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

population

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

weights

$$\begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

element-wise multiplication

40

Compute the weight matrix

```
void TumorGenerator::evolve_step()
{
1.  Eigen::ArrayXXd weights = transition_rates.colwise() * population;

    ...
}
```

transition_rates

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

population

$$\begin{pmatrix} 0.1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

weights

$$\begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

element-wise multiplication

41

Compute the weight matrix

```
void TumorGenerator::evolve_step()
{
1.  Eigen::ArrayXXd weights = transition_rates.colwise() * population;

    ...
}
```

transition_rates

$$\begin{pmatrix} 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

population

$$\begin{pmatrix} 0 \\ 0.2 \\ 2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

weights

$$\begin{pmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0.5 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

element-wise multiplication

42

Compute the weight matrix

```
void TumorGenerator::evolve_step()
{
1.  Eigen::ArrayXXd weights = transition_rates.colwise() * population;

    ...
}
```

transition_rates

$$\begin{pmatrix} 0 & 0.1 & 0 & \boxed{0} \\ 0 & 0 & 0.2 & \boxed{0} \\ 0 & 0 & 2 & \boxed{0.5} \\ 0 & 0 & 0 & \boxed{3} \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 0.5 \\ 3 \end{pmatrix}$$

population

$$\begin{pmatrix} 5 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

weights

$$\rightarrow \begin{pmatrix} 0 & 0.5 & 0 & \boxed{0} \\ 0 & 0 & 0 & \boxed{0} \\ 0 & 0 & 2 & \boxed{0.5} \\ 0 & 0 & 0 & \boxed{0} \end{pmatrix}$$

element-wise multiplication

43

Appendix B: Time evolution

```
void TumorGenerator::evolve_step()
{
    Eigen::ArrayXXd weights = transition_rates.colwise() * population;
2.  time += lifespan(weights);

    ...
}

double TumorGenerator::lifespan(const Eigen::ArrayXXd& weights){
    // The function computes the waiting time for next population change.
    double lambda = weights.sum();
    std::exponential_distribution<> exp{ lambda };
    return exp(rnd_generator);
}
```


44

Sample an entry in an array randomly

```
void TumorGenerator::evolve_step()
{
    ...
3.  auto [decre_type, incre_type] = increment_type(weights);
    population(incre_type)++;
    if (decre_type != -1) population(decre_type)--;
}
```

```
std::tuple<int,int> TumorGenerator::increment_type(const Eigen::ArrayXXd &weights)
{
    int ntype = weights.rows();
    std::discrete_distribution<> d(weights.reshaped().begin(),weights.reshaped().end());
    int random_entry = d(rnd_generator);

    int c = random_entry / ntype;
    int r = random_entry - c * ntype;
    int decrement = -1;
    if(r != c) decrement = r;

    return {decrement, c};
}
```


45

```
Eigen::ArrayXXd TumorGenerator::single_tumor(unsigned seed)
{
    rnd_generator.seed(seed); // set the seed of the random number generator
    const int datalen = (int)(tmax / dt) + 1;  // length of recorded data
    const Eigen::VectorXd record_time = Eigen::VectorXd::LinSpaced(datalen, 0, tmax); //
record population at these populations
    Eigen::ArrayXXd time_population = Eigen::ArrayXXd::Constant(datalen, ntype + 1, -1.0); //
initiate the data table
    time_population.block(0, 0, datalen, 1) = record_time; // flush the first column with the
record times
    // initialize a tumor
    initiation();
    Eigen::ArrayXd population_old = population;
    // data_index indicates the current row of the result
    int data_index = 0;
    while(data_index < datalen){
        for ( ; (data_index < datalen) && (record_time(data_index) <= time) ; data_index++){
                time_population.block(data_index, 1, 1, ntype) = population_old.transpose();
        }
        population_old = population;
        evolve_step();
    }
    return time_population;
}
```

Initial value in reduce is necessary

```
reduce(first, last, typename std::iterator_traits<InputIt>::value_type{})
```

value_type: It is a nested type of std::iterator_traits that represents the type of the elements that the iterator points to.

{}: This is the syntax for value initialization, which will call the default constructor of the value_type.

Eigen::ArrayXXd: A default constructor is always available, never performs any dynamic memory allocation, and never initializes the matrix coefficients.

Cppreference.com,