@code\_type c++ .cpp @comment\_type // %s @compiler lit -t main2.md && g++ random.o -Wall main2.cpp @title 2D simulation

# Introduction

This is the 2D version of the simulation. It is identical to the 3D version, except that dimensions of arrays are 2, and loops are smaller. This file was derived from the 3D version.

The program simulates a compartment with possible cancer cells by iterating steps of the Moran process. Each run starts from a single mutation till fixation. One modification to the regular Moran is that cells to replace are chosen so that the lump of mutants stays compact.

For more information about compiling this program, see <https://github.com/mic-lachmann/SignalingCancer>

# Main program

## The Moran process

Let us start with an overview of the implementation of the Moran Process.

In each single run first initialise the full compartment, stored in MatrixState of size iSize\*jSize\*kSize, to type 0, and set one single cell (the middle one) to type 1.

### initialise compartment and introduce single mutant

bool MatrixState [ iSize ][ jSize ] = {{ 0 }} ;  
MatrixState [ iSize/2 ][ jSize/2 ] = 1 ; // Mutant

### Single moran step

Then at each Moran step choose a cell to replicate, the *reproducer*, and a cell that will be replaced.

@{Choose \*reproducer\* according to fitness}  
@{Choose random cell within range L \*to be replaced\*}  
@{Find cell of the same type as \*to be replaced\*, having maximum neigbours of same type as the \*reproducer\*, and replace it instead}

Note that finding the cell to be replaced has two stops: first chose one in radius L, then chose an identical one anywhere that has a maximal number of neighbours of the same type as the reproducer.

### main loop single run

This is done till the mutant either is extinct, or takes up the whole space. call\_count records how long this took.

@{initialise compartment and introduce single mutant}  
 static unsigned int call\_count = 0;  
 do {  
 @{Single moran step}  
 call\_count++ ;  
 } while( SumArr2(MatrixState, iSize, jSize ) != 0 &&   
 SumArr2(MatrixState, iSize, jSize ) != PopSize );  
 // Notice that here SumArr3 is called twice. Also, in a single step, only one replacement takes  
 // place, and yet the 6x6x6 sum is calculated again (twice)

#### definition of PopSize

Notice that PopSize is set to the size of the array

const int PopSize = iSize\*jSize ;

## Main program for 3D run

The program has a very simple structure, of a main loop, with inits and printing results.

### main2.cpp

//  
// main.cpp  
// Cancer3D  
//  
// Created by Leonardo Ona Bubach on 04/11/2015.  
// Copyright Â© 2015 Leonardo Ona Bubach. All rights reserved.  
//  
  
@{includes}  
@{Parameters}  
@{Functions}  
int main() {  
 ofstream outputfile ("output.csv");  
   
 @{main program inits}  
   
 @{main loop stats init}  
   
 @{main loop}  
  
 outputfile.close();  
   
 return 0;  
}

# includes

We will gather includes as we need them over the program. <fstream> is needed for ofstream.

#include <fstream>

## main loop

The main loop runs across different parameters. In the current incarnation, it runs across parameter R, and for each value of R runs nRuns runs. The arrays whoWon[] and fixTime[] will collect results on each single run.

for (int R=Rmin; R <= Rmax ; R=R+Rstep) {  
  
 int whoWon[nRuns]={0};  
 int fixTime[nRuns]={0};  
   
 for (int i=0; i<nRuns; i++){  
 @{main loop single run}  
 @{record who won and fixation time}  
 }  
   
 @{collect stats and print results}  
}

## Stats and printout

### record who won and fixation time

After the run record who fixed by sampling a single cell, and record time to fixation from call\_count.

whoWon[i] = MatrixState[1][1] ;  
fixTime[i] = call\_count;

During the actual main loop, whoWon and fixTime recorded results. These are converted to the statistics we actually want to print out. These are the fixation probability, and average time to fixation, conditional on fixation.

### main loop stats init

double fixProbR[ nR ] ={ 0.0 };  
double avFixTimeR[ nR ]={ 0.0 };

We loop over all runs, and for cases in which the mutant fixed, record time to fixation in OnlyCasesOneFixTime[]. All other entries stay 0. Counter1 records how often this happened. ### collect stats and print results

//\*\*\*\*The following vector save cases where 1 reach fixation\*\*\*\*\*  
   
int OnlyCasesOneFixTime[nRuns]={0};  
  
int Counter1=0;  
  
for (int i=0; i<nRuns; i++) {  
 if(whoWon[i]==1){ // mutant fixed  
 Counter1++;  
 OnlyCasesOneFixTime[i]=fixTime[i];  
 }  
 } // End of the For loop for Runs values  
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Now we just need ### collect stats and print results +=

fixProbR[ (R-Rmin) / Rstep ]= (double(Sum( whoWon, nRuns))/nRuns);  
avFixTimeR[ (R-Rmin) / Rstep ]= (double(Sum( OnlyCasesOneFixTime, nRuns))/Counter1)/PopSize;  
  
  
cout << endl;  
cout << endl;  
  
  
cout << "Radius:"<< endl;  
for (int i= Rmin; i <= Rmax; i= i+Rstep) {  
 cout << i << " ";  
}  
  
cout << endl;  
cout << endl;  
  
  
  
cout << "Fix Prob dep R:"<< endl;  
for (int i=0; i<=(Rmax-Rmin)/Rstep; i++) {  
 cout << fixProbR[i] << " ";  
 }  
  
cout << endl;  
cout << endl;  
  
cout << "Av Fix Time dep R:"<< endl;  
for (int i=0; i<=(Rmax-Rmin)/Rstep; i++) {  
 cout << avFixTimeR[i] << " ";  
}  
  
  
  
cout << endl;  
cout << endl;  
cout << endl;  
  
  
  
for (int i=0; i<=(Rmax-Rmin)/Rstep; i++) {  
 outputfile << i << "," << fixProbR[i] << "," << avFixTimeR[i] << endl;  
}

##### includes +=

Using cout means we need

#include <iostream>

# Parameters

using namespace std;  
  
  
  
  
  
//PARAMETERS  
  
const double b=1.0; //Benefit  
const double c=0.01; //Cost  
//const int R=30; //Radius  
const int Rmin=0; //Min Radius in a loop  
const int Rmax=15; //Max Radius in a loop  
const int Rstep=2; //R step in a loop  
const int nR= ((Rmax-Rmin)/Rstep)+1; //Number of data per Radius  
const int L=7; //Range of local competition L  
const int iSize=30;  
const int jSize=30;  
  
const int PopSize = iSize \* jSize;  
//const int first1=PopSize/2; //Position of the unique mutant signalling cell in the begining of the simulation  
  
const int nRuns=1000; //Number of runs  
  
  
int MapVM[iSize][jSize]= {{ 0 }} ;

# Functions

@{IntegerRandom}  
@{Sum}  
@{SumArr2}  
@{WithinCompartment}  
@{MatMappingInitialization}  
@{MatToVet}  
@{VetToMat}  
@{ChooseAnElement}  
@{NeighbourWithinL}  
@{Replacement}

//FUNCTIONS

### IntegerRandom

//The function "IntegerRandom" returns a random integer from min (inclusive) to max (exclusive): [min,max). When called with one argument gives [0,max)  
int IntegerRandom(const int max, const int min=0)  
{  
 return static\_cast<int>( rnd::uniform()\*(max-min)+min );  
}

rnd is defined in random.h ##### includes +=

#include "random.h" // for rnd::uniform()

### Sum

int Sum(const int \*pnArray, const int nLength)  
{  
 int val =0;  
 for ( int i=0; i<nLength; i++)  
 val += pnArray[i];  
 return val;  
}

### SumArr2

int SumArr2(const bool pnArray[][jSize], const int nLength, const int mLength)  
{  
 int val =0;  
 for (int i=0; i<nLength; i++){  
 for (int j=0; j<mLength; j++) {  
 val+=pnArray[i][j];  
 }  
 }  
 return val;  
}

### WithinCompartment

int WithinCompartment( int i, int j, int k)  
{  
 return (0 <= i) && (i < iSize ) &&  
 (0 <= j) && (i < jSize ) ;  
}

## Conversion between cell index and coordinates.

The following functions convert between the index of a cell and its coordinates.

### main program inits

This initialises the mapping between multidimensional array representation of the compartment and the one-dim index representation.

MatMappingInitialization(MapVM);

#### MatMappingInitialization

We initialise the 3D MapVW array to hold the index of every cell.

void MatMappingInitialization(int (&MapVM)[iSize][jSize]){  
   
 int index=0;  
 for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 MapVM[i][j] = index;  
 index++;  
 }  
   
 }  
}

#### MatToVet

To get the index of a cell we simply return that entry.

int MatToVet(const int ik, const int jk)  
{  
 int ret=0;  
 int counn=0;  
 for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 if (i==ik && j==jk) {  
 ret= counn;  
 }  
 counn++;  
 }  
 }  
 return ret;  
}

### VetToMat

To get the coordinates of a certain index we loop over all of VetToMat to find the entry that is equal to the one needed.

int VetToMat(const int valueFromVector, const int ZeroOrOne)  
{  
 int MapVM[PopSize][3]={{ 0 }};  
   
 int count2=-1;  
 int count3=0;  
   
 for (int i=0; i<PopSize; i++) {  
 count2++;  
 for (int j=0; j<3; j++) {  
 if (j==0) {  
 MapVM[i][j]=count2;  
 } else  
 if (j==1){  
 MapVM[i][j]= count3%iSize;  
 if (j==1 && count2%jSize == (jSize-1)) {  
 count3++;  
 }  
 }  
 else  
 if (j==2) {  
 MapVM[i][j]= count2%jSize;  
 }  
 }  
 }  
 return MapVM[valueFromVector][ZeroOrOne+1];  
}

# Fitness and replacement

## Choose *reproducer* according to fitness

Choosing the reproducing cell is done in the function ChooseAnElement.

int reproducer = ChooseAnElement( MatrixState, R);

### ChooseAnElement

This function calculates the fitness of each cell in the compartment, and then choses a cell to reproduce by fitness.

int ChooseAnElement(const bool StateArray[iSize][jSize], const int R)  
{  
 @{ChooseAnElement: initialise fitness array (MatrixFitness)}  
 @{Calculate fitness}  
 @{Choose random cell}  
}

#### ChooseAnElement: initialise fitness array (MatrixFitness)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Initialise Fitness Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 double MatrixFitness[iSize][jSize] = {{ 0 }};

#### Calculate fitness

There are two separate models to calculate fitness

#ifdef CUMULATIVE  
 @{Calculate fitness cumulative}  
#else  
 @{Calculate fitness non cumulative}  
#endif

#### Calculate fitness cumulative

In the cumulative model, with neighbours that are of type 1, the fitness of a type 1 cell is , and of a type 0 cell . Thus we calculate fitness in the following way:

@{set fitness to 1}  
@{for every cell of type 1 add b to all neighbours}  
@{subtract c from every cell of type 1}

#### Calculate fitness non cumulative

In the cumulative model, the difference is simply that if there is any neighbour of type 1, benefit is , otherwise 0.

@{set fitness to 1}  
@{for every cell of type 1 set all neighbours to 1+b}  
@{subtract c from every cell of type 1}

#### set fitness to 1

for (int i=0; i<iSize; i++){  
 for (int j=0; j<jSize; j++) {  
 MatrixFitness[i][j] = 1;  
 }  
}

#### for every cell of type 1 add b to all neighbours

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* This Part A: Cumulative Model \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 if (StateArray[i][j]==1) {  
 for (int k = -R; k <= R ; k++) {  
 for (int l = -R ; l <= R ; l++) {  
 if ((i+k)>=0 && (i+k)<iSize && (j+l)>=0 && (j+l)<jSize) {  
 MatrixFitness[i+k][j+l] += b ; // was \* StateArray[i][j]. but it is 1.  
 }  
 }  
 }  
 }  
 }  
 }

#### for every cell of type 1 set all neighbours to 1+b

for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 if (StateArray[i][j]==1) {  
 for (int k = -R; k <= R ; k++) {  
 for (int l = -R ; l <= R ; l++) {  
 if ((i+k)>=0 && (i+k)<iSize && (j+l)>=0 && (j+l)<jSize) {  
 MatrixFitness[i+k][j+l] = 1 + b; // was b, but should be 1+b.  
 }  
 }  
 }  
 }  
 }  
 }

#### subtract c from every cell of type 1

for (int i=0; i<iSize; i++){  
 for (int j=0; j<jSize; j++) {  
 if (StateArray[i][j]==1) {  
 MatrixFitness[i][j]+= -(c);  
 }  
 }  
 }

#### Choose random cell

First, calculate non-normalised CDF for the fitness of all cells in compartment. Then sample 1 of them using this CDF.

double FitnessArray[PopSize] = {0};  
@{transform fitness matrix to fitness array}  
@{calculate (non normalised) CDF of fitness array}  
@{sample 1 random cell using CDF}

##### transform fitness matrix to fitness array

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Transforming Matrix Fitness to Array Fitness\*\*\*\*\*\*\*\*\*\*\*\*\*  
 int counter=0;  
 for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 FitnessArray [counter]= MatrixFitness[i][j];  
 counter++;  
 }  
 }

##### calculate (non normalised) CDF of fitness array

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Cumulative vector \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 double FitnessCDF[PopSize]={0};  
 for (int i=0; i<PopSize; i++){FitnessCDF[i] =FitnessArray[i];}  
 for (int i=1; i<PopSize; i++){FitnessCDF[i] += FitnessCDF[i-1];}

##### sample 1 random cell using CDF

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Return an Element weighted sampled \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 return rnd::sample\_1( FitnessCDF, PopSize);

rnd is defined in random.h ##### includes +=

#include "random.h" // for rnd::sample\_1

## Choose random cell within range L *to be replaced*

int to\_be\_replaced = NeighbourWithinL( reproducer ) ;

### NeighbourWithinL

Find a neighbour in radius L from focal, return its index. This is done by choosing random neighbour, and checking if it fulfils some conditions.

int NeighbourWithinL(const int addresoffocal)  
{  
 int a[2]={0,0};  
 do{  
 a[0] = VetToMat(addresoffocal,0) - L + IntegerRandom(2\*L+1);  
 a[1] = VetToMat(addresoffocal,1) - L + IntegerRandom(2\*L+1);  
 } while ((a[0] == VetToMat(addresoffocal,0) && a[1] == VetToMat(addresoffocal,1))  
 || a[0]<0 || a[1]<0  
 || a[0]>(iSize-1) || a[1]>(jSize-1));  
 return MatToVet( a[0], a[1]);  
}

## Find cell of the same type as *to be replaced*, having maximum neigbours of same type as the *reproducer*, and replace it instead

Replacement( MatrixState, reproducer, to\_be\_replaced);

### Replacement

void Replacement( bool (&MatrixSt)[iSize][jSize],   
 const int focal\_Vet, const int replace\_Vet)   
{  
 // call\_count++; I moved this out  
   
 // If the focal and replace are different, do the replacement  
 int focal[2] = { VetToMat( focal\_Vet , 0),  
 VetToMat( focal\_Vet , 1) } ;  
 int replace[2] = { VetToMat( replace\_Vet, 0),  
 VetToMat( replace\_Vet, 1) } ;  
 bool focal\_type = (MatrixSt)[ focal[0] ][ focal[1] ] ;  
 bool replace\_type = (MatrixSt)[ replace[0] ][ replace[1] ] ;  
   
 if( focal\_type != replace\_type) {  
 if ( focal\_type == 1) {  
   
 @{Calculate number of immediate neighbours of type 1 for each cell of type 0}  
 @{Find the cells with the maximal number of neighbours of type 1}  
   
 // choose one of them, those with the maximal #of neighbours diff from them.  
 int choo= VectorAddresMaximum[ IntegerRandom(static\_cast<int>(VectorAddresMaximum.size()))];  
  
 // Finally, set that one to 1, we replicated.  
 MatrixSt[VetToMat(choo, 0)][VetToMat(choo, 1)] = 1 ;  
   
 // Erase the vectors. ### why is this needed? Won't they be erased on their own?  
 VectorZeroAddress.erase (VectorZeroAddress.begin(), VectorZeroAddress.begin()+VectorZeroAddress.size());  
 VectorNOnes.erase (VectorNOnes.begin(), VectorNOnes.begin()+VectorNOnes.size());  
 VectorAddresMaximum.erase (VectorAddresMaximum.begin(), VectorAddresMaximum.begin()+VectorAddresMaximum.size());  
 } else { // if the focal isn't 1, i.e. if the focal is 0.  
 @{Calculate number of immediate neighbours of type 0 for each cell of type 1}  
 @{Find the cells with the maximal number of neighbours of type 0}  
  
 // chose a random maximal cell  
 int choo=VectorAddresMaximum[IntegerRandom(static\_cast<int>(VectorAddresMaximum.size()))];  
  
 // set it to 0, i.e. to the value of the focal  
 MatrixSt[VetToMat(choo, 0)][VetToMat(choo, 1)] = 0 ;  
   
   
 VectorOneAddress.erase (VectorOneAddress.begin(), VectorOneAddress.begin()+VectorOneAddress.size());  
 VectorNZeros.erase (VectorNZeros.begin(), VectorNZeros.begin()+VectorNZeros.size());  
 VectorAddresMaximum.erase (VectorAddresMaximum.begin(), VectorAddresMaximum.begin()+VectorAddresMaximum.size());  
   
   
   
 }  
   
 }  
   
}

#### Calculate number of immediate neighbours of type 1 for each cell of type 0

counter2 records the number of neighbours of type 1. VectorZeroAddress records location of each cell that has more than one neighbour of type 1. VectorNOnes records the number of such neighbours.

vector<int> VectorZeroAddress;  
vector<int> VectorNOnes;  
  
// Find places in MatrixSt that are 0 and have   
// neighbours that are 1.  
// Store the place and how many neighbours are 1 in VectorNOnes.  
// neighbourhood is -1, 0, +1.  
for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 if (MatrixSt[i][j]==0) {  
 int counter2=0;  
 for (int k=-1+i; k<2+i; k++) {  
 for (int l=-1+j; l<2+j; l++) {  
 if (k>=0 && k < iSize && l>=0 && l<jSize && MatrixSt[k][l]==1)  
 counter2++;  
 }  
 }  
 if (counter2 !=0) {  
 VectorZeroAddress.push\_back(MatToVet(i,j));  
 VectorNOnes.push\_back(counter2);  
 }  
 }  
 }  
 }  
  
//############################

Since VectorNOnes and VectorZeroAddress are vector<int>, we need ##### includes +=

#include <vector>

#### Calculate number of immediate neighbours of type 0 for each cell of type 1

counter2 records the number of neighbours of type 0. VectorZeroAddress records location of each cell that has more than one neighbour of type 0. VectorNOnes records the number of such neighbours.

vector<int> VectorOneAddress;  
vector<int> VectorNZeros;  
  
// Find places in MatrixSt that are 1 and have   
// neighbours that are 0.  
// Store the place and how many neighbours are 0 in VectorNZeros.  
// neighbourhood is -1, 0, +1.  
  
for (int i=0; i<iSize; i++) {  
 for (int j=0; j<jSize; j++) {  
 if (MatrixSt[i][j]==1) {  
 int counter2=0;  
 for (int k=-1+i; k<2+i; k++) {  
 for (int l=-1+j; l<2+j; l++) {  
 if (k>=0 && k < iSize && l>=0 && l<jSize && MatrixSt[k][l]==0)  
 counter2++;  
 }  
 }  
 if (counter2 !=0) {  
 VectorOneAddress.push\_back(MatToVet(i,j));  
 VectorNZeros.push\_back(counter2);  
 }  
 }  
 }  
 }  
  
//############################

Since VectorNZeros and VectorOneAddress are vector<int>, we need ##### includes +=

#include <vector>

#### Find the cells with the maximal number of neighbours of type 1

vector<int> VectorAddresMaximum;  
  
// Among VectorNOnes, find those that are equal to the maximum,  
// and put those on VectorAddresMaximum.  
// Notice that max\_element is computed every step of the loop,  
// though it could be computed just once.  
for (unsigned int i=0; i < (VectorZeroAddress.size()); i++) {  
 if ( VectorNOnes[i] == \*max\_element( begin( VectorNOnes), end( VectorNOnes) ) ) {  
 VectorAddresMaximum.push\_back( VectorZeroAddress[ i]);  
 }  
}

#### Find the cells with the maximal number of neighbours of type 0

vector<int> VectorAddresMaximum;  
  
// Among VectorNZeros, find those that are equal to the maximum,  
// and put those on VectorAddresMaximum.  
// Notice that max\_element is computed every step of the loop,  
// though it could be computed just once.  
  
for (unsigned int i=0; i< (VectorOneAddress.size()); i++) {  
 if( VectorNZeros[i] == \*max\_element( begin(VectorNZeros), end(VectorNZeros) )) {  
 VectorAddresMaximum.push\_back( VectorOneAddress[ i] );  
 }  
}

Since VectorAddresMaximumis vector<int>, we need ##### includes +=

#include <vector>