Documentation for the Cancerdynamics Simulator

Boris Prochnau

February 26, 2015

Contents

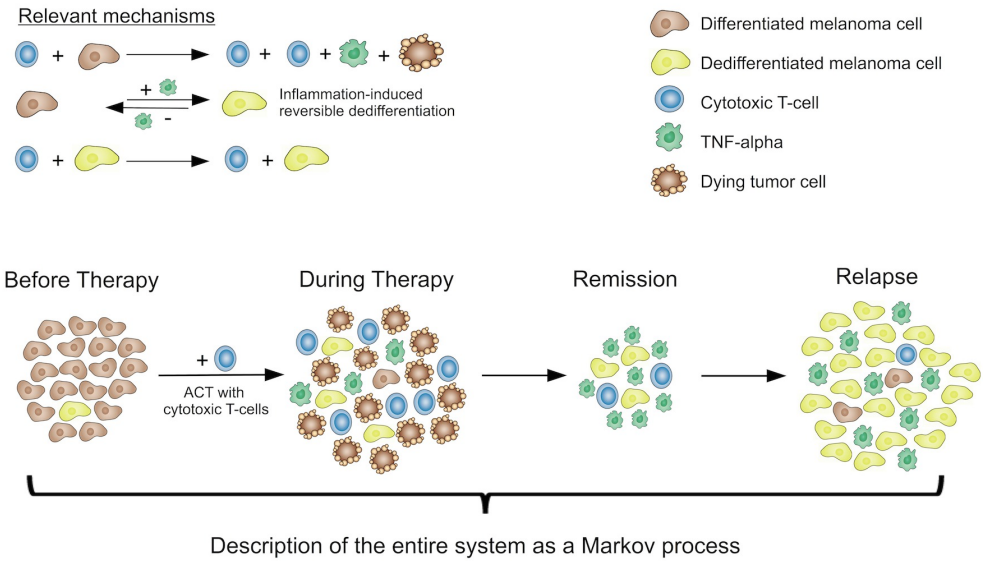

Figure 1: rescaling Tumor

General Information

First I would like to say that I tried to make the code as readable as possible with the knowledge and experience I had at this time. Of curse both grew since than and I would make some things different. Still this documentation is pretty tight coupled to the code and uses the assumption that many variables, function names and some computations should be self-explaining once you know what to look at.
Also it seems long ago when I wrote this simulation and I lost my records of notes. That's why I had to reassemble my understanding of the algorithm by explaining the code within the simulation. Some things are still not totally clear.

1.1 Work-flow and levels of abstraction

The simulation works with an algorithm that can be applied repeatedly to changing Data. Every application of the algorithm will modify the data (the given population). So the basically we see the goal of creating a process by displaying the stages of the data after one application of the algorithm. That's why the function containing the algorithm is called 'makeEvolutionStep' (or in older versions just EvolutionStep).
The work-flow was separated into different abstraction levels. They can be seen in a depth-diagram (fig. 2).

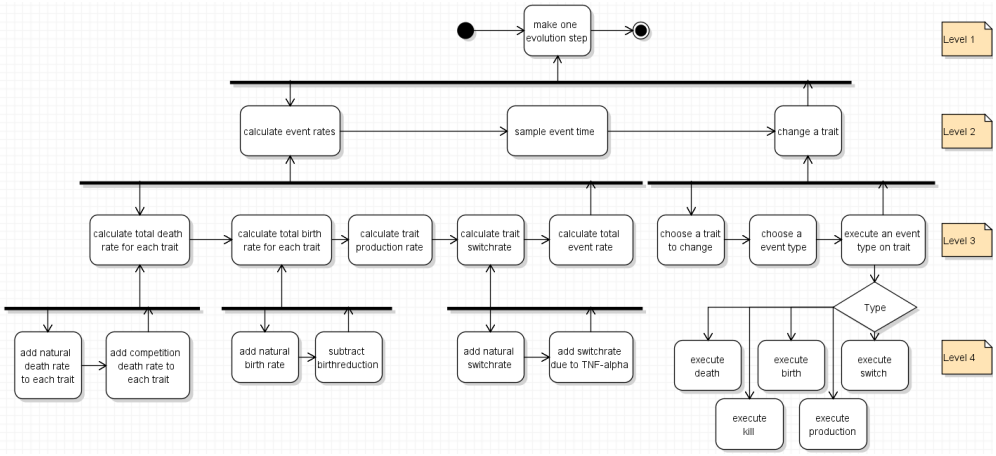

Figure 2: Depth-diagram

1.2 Expressions

The population consists of members that are grouped by their traits. Each trait represents its own behaviour and doesn't vary between members of this trait. Thats why this algorithm (and the one from my bachelor thesis) works with traits (this means the data structure doesn't know about t-cells or melanoma cells, they are all traits). The algorithm expects a specific pattern of traits with specific behaviour. It is specified in the 'Musterinstanz.txt' and 'Testinstanz_kommentiert'. The population is grouped into:

1. TNF-α
2. t-cells
3. melanoma cells (phenotypes $\times$ genotypes)

where every group has their own behaviour within the algorithm, i.e. the t-cells have no possibility to switch like the melanoma cells (see fig. 1).

Abstraction level one

In this and the following chapters I will try to explain the behaviour for each responsibility in the different abstraction levels (see fig. 2).

2.1 makeEvolutionStep()

The top level contains only 'makeEvolutionStep()' with the primary functionalities:

1. '**calculateTotalEventRates**()': This function calculates all rates (like dying, switch etc.), which together form a total rate for the first event.

2. '**samlpleEventTime**()': The 'total event rate' from point 1 is the parameter for a exponentially distributed random variable to sample the time for the next occasion.

3. '**changePopulation**()': Finally the Population will be changed. The function uses the rates that were calculated in the first function to decide who will be affected and what happens to him.

After this evolutionary step of the population, the point set will be updated with the change made to the population. After the desired steps, point set will be plotted in the graphical user interface.
The algorithmic code should be self-explaining:

```
1  void PopulationManager::makeEvolutionStep()
   {
3      calculateTotalEventRates();
       sampleEventTime();
5      changePopulation();
   }
7
```

Abstraction level two

3.1 calculateTotalEventRates()

This function requests the population to calculate all interesting rates. So it calls:

1. '**calculateTraitDeathRates**()': Provides death rates for each trait. They contain natural and competition deaths.

2. '**calculateTraitBirthRates**()': Provides birth rates for each trait. They contain natural birth rates and the reduction.

3. '**calculateTraitProductionRates**()': Provides production rates for each t-cell.

4. '**calculateTraitSwitchRates**()': Provides switch rates for each melanoma cell. They contain natural and competition switches.

5. '**calculateTraitKillRates**()': Provides kill rates for each t-cell.

6. '**calculateTraitRates**()': Provides overall rates for each trait. This is the rate that something (all rates together) happens to this trait.

7. And finally the new **totalEventRate** made from all trait rates.

The function was separated to split the responsibilities of the calculation. First it calculates all trait specific rates with 'calculateTotalTraitRates()' and than the total event rate for the first occasion in the for loop. It uses a container named 'traits' that contains all specific traits with their properties. Each trait object in the container specifies his individual population size and rates.

```
void PopulationManager::calculateTotalEventRates()
{
    calculateTotalTraitRates();
    totalEventRate = 0; // resets the last event rate
    for(TraitClass trait: traits)
        totalEventRate += trait.TraitRate;
}

```

```
void PopulationManager::calculateTotalTraitRates()
{
    calculateTraitDeathRates();
    calculateTraitBirthRates();
    calculateTraitProductionRates();
    calculateTraitSwitchRates();
    calculateTraitKillRates();
    calculateTraitRates();
}
```

3.2 sampleEventTime()

This should be self-explaining. The 'dice' class mas a method called 'rollExpDist(λ)' that samples $x \sim exp(\lambda)$. Obviously the eventTime doesn't contain the total time line. It represents the time between the occasions.
I will not describe the functionality of my dice object because its responsibility doesn't belong to the algorithm, but you can be assured that it has been tested very accurately before it was put into use.

```
void PopulationManager::sampleEventTime()
{
    event.eventTime = dice.rollExpDist(totalEventRate);
}

```

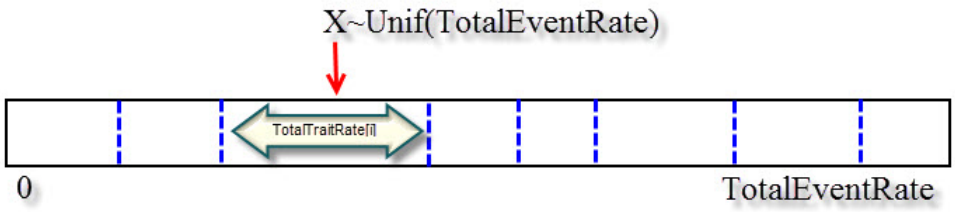

Figure 3: uniform trait selection

3.3 changePopulation()

To change the population, we first have to be sure that there is a need for changing. If there is no positive total Event Rate, the population has no reason to look for changes. That's the job of the first if statement. The names of the functions in the statement should clearly state their job description.

```
void PopulationManager::changePopulation()
{
    if(totalEventRate > 0){
        choseTraitToChange();
        choseEventType();
        executeEventTypeOnTrait();
    }
}
```

1. '**choseTraitToChange**()': The total event rate is made from all trait rates. This trait rates are used to decide witch trait triggered the event. The decision process is pictured in (fig. 3).

2. '**choseEventType**()': To chose the event type that occurs with the chosen trait, we will do the same as before in (fig. 3), but instead of the total event rate we use the total trait rate and the intervals will be the different event rates from that the total trait rate was made (death, birth, siwtch etc.).

3. '**executeEventTypeOnTrait**()': This function decides what population changing method should be called based on the event type. More on this in the next level.

Abstraction level three

4.1 calculating specific rates

This subsection summarizes the details of specific rate calculations that were mentioned in (calculateTotalEventRates()).

4.1.1 calculateTraitDeathRates()

The first for loop resets the last death rate of each trait. Than the natural deaths will be added. They aren't depending on other traits like the competition death rates. The order of these two isn't important.

```
    void PopulationManager::calculateTraitDeathRates()
    {
        for(TraitClass& trait : traits)
            trait.TraitDeathRate = 0;
        addNaturalDeathRates();
        addCompetitionDeathRates();
    }

```

4.1.2 calculateTraitBirthRates()

Like before the last birth rate for each trait has to be reset in the for loop. After this the natural birth rate will be added. Then each member of the population will reduce the this natural birth rate with 'subtractBirthReduction()'.

```
    void PopulationManager::calculateTraitBirthRates()
    {
        for(TraitClass& trait : traits)
            trait.TraitBirthRate = 0;
        addNaturalBirthRates();
        //addMutationalBirthRates();
        subtractBirthReduction();
    }

```

There is also a commented line '//addMutationalBirthRates();'. It indicates that it would be possible to change the algorithm with adding the mutational births to the birthrate in this place. A working algorithm for this already exists but would be inefficient (yet neatly arranged).

4.1.3 calculateTraitProductionRates()

To calculate the production of the t-cells, we first have to determine first how each melanoma population effects this production.

To achieve this we first count all effecting melanoma cells for each t-cell type. This is done in the first for-statement (line 4-8). The resulting amount is stored in the 'melCells' vector (initialized in line 3 with size of t-cells and values 0). The next for loop (line 9-11) calculates the trait production rate (t-cell production rate) for each trait with the product of the t-cell members, the effecting melanoma cells and the t-celltype specific production rate (line 10).

```
1  void PopulationManager::calculateTraitProductionRates()
2  {
3      vector<double> melCells(tCells,0);
4      for(size_t i = 0; i < tCells; ++i){
5          for(size_t j = 0; j < genotypes * phenotypes; ++j)
6              if(tCellKillRate[i][j] > 0)
7                  melCells[i] += traits[1 + tCells + j].Members;
8      }
9      for(size_t i = 1; i <= tCells; ++i){
10         traits[i].TraitProductionRate = traits[i].Members * melCells[
   i-1] * traits[i].ProductionRate;
11     }
12 }
```

It might be **confusing to track the indices**. The 'traits' container contains all traits without knowing the difference between them. Thats why we have to know that the first one in the container (traits[0]) is the TNF-α population, the next are all t-cell populations and then the melanoma cells ('traits' contains 1 + t-celltypes + genotypes $\times$ phenotypes trait objects).

4.1.4 calculateTraitSwitchRates()

The switch rate also consists of natural and competition switch rates. The competition switch rate is effected by the amount of TNF-α.

```
1  void PopulationManager::calculateTraitSwitchRates()
2  {
3      for(TraitClass& trait : traits)
4          trait.TraitSwitchRate = 0;
5      addNaturalSwitchRates();
6      addCompetitionSwitchRates();
7  }
8
```

4.1.5 calculateTraitKillRates()

These rates belong to the t-cells and describe how fast they are killing the melanoma cells. You can see on the influence for each t-cell type by the melanoma cells in line 6. And in line 8 we see the linear influence of the t-cell type population size on the rate.

```
1   void PopulationManager::calculateTraitKillRates()
    {
3       for(size_t i = 0; i < tCells; ++i){
            traits[1 + i].TraitKillRate = 0;
5           for(size_t j = 0; j < genotypes * phenotypes; ++j){
                traits[1 + i].TraitKillRate += tCellKillRate[i][j] *
    traits[1 + tCells + j].Members;
7           }
            traits[1 + i].TraitKillRate *= traits[1 + i].Members;
9       }
    }
11
```

4.1.6 calculateTraitRates()

Here you can track that the trait rate describes the event triggering rate of a specific trait. The for loop iterates over each trait in 'traits'. The '&' is important to address the real trait values and not just copies of it.

```
    void PopulationManager::calculateTraitRates()
2   {
        for(TraitClass& trait: traits){
4           trait.TraitRate = 0;
            trait.TraitRate += trait.TraitDeathRate;
6           trait.TraitRate += trait.TraitBirthRate;
            trait.TraitRate += trait.TraitProductionRate;
8           trait.TraitRate += trait.TraitSwitchRate;
            trait.TraitRate += trait.TraitKillRate;
10      }
    }
12
```

4.2 chose trait, chose event and execute

This subsection summarizes how we decide what event trigged the sampled time on which trait at the sampled time. You can find the time sampling in (sampleEvent-Time()). The following functions are executed in (changePopulation()).

4.2.1 choseTraitToChange

The first if statement (line 3) is for testing purpose so that this function can be tested individually. But the total event rate > 0 was already ensured in (changePopulation()) so it could be left out without expecting errors.
We use the decision process from the (uniform trait selection, fig. 3) to chose the trait that triggered a event. This process begins with sampling $X \sim Unif[0, r]$, r = totalEventRate in line 4.

The next for loop (line 5-11) represents the decision whether the current trait rate interval was hit by X or not.

```
1   void PopulationManager::choseTraitToChange()
    {
3       if(totalEventRate == 0){event.chosenTrait = 0; return;} // update
    : redundant
        double HittenTrait = dice.rollContUnifDist(totalEventRate);
5       for(size_t i = 0; i < populations; i++){
            if(HittenTrait <= traits[i].TraitRate){
7               event.chosenTrait = i;
                return;
9           }
            HittenTrait -= traits[i].TraitRate;
11      }
    }
13
```

4.2.2 choseEventType

First I would like to say that this function can be done more nicely. To ensure that this functionality is also self-explaining and is 'well written code', it could be implemented as a state pattern to switch the behaviour. Other patterns would also work (strategy/command pattern etc.), but the state pattern would nicely work with the next functionality.
Here you can find a very nice and simple (german) introduction to this pattern (http://www.philipphauer.de/study/se/design-pattern/state.php).

This method chooses the event type that was triggered by the chosen trait from before. The idea is the same as before (uniform trait selection, fig. 3). This time we use r = TraitRate instead of r = totalEventRate for sampling $X \sim Unif[0, r]$. This is done in line 4.
Then we again decide which event interval was hit by X. But it's not so easy to access the intervals in a for loop, that's why they where tested individually. The 'if' and 'else if' are passing until the first event decision returns true. The decision process is attached below the code.

```
    void PopulationManager::choseEventType()
2   {
        size_t i = event.chosenTrait;
4       double HittenEvent = dice.rollContUnifDist(traits[i].TraitRate);

6       if(decideIfEventTimeWasHitten(HittenEvent,traits[i].
    TraitDeathRate,DEATH)){}
        else if(decideIfEventTimeWasHitten(HittenEvent,traits[i].
    TraitBirthRate,BIRTH)){}
8       else if(decideIfEventTimeWasHitten(HittenEvent,traits[i].
    TraitSwitchRate,SWITCH)){}
```

```
        else if(decideIfEventTimeWasHitten(HittenEvent,traits[i].
    TraitProductionRate,PRODUCTION)){}
10      else if(decideIfEventTimeWasHitten(HittenEvent,traits[i].
    TraitKillRate,KILL)){}
        else if(totalEventRate == 0){event.type = NONE;}
12      else throw (string) "No Event was hit! Trait rate roll was bigger
     than the trait rate!";
    }
14
```

This is the decision process (it is equal to choseTraitToChange line 6-10):

```
1   bool PopulationManager::decideIfEventTimeWasHitten(double &
    HittenEvent, double &ChosenEventRate, EventType type)
    {
3       if(HittenEvent < ChosenEventRate){
            event.type = type;
5           return true;
        }
7       HittenEvent -= ChosenEventRate;
        return false;
9   }
```

4.2.3 executeEventTypeOnTrait

Here we decide on the basis of the event type what we will do to the population. This decision is visualized in the (Depth-diagram) by the diamond with the decision parameter 'Type' which in our case is the 'event.type'.

```
1   void PopulationManager::executeEventTypeOnTrait()
    {
3       if(event.type == DEATH){traits[event.chosenTrait].Members -= 1.;}
        else if(event.type == BIRTH){executeBirth();}
5       else if(event.type == SWITCH){executeSwitch();}
        else if(event.type == PRODUCTION){executeProduction();}
7       else if(event.type == KILL){executeKill();}
        else if(event.type == NONE){}
9       else throw (string) "No event type has been chosen before:
    executeEventTypeOnTrait()";
    }
11
```

The 'throw' statement in the else part is already unnecessary. It simply throws a custom exception if the event type has no valid value. But this can't happen, so it can be left out. In fact it should be deleted because it counts as a very light version of dead code which is known as a 'code smell'.

This many 'if' and 'else if' statements are not very nice and can easily be replaced by a (switch, case) statement. This would look like this:

```
void PopulationManager::executeEventTypeOnTrait()
{
  switch (event.type){
  case BIRTH:
    executeBirth();
    break;
  case SWITCH:
    executeSwitch();
    break;
  case PRO...
    ...
  }
}

```

Abstraction level four

This section describes the details of some functions of level three with more than one responsibility.

5.1 death rate details

The death rate is composed of the natural deaths and the competition deaths.

5.1.1 addNaturalDeathRates()

Each member of a trait has a constant natural death rate. Line 4 should be self-explaining.

```
1  void PopulationManager::addNaturalDeathRates()
   {
3      for(TraitClass& trait : traits)
           trait.TraitDeathRate += trait.Members * trait.DeathRate;
5  }
```

5.1.2 addCompetitionDeathRates()

To observe the competition deaths, we iterate over each member of the population (for-loop in line 4) and consider the competition effect (for-loop in line 6) for each member of the population (line 9).

```
1   void PopulationManager::addCompetitionDeathRates()
    {
3       double sum;
        for(size_t i = 0; i < populations; ++i){
5           sum = 0;
            for(size_t j = 0; j < populations; ++j){
7               sum += competition[i][j] * traits[j].Members;
            }
9           traits[i].TraitDeathRate += sum * traits[i].Members;
        }
11  }
```

5.2 birth rate details

The birth rate is composed of the natural births and the birth reduction.

5.2.1 addNaturalBirthRates()

Natural births are very straight forward like the natural deaths.

```
1   void PopulationManager::addNaturalBirthRates()
    {
3       for(TraitClass& trait : traits)
            trait.TraitBirthRate += trait.Members * trait.BirthRate;
5   }
```

5.2.2 subtractBirthReduction()

There is also an birth reducing effect for each new member of a trait. This reducing effect gets calculated in the for loop line 6-8 and effects the birth rate for this trait in line 9.
This may cause the birth rate for the trait to be negative what would lead to problems to technically process the concept of (fig. 3) which is used in (choseTraitToChange and choseEventType). This justifies the need of line 11-12.

```
1   void PopulationManager::subtractBirthReduction()
    {
3       double sum;
        for(size_t i = 0; i < populations; ++i){
5           sum = 0;
            for(size_t j = 0; j < populations; ++j){
7               sum += birthReduction[i][j] * traits[j].Members;
            }
9           traits[i].TraitBirthRate -= sum * traits[i].Members;
```

```
11              if(traits[i].TraitBirthRate < 0)
                    traits[i].TraitBirthRate = 0;
13          }
        }
15
```

5.3 switch rate details

The switch rate is composed of the natural switch and the switch due to TNF-alpha.

5.3.1 addNaturalSwitchRates()

The switch rate matrix differs for each genotype. That's why we need three for loops (2 loop → matrix, 3 loops → tensor 3rd dim). Within the first for loop we see the usual behaviour to calculate the switch rate for the k.th genotype. It calculates the effect switch rate to make a switch for each member (line 6-7) and considers that for the entire trait in line 9-10. The '+=' in line 10 can also be a '='. But the '+=' symbolizes that there is also a competition rate as a part of the trait switch rate. In this version both (addNaturalSwitchRates and addCompetitionSwitchRates) don't need to be executed in a particular order (→ more independency).

```
    void PopulationManager::addNaturalSwitchRates()
2   {
        double switchrate;
4       for(size_t k = 0; k < genotypes; ++k){
            for(size_t i = 0; i < phenotypes; ++i){
6               switchrate = 0;
                for(size_t j = 0; j < phenotypes; ++j) // Improve: reuse!
8                   switchrate += naturalSwitch[k][i][j];
                switchrate *= traits[getMelanomIndex(k,i)].Members;
10              traits[getMelanomIndex(k,i)].TraitSwitchRate +=
    switchrate;
            }
12      }
    }
14
```

There is one comment 'Improve:...'. It states a simple improvement that can be made by calculating the inner for loop (line 7-8) once the simulation starts, because its the same calculation each step. If you would like to have this improvement than please contact me and I send you the updated version.

5.3.2 addCompetitionSwitchRates()

First I would like to emphasise that in the old version, the 'traitNumber' variable in this function is a relict before there was a 'getMelanomIndex(k,i)' method. If you got

the version with the 'traitNumber' than please simply delete it because I has no effect on the algorithm. As you can see below it get only incremented but never used. The less confusing algorithm is the second code snippet.
In general there is one issue with this calculation. It is very similar to the natural rate with additional influence of TNF-α in line 9 (and competition instead of natural rate). That's why there is a simple way to increase the efficiency by adding the additional increment after the natural switchrate. You can find this approach in the third code snippet.
I assume that this happened because I stopped working after the natural switch rates were tested and implemented and started the competition switch rates the next morning.
Snippet with dead code:

```
1   void PopulationManager::addCompetitionSwitchRates()
    {
3       size_t traitNumber;
        double switchrate;
5       for(size_t k = 0; k < genotypes; ++k){
            traitNumber = k * phenotypes;
7           for(size_t i = 0; i < phenotypes; ++i){
                traitNumber += i;
9               switchrate = 0;
                for(size_t j = 0; j < phenotypes; ++j)
11                  switchrate += competitionSwitch[k][i][j];
                switchrate *= traits[getMelanomIndex(k,i)].Members *
    traits[0].Members;
13              traits[getMelanomIndex(k,i)].TraitSwitchRate +=
    switchrate;
            }
15      }
    }
17
```

Snippet without dead code:

```
    void PopulationManager::addCompetitionSwitchRates()
2   {
        double switchrate;
4       for(size_t k = 0; k < genotypes; ++k){
            for(size_t i = 0; i < phenotypes; ++i){
6               switchrate = 0;
                for(size_t j = 0; j < phenotypes; ++j)
8                   switchrate += competitionSwitch[k][i][j];
                switchrate *= traits[getMelanomIndex(k,i)].Members *
    traits[0].Members;
10              traits[getMelanomIndex(k,i)].TraitSwitchRate +=
    switchrate;
            }
12      }
    }
14
```

The improved code would be:

```
void PopulationManager::addSwitchRatesImproved()
{
    double natSwitchSum;
    double compSwitchSum;
    for(size_t k = 0; k < genotypes; ++k){
        for(size_t i = 0; i < phenotypes; ++i){
            natSwitchSum = 0;
            compSwitchSum = 0;
            for(size_t j = 0; j < phenotypes; ++j){
                natSwitchSum += naturalSwitch[k][i][j];;
                compSwitchSum += competitionSwitch[k][i][j];
            }
            natSwitchSum *= traits[getMelanomIndex(k,i)].Members;
            compSwitchSum *= traits[getMelanomIndex(k,i)].Members *
traits[0].Members;
            traits[getMelanomIndex(k,i)].TraitSwitchRate =
natSwitchSum + compSwitchSum;
        }
    }
}

```

Attention: I haven't tested the improved code. If you want to have this improvement (it could increase the efficiency noticeably), please contact me about this matter. I simply don't want to force a newer version of the simulator if the current version already works fine and is already modified by you.

5.4 execute event type details

Here we see how the events getting executed on the chosen traits (except the killing event. This trait rather executes than getting executed ◡̈). (see executeEventTypeOnTrait)

execute death

This is simply one line and doesn't deserve is own function. That's why it gets executed directly in (executeEventTypeOnTrait).

```
    traits[event.chosenTrait].Members -= 1.;

```

5.4.1 executeBirth()

Here comes the influence of a mutation. While a birth can be triggered from the chosen trait, it still can be a mutant and belong to another genotype. If its a melanoma cell (or a TNF-α) than we can simply ask 'isMutation()' if its a mutant or not(TNF-α's and t-cells mutation probability gets initialized with zero). In case of mutation the function 'choseMutantGenotype' redirects the chosenTrait to the mutant trait. I will attach both ('isMutation()' and 'choseMutantGenotype') snippets after the birth method.

```
1   void PopulationManager::executeBirth()
    {
3     if(isMutation()){
            choseMutantGenotype();
5           traits[event.chosenTrait].Members++;
      }
7     else{
            traits[event.chosenTrait].Members++;
9     }
    }
11
```

Below you see the decision if its a mutation or not. It simply considers the mutation probability of the chosen trait (only the melanoma cells have positive probabilities $\rightarrow$ in fact only the genotypes have differ from each other). The comment shows that this decision can be done in one line but would not be so overseeable.

```
    bool PopulationManager::isMutation()
2   {
        double roll = dice.rollContUnifDist(1);
4       if(roll < traits[event.chosenTrait].Mutation)
            return true;
6       else
            return false;
8     //return roll < traits[event.chosenTrait].Mutation ? true : false;
    }
10
```

To chose the mutant after we know that there will be a mutant is again done with (uniform trait selection, fig. 3).
First we select the genotype and phenotype of the chosen trait in line 3 and 4 (you can look the functions up in the source code). Than we sample $X \sim Unif([0,1])$ (line 6) and chose the intervals like the ones in the given mutation kernel (line 9). This procedure should familiar by now.

```
1   void PopulationManager::choseMutantGenotype()
    {
3       size_t chosenGenotype = getGenotypeOfTraitIndex(event.chosenTrait
    );
        size_t chosenPhenotype = getPhenotypeOfTraitIndex(event.
    chosenTrait);
```

```
5         size_t mutantGenotype = 0;
          double HittenTrait = dice.rollContUnifDist(1);
7
          for(mutantGenotype = 0; mutantGenotype < genotypes; ++
    mutantGenotype){
9             if(HittenTrait <= mutationKernel[chosenGenotype][
    mutantGenotype])
                  break;
11            HittenTrait -= mutationKernel[chosenGenotype][mutantGenotype
    ];
          }
13        event.chosenTrait = getMelanomIndex(mutantGenotype,
    chosenPhenotype);
      }
15
```

5.4.2 executeSwitch()

This function is rather simple, but the 'getSwitchedTrait()' call gets a bit confusing because there are many linebreaks. If you look it up in the code editor than it will be much more readable.

```
    void PopulationManager::executeSwitch()
2   {
        size_t switchedTrait = getSwitchedTrait();
4       traits[event.chosenTrait].Members--;
        traits[switchedTrait].Members++;
6   }
```

Below you see 'getSwitchedTrait()' which finds the switched trait among the genotype (the chosen trait will switch to the switched trait). It is very recommended to look at this method in the Qt Creator or another editor.
In line 3 we sample $X \sim Unif([0, r])$ with $r = \frac{TraitSwitchRate[k]}{Members[k]}$. Where k represents the chosen trait index. This gives us the actual switchrate for one member (consisting of the natural rates and the competition rates times the TNF$-\alpha$).
Than we determine the genotype and phenotype of the chosen trait in line 4 and 5. In the for loop we consider each possible phenotype to switch to. In line 7 you can see that the switchrate for the considered switchable trait gets calculated and like before in uniform trait selection (fig. 3), we decide of this switchrate caused the switch. It this was the case than we return the current considered phenotype as switch destination.

```
    size_t PopulationManager::getSwitchedTrait()
2   {
        double HittenSwitch = dice.rollContUnifDist(traits[event.
    chosenTrait].TraitSwitchRate/traits[event.chosenTrait].Members);
```

```
4         size_t chosenGenotype = getGenotypeOfTraitIndex(event.chosenTrait
    );
          size_t chosenPhenotype = getPhenotypeOfTraitIndex(event.
    chosenTrait);
6         for(size_t switchPhenotype = 0; switchPhenotype < phenotypes; ++
    switchPhenotype){
              double currentSwitchRate = getSpecificSwitchRate(
    chosenPhenotype, chosenGenotype, switchPhenotype);
8             if(HittenSwitch <= currentSwitchRate)
                  return getMelanomIndex(chosenGenotype, switchPhenotype);
10            HittenSwitch -= currentSwitchRate;
          }
12        throw (string) "getSwitchedTrait() could not find a matching
    switch!";
          return 0;
14    }
```

5.4.3 executeProduction()

If a t-cells gets produced, than there appear 'ProductionAmount' (depending on the chosen t-cell) new TNF-α's and the t-cell population increases by one.

```
    void PopulationManager::executeProduction()
2   {
        traits[0].Members += traits[event.chosenTrait].ProductionAmount;
4       traits[event.chosenTrait].Members++;
    }
6
```

5.4.4 executeKill() - unfinished

Self-explaining.

```
1   void PopulationManager::executeKill()
    {
3       traits[getKilledTrait()].Members--;
        traits[0].Members += TNFsAfterKill;
5   }
```

This is again better in the editor, else the linebreaks may cause confusion.

```
1   size_t PopulationManager::getKilledTrait()
    {
3       double HittenCell = dice.rollContUnifDist(traits[event.
    chosenTrait].TraitKillRate/traits[event.chosenTrait].Members);
        size_t tcell = event.chosenTrait -1;
```

```
5         for(size_t i = 0; i < genotypes*phenotypes; ++i){
              if(HittenCell < tCellKillRate[tcell][i] * traits[1+ tCells +
    i].Members)
7                 return 1+tCells+i;
              HittenCell -= tCellKillRate[tcell][i] * traits[1+ tCells + i
    ].Members;
9         }
          throw (string) "no killed trait found!";
11        return 0;
      }
13
```

Background Information

6.1 Parameter Input

This section reminds (very brief) how the input of the simulator has to be and how it gets processed.
There exists a 'MusterInstanz' and a 'Testinstanz_kommentiert' file in the repository and in the milestone 1.1.4. The first explains the structure of the parameter files and the second gives an example of this. You can observe the second file in (fig. 4). Make sure you know how to read the 'Deathratematrix due T-cell resistance'. The Parameters were read with my own builded 'CFileStreamer' that can be seen in the source files. But in general its not important. In the following snippet you see how the Parameters are processed. In line 3-4 the streaming object of the desired file gets created. It contains all informations as a string. In each function line 5-12 it iterates over the strings in the stream and stores the expected parameters.

```
  void PopulationManager::initWithFile(const string FName)
2 {
      CFileStreamer object(FName);
4     IFileStreamer& stream(object);
      readPopulations(stream);
6     createTraits(stream);
      readDeathBirthProductionRates(stream);
8     readCompetition(stream);
      readBirthReduction(stream);
10    readNaturalCompetitionSwitchRates(stream);
      readTCellKillRateAndAmount(stream);
12    readMutationWithKernel(stream);
      K = std::stod(stream.getNextWord()); // Read K from filestream
14    applyKToParameters();
  }
16
```

```
2               N1 - T cells
3               N2 - Genotypes
2               N3 - Phenotypes
10              TNF-alpha
10 10           T-cells
20 20           Phenotypes for 1. Genotype (N3 entries)
20 20           Phenotypes for 2. Genotype
20 20           Phenotypes for 3. Genotype
1 1 1 2 2 2 2 2       Deathrates (N entries)
5 5 5 5 5 6 6 6 6       Birthrates (N entries)
1 2                   Productionrate for T-cells (N1 entries)
2 3                   Productionamount for T-Cells
1 0 0 0 0 0 0 0 0       Competition matrix (NxN)
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
0.1 0 0 0 0 0 0 0 0     Birthreducing matrix (NxN)
0 0.1 0 0 0 0 0 0 0
0 0 0.1 0 0 0 0 0 0
0 0 0 0.1 0 0 0 0 0
0 0 0 0 0.1 0 0 0 0
0 0 0 0 0 0.1 0 0 0
0 0 0 0 0 0 0.1 0 0
0 0 0 0 0 0 0 0.1 0
0 0 0 0 0 0 0 0 0.1
0 2             1. Switchmatrix (natural) N3xN3 (1. genotype)
1 0
0 1             2. Switchmatrix
0 0
0 1             3. Switchmatrix
0.5 0
0 1             1. Switchmatrix (due to TNF-alpha) N3xN3 (1. genotype)
0.5 0
0 0.5           2. Switchmatrix
1 0
0 0             3. Switchmatrix
0.5 0
2 2 2 2 1 1     Deathratematrix due T-cell resistance.
1 1 1 2 2 2     It's Different for phenotypes and genotypes.
3               resulting TNF-alpha after T-cell killings.
0.2 0.2 0.2     Mutation probabilities for each genotype
0 0.2 0.8       Mutationkernel
0.5 0 0.5
0.5 0.5 0
1               K
```

Figure 4: Testinstanz_kommentiert

6.2 Influence of K

As you maybe noticed, there is nothing about the influence of the K. The simulation itself doesn't use the K to iterate the algorithm. The K gets applied when the parameters were read and after the algorithm has finished its work. In 'Parameter Input' you can see in the code snipped line 14 that the K gets applied to the parameters after they are known. In the following snipped you can observe what happens.

```
1  void PopulationManager::applyKToParameters()
2  {
3      for(TraitClass& trait : traits){ // Members and ProductionRates
4          trait.Members *= K;
5          trait.ProductionRate /= K;
6      }
7      for(size_t i = 0; i < populations; ++i){     // competition death
8          for(size_t j = 0; j < populations; ++j)
9              competition[i][j] /= K;
10     }
11     for(size_t i = 0; i < populations; ++i){     // birth reduction
12         for(size_t j = 0; j < populations; ++j)
13             birthReduction[i][j] /= K;
14     }
15     for(size_t k = 0; k < genotypes; ++k){       // competition switch
16         for(size_t i = 0; i < phenotypes; ++i){
17             for(size_t j = 0; j < phenotypes; ++j)
18                 competitionSwitch[k][i][j] /= K;
19         }
20     }
21     for(size_t i = 0; i < tCells; ++i){          // killRate
22         for(size_t j = 0; j < phenotypes * genotypes; ++j)
23             tCellKillRate[i][j] /= K;
24     }
25 }
```

To ensure that the plot shows the population with the desired K, it accesses the members of the trait only through a special accessor method:

```
1  double PopulationManager::getKMembers(int TraitIndex) const
2  {
3      return traits[TraitIndex].Members / K;
4  }
5
```

Unfortunately it would be far to expensive to document the actual plotting process and the graphical implementation because it's very technical and even more work than the algorithm itself. Also its much harder to test.

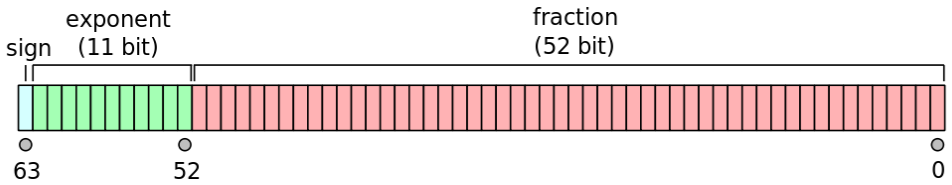

Figure 5: double precision format ®Codekaizen/Wikimedia

6.3 Precision problems

The precision was chosen to be the standard double precision. Now the interesting question would be, what can cause precision related false data? To answer that I would like to explain what double precision is and what a 'Mantissa' is.
The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. I.e. of 210.45 would have the scientific notation 2.1045×10^2 where 2.1045 are the significant digits thus the mantissa. Its also called the significand.
In nearly any case, precision problems are caused by mantissas that get to long. Now, how does a double variable look and when do precision errors occur? As you can see in (fig. 5) a double variable can store 63 bits. In our case are only 62 interesting because we don't get negative numbers. The first 11 bits (ignoring the sign) are used to represent the exponent of the scientific notation. In the example of the mantissa this would be 2 (from 2.1045×10^2). This means that the exponent can maximal 2048. In the case of the simulation this is very unlikely to reach. The more interesting matter are the next 52 bits for the mantissa. 52 bits mean that the maximal mantissa can be 4503599627370496. This means that we can expect the digits to be precise for a mantissa with maximal length $\log_{10}(2^{53}) = 53 \log_{10}(2) \approx 15.95 \approx 16$.
In short, if the mantissa exceeds 16 digits, than the less important digits will get lost (lowest digits). Normally the exponent doesn't exceed, thus we don't have to worry that the numbers are getting very small or big.