**StemCellSim instructions manual**

This document is a manual for using the StemCellSim program. Here we provide detailed explanations for simulating data and inferring model parameters using ABC from either real or simulated data. We also provide implementation details of our algorithms at the end of the document.

**StemCellSim class**

StemCellSim is a class implemented in c++. The StemCellSim class consists of two files: StemCellSim.h and StemCellSim.cpp. StemCellSim.h contains the class definition and StemCellSim.cpp contains the implementation of its member functions. To carry out simulation, the user must write a main.cpp file and use the functionality of the StemCellSim class by declaring a StemCellSim object and calling its member functions. Example main files for simulating data and inferring model parameters from real and simulated data are provided with the StemCellSim.zip file (found on http://gitlab.com/hormozlab/stemcellsim. Also, see Demo for instructions on using them).

**Basic member functions**

There are three basic member functions of StemCellSim which are used by other member functions to carry out ABC. The first basic member function is:

1. bool StemCellSim::runClonalExpansion(double s, int N, int g, int L, double x)

The function parameters s, N, g, and L of runClonalExpansion() correspond to the parameters of the WF model, and the function parameter x corresponds to the level of feedback in the WF model (see Supplement for model details). runClonalExpansion() simulates the number of mutant cells as a function of time by performing a series of g binomial draws, where the parameters of each binomial draw depend on the number of mutant cells at the current generation. The number of mutant cells at each generation are stored internally in the StemCellSim class in the array cloneSize\_simulated. runClonalExpansion() also stores the value of the function parameter g internally as g\_simulated to hold the length of the cloneSize\_simulated array.

The second basic member function is:

1. double StemCellSim::generateLTT(double mutationRate)

generateLTT() uses the number of cancer cells as a function of time generated by runClonalExpansion() and stored in cloneSize\_simulated to simulate the genealogical history of a random sample of k cells, add the number of mutations from a Poisson distribution to the tree using the function parameter mutationRate given in mutations per generation, average the tree (pseudocode for our averaging algorithm is in Methods), and construct the LTT curve (see ABC schematic in SI Figure 22b-e). After the tree is constructed, the coalescence times in number of mutations (i.e. the number of mutations from the present to the point of coalescence) of the averaged tree are stored in the vector LTT\_sim. Knowing the coalescence times of a tree is equivalent to knowing its LTT curve, since the vector element i = 1 corresponds to the coalescence time of k lineages, i = 2 to the coalescence time of k – 1 lineages, etc., up until the last coalescence event which corresponds to the time of common ancestry of the sample. i = 0 always holds the value 0, a technical detail that simplifies the algorithm used in the function computeEpsSimAndData() (described later in the document).

Note that k, the number of randomly sampled cells, is a StemCellSim private variable, and should be initialized before generateLTT() is called (The StemCellSim functions that call generateLTT() always either ask the user to specify k, or determine k implicitly. The possibility of misusing generateLTT() would thus never be an issue unless the user decides to create a new member function).

generateLTT() also returns the total length of the tree in number of mutations from the present to the birth of the patient. This return value is used when producing simulated tree data. Simulated tree data includes both the coalescence times of the tree, along with the total length of the tree in number of mutations (the return value) because it is necessary to know the total tree length to estimate the mutation rate when carrying out ABC on the data tree.

Implementation details of the algorithm used in generateLTT() are at the end of the document.

After storing the LTT curve of a simulated tree as coalescence times in LTT\_sim, we can find the area between the LTT curve in LTT\_sim, with the LTT curve of a data tree that has been read into StemCellSim (see sections ABC inference on simulated data and ABC inference on real data) and its coalescence times stored in LTT\_data. This is accomplished with the third basic member function:

1. double StemCellSim::computeEpsSimAndData()

The return value of computeEpsSimAndData() is the area between the LTT curve from ABC (LTT\_sim) and the LTT curve from the data tree (LTT\_data). Implementation details of the algorithm are at the end of the document.

Note that computeEpsSimAndData() does not have any function parameters. This is because both LTT curves are stored internally.

**Simulating data**

To simulate data, we call the member function

void StemCellSim::simulateDataAndStoreObservation(double ageOfIndividual, int numOfLineages, double mutationsAccrued, double model\_x)

where the function parameters denote:

ageOfIndividual = age of patient

numOfLineages = number of mutant cells will sample from clonal expansion

mutationsAccrued = age of patient in mutations (This is equivalent to knowing the mutation rate in mutations per year, since the mutation rate is mutationsAccrued/ageOfIndividual)

model\_x = specifies the amount of feedback in the WF model (x = 0 corresponds to the usual WF model without feedback included)

simulateDataAndStoreObservation() first draws parameter values from prior distributions. runClonalExpansion() is called repeatedly with those parameter values until a clonal expansion that does not go extinct is produced and stored in cloneSize\_simulated, and the corresponding parameter values are stored internally in StemCellSim afterwards. generateLTT() is then called, and uses that clonal expansion along with the mutation rate in mutations per generation as input (computed empirically as mutationsAcrrued/(L - 1), where L was drawn from the prior) to produce an LTT curve.

The prior distributions that the parameter values are drawn from in simulateDataAndStoreObservation() can be easily changed to produce trees with fixed parameter values, or with different priors (See code in StemCellSim.cpp). This was done in Methods to test the robustness of our model assumptions.

When calling simulateDataAndStoreObservation(), the tree data is stored internally and should be printed to a text file afterwards. We can accomplish this by calling the printData() function, which takes a file name as input and stores the tree in a text file with that file name. The output file has the following format:

    g = 21

    L = 34

    g/L = 0.617647

    N = 17415

    n = 70

    s = 0.233745

    age = 34

    k = 22

    trajectory: 1,2,3,3,8,12,11,10,16,12,13,18,24,25,28,29,32,39,50,72,70,

    LTT: 20.782,20.782,37.5372,42.1257,43.011,43.011,56.5434,57.0947,57.0947,

    61.9103,67.7438,79.4597,79.4597,79.4597,102.075,108.873,193.097,

    199.235,299.869,300.742,354.301,

    mutationsAccrued: 706.301

The line that begins with “LTT” contains the coalescence times of the tree in mutations, delimited by “,”. The line that begins with trajectory contains the number of cancer cells at each generation, also delimited by a “,”. The line that begins with mutationsAccrued contains the total length of the tree in number of mutations. The line that beings with k contains the number of sampled mutant cells, and the line that begins with age contains the age of the patient for the simulated tree. The remaining lines are the parameter values that produced the data tree. Tree data in this format can be read into the StemCellSim object and stored internally to carry out ABC on the data tree.

**ABC on simulated data**

ABC is run on simulated tree data by first reading it into the StemCellSim object using

void StemCellSim::readParamFile(std::string paramFile),

where the parameter paramFile is the name of the text file storing the tree data in the format shown in the previous section. The tree data is then stored internally in the StemCellSim object, where the coalescence times are stored internally in LTT\_data, and the remaining tree data and parameter values are stored internally in variables that have “\_data” appended to them (see private variables in StemCellSim.h). Only LTT\_data is used by ABC for inference, and the parameter values that produced the tree are stored internally only for convenience. The inference is carried out by calling the following function repeatedly in a for loop:

void StemCellSim::samplePosterior(double eps, double model\_x,

const std::tuple<std::string, std::string,

std::string, std::string, std::string,

std::string, std::string, std::string> & outputFiles)

The samplePosterior() function samples a point from the posterior distribution. samplePosterior() works by drawing model parameter values from prior distributions. It then calls runClonalExpansion() in a loop until a clonal expansion that does not go extinct is produced, and then calls generateLTT() to produce an LTT curve. samplePosterior() then calls

double StemCellSim::computeEpsSimAndData()

computeEpsSimAndData() returns the area between the data LTT curve and the simulated LTT curve. If the returned area is larger than the function parameter eps, samplePosterior() repeats itself. If the returned area is smaller, the parameter values drawn from the priors are retained and printed to their corresponding text file, given by the components of the function parameter outputFiles, which is a tuple. The components of the tuple outputFiles correspond to the retained s, n, g, L, the trajectory of the clonal expansion, the LTT curve of the tree, and the area between the LTT curves respectively.

The function parameter model\_x in samplePosterior() determines the level of feedback the WF model uses for ABC. In general, model\_x in samplePosterior() is zero since we usually are only interested in determining how well we can infer the parameter values from data trees simulated under different levels of feedback by using model\_x = 0 for ABC.

Similar to how the prior distributions for simulating data trees are easily changed by the user, the prior distributions for ABC are also easily changed in samplePosterior() (see code in StemCellSim.cpp)

**ABC on real data**

ABC on real data is identical to carrying out ABC on simulated data trees, except that instead of using the readParamFile() function to read data, we use the function

void StemCellSim::readDataAndStoreLTT(std::string newickFile, double ageOfIndividual)

instead. The function samplePosterior() is then called in an identical manner to the previous section on ABC inference from simulated data trees to produce points sampled from the posterior distribution. The function parameter newickFile in readDataAndStoreLTT() specifies the name of the file where the data tree is stored, and the function parameter ageOfIndividual specifies the age of the patient that corresponds to the data tree. For real tree data, the file containing the tree data must have a different format to that of simulated tree data. In particular, it must contain a tree in Newick format that looks as follows:

((((MPP-39:391,HSC-10:364):9,((HSC-32:373,HSC-37:316):2,HSC-35:365):12):33,((HSC-51:353,HSC-47:345):4,HSC-46:300):34):117,(((((HSC-45:338,HSC-48:355):9,(((MPP-48:291,HSC-5:304):10,HSC-50:323):37,HSC-30:361):2):20,(((MPP-22:243,HSC-49:281):46,HSC-34:279):66,MPP-34:313):40):103,(MPP-1:480,HSC-39:488):36):11,(HSC-6:423,HSC-33:454):57):2):219;

Take note of number 219 before the ‘;’ at the very end. 219 is the number of mutations from the birth of the patient to the most recent common ancestor of the sampled cells. It is very important to specify this in the tree since ABC uses the number of mutations accrued from birth of the patient to the present (the total length of the tree in mutations) to estimate the mutation rate in mutations per generation when calling samplePosterior() by dividing the number of mutations accrued by L – 1, where L was drawn from the prior in samplePosterior().

The function parameter ageOfIndividual must also be specified to determine the default prior on L when calling samplePosterior(). The default prior assumes 1 generation per year so that L = ageOfIndividual + round(Gaussian noise), where the Gaussian noise has mean 0 with an std of roughly 15% of the ageOfIndividual to account for uncertainty, and where round() is used because L must be an integer.

After reading the Newick format tree, readDataAndStoreLTT() performs the averaging algorithm on the tree and converts it to an LTT curve, and the LTT\_curve is stored as coalescence times in LTT\_data. samplePosterior() is then called to infer the parameter values from LTT\_data. The averaging algorithm in readDataAndStoreLTT() is the same one used in generateLTT() but the implementation is different.

**Implementation details of the averaging algorithm**

This section provides details for the implementation of our averaging algorithm described in Methods.

Our averaging algorithm is used in the generateLTT() function and in the readDataAndStoreLTT() function, and both are implemented differently. Recall that generateLTT() is used when simulating data trees, and when carrying out ABC. readDataAndStoreLTT() on the other hand is used when reading a real data tree from a text file in Newick format.

The algorithm implemented in generateLTT() begins with a random sample of k cells from the final generation. Those cells choose parents from the previous generation at random, one at a time. Mutations are assigned to the edge associated with each pick so that the mutational process is implemented simultaneously with the genealogical process. For each pick, a coalescence event may or may not occur. If at any point a coalescence event occur, the length in number of mutations of the left and right subtree of the point of coalescence (the parent chosen by both cells) are immediately averaged, and the length of the subtrees are both scaled so that they equal the average. The coalescence times of the subtrees are kept in vectors, and so in practice we are actually scaling the coalescence times of the subtrees, rather than branches. After scaling the coalescence times of both subtrees held in vectors, the two vectors are combined into one vector to represent a new averaged subtree. We therefore always begin with k vectors, and after the final coalescence event recover one vector which corresponds to the coalescence times of the averaged tree of the random sample.

The algorithm implemented in readDataAndStoreLTT() works by first reading the Newick format tree into a string. A vector of vectors called array is then declared, and an empty vector is pushed into the back. The vectors within the vector represent subtrees, and contain the coalescence times of subtrees. The algorithm then loops over every element of the string containing the Newick tree. At each iteration, it checks the current string element. If the current string element is “:”, the algorithm pushes the numbers that come after it to the back of the last vector in array. If the current string element is “,”, the algorithm pushes an empty vector to the back of array. Lastly, if the current string is “)”, the algorithm does the following: it first takes the last and penultimate vector in array, which contain the coalescence times of subtrees, and scales the coalescence times so that the length of both subtrees equals the average length between the two. After this, the last and penultimate vectors are combined to form one vector forming a newly averaged subtree. Lastly, it checks to see if the string element that comes next is “:” followed by a number. If so, the algorithm pushes that number to the back of the last vector in array, which corresponds to the newly averaged subtree. After these 3 conditions are checked, the loop proceeds to the next iteration.

These algorithms are statistically equivalent to the pseudocode in the methods only if more than two coalescence events never occur simultaneously. For example, suppose 3 lineages coalesce simultaneously. Then averaging lineage 1 and 2, and then averaging with lineage 3 may produce different coalescence times than averaging lineage 2 and 3 first, and then averaging with lineage 1. In this case, the averaging algorithm would depend on which node it starts on. More than 2 coalescence events rarely occur, and if they did, either averaged tree would produce a good estimate of the coalescence times. We therefore believe the contribution to the statistics is negligible and can be safely ignored. Furthermore, the implementation of the algorithm on tree data is equivalent to the implementation of the algorithm on trees generated by ABC. Therefore, any contribution to the statistics from multiple coalescence events occurring simultaneously would be expressed as uncertainty in the error bars.

**Implementation details of the algorithm for computing the area between two LTT curves**

Whenever simulated or real data tree is read in and stored internally in StemCellSim, the LTT curve of the data tree is stored in the LTT\_data vector as coalescence times. Whenever a tree is generated by ABC, the LTT curve of the simulated tree is stored in the vector LTT\_sim as coalescence times. The coalescence times are in other words a representation of the LTT curves of a tree, since the vector elements from i = 1, … , k – 1 correspond to the time in number of mutations at which k, k – 1, … , 2 lineages coalesce respectively. Recall that to simplify the algorithm, the 0th element of LTT\_data and LTT\_sim both hold the number zero.

To compute the area between the LTT curves, we apply an algorithm that uses the coalescence times stored in LTT\_sim and LTT\_data. The algorithm first initializes two int variables to equal one called counterSim and counterData, where counterSim corresponds to LTT\_sim and counterData corresponds to LTT\_data. The algorithm then incrementally computes the area by implementing the following pseudocode:

Let A = 0

do{

1. Let x1 be the min of the two coalescence times LTT\_sim[counterSim] and LTT\_data[counterData], and let y1 be the number of lineages that exist immediately before x1.
2. Let x0 be the max of the two coalescence times LTT\_sim[counterSim - 1] and LTT\_data[counterData - 1], and let y0 be the number of lineages that exist immediately after x0.
3. A += (y1 – y0)\*(x1 – x0)
4. If counterSim minimized the two coalescence times in 1), then increment counterSim. Else, increment counterData.

while(both lineages are not 1)

return A;

There are two important difference between the actual implementation to the pseudocode. In the implementation, we instead do A += [ (y1 – y0)\*(x1 – x0) ]/LTT\_data[LTT\_data.size() - 1]. In other words, we divide the area by the number of mutations from the present until the most recent common ancestor of the sample corresponding to the data tree. Intuitively this is like finding the percent difference between the trees. The second difference is that the implementation differs slightly in terms of technical detail. In particular, extra care must be taken into account when adding area in the LTT plot where lineages have coalesced to one (the full implementation can be found in StemCellSim.cpp).