MergeSplit - Documentation

Gokul G. Nair
Indian Institute of Science

September 11, 2017

This document attempts to elucidate the algorithms, data structures, and functions involved in the program 'MergeSplit', written in C++11, which is used to simulate the Stochastic 'Merge-Split' process for heterogeneous populations.

1 Dependencies

To compile the code, a C++11 compiler is required. The program uses the following libraries:

`iostream, iomanip, vector, cstdlib, chrono, random, algorithm, ctime, fstream, string`

2 Classes

There is only one class apart from those already defined in libraries.

`class MergeSplit(N,N1,Ps0,d,Pm,s)`

Parameters and Attributes:

1. `int N:`

   Total population to be considered for the simulation. eg: N= 10,000

2. `int N1:`

   The number of individuals of type-1 (species-1) in the population. eg: N1= 5,000

3. `double Ps0:`

   Baseline (homogeneous) split-rate. A homogeneous group (comprising exclusively one species) would split intervals distributed exponentially with parameter Ps0, i.e. t $\sim$ exp(Ps0).

4. `double d:`

   This parameter introduces a difference in split rate depending on the heterogeneity of the group. Split rate is calculated using the formula $Ps(n,k) = Ps0 + \frac{k(n-k)}{n^2}d$, where n is the size of the group, and k is the number of individuals of type-1 in that group.

5. `double Pm:`

   Merge rate; groups have merge events in time intervals distributed as t $\sim$ exp(Pm).

6. `int s:`

   The number of sites available to occupy. In the mean field approximation, individuals occupying the same site form a group. Usually s is kept equal to N, the total population.

7. `vector<vector<int> > counter:`

   This structure is a dynamic 2 dimensional array, a vector of vectors. The number of groups of size n with k individuals of type-1 is registered in the n^{th} row, and k^{th} column. The structure is dynamic, that is, trailing cells that are not in use are deleted, and re allocated if they are back in use. This helps speed up iteration and frees up a lot of memory.

Methods:

1. `MergeSplit::MergeSplit(N,N1,Ps0,d,Pm,s):`

   Constructor. The counter starts with a random distribution, which is obtained using the function `random_distribution`.

2. `void MergeSplit::next_event():`

   This method calculates the net split (`split_rate`) and merge rates (`merge_rate`) and chooses an event using a Bernoulli distribution. This function uses the fact that the minimum of n exponential random variables is exponential with a parameter equal to the sum of the individual parameters.

   `split_rate` is calculated by adding up the split rates of each group (except those of size equal to 1) calculated using the function `het_split_rate`.
   `merge_rate` is calculated in a similar way except that groups of maximum size are excluded. We also multiply this value by a fudge factor which is the ratio of total number of groups (calculated by the method `get_group_count`) to the total number of sites. This fudge factor arises because the probability of a merge event has to increase with the number of occupied sites.

   The probability of a split event is given by Ber(`split_rate/split_rate + merge_rate`). The merge event is the complement of the split event. Depending on the outcome of the Bernoulli distribution one of the functions among `split` or `merge` is called.
   Once control is back to this function, `clean_counter` is called.

3. `void MergeSplit::get_population():`

   This function computes the total population by taking the sum of products of group size and the corresponding count for that group size. This function is used only for testing purposes to ensure conservation of total population throughout the span of the simulation.

4. `void MergeSplit::get_group_count():`

   This function computes the total number of groups by adding up all the entries of `counter`.

5. `void MergeSplit::split():`

   This function decides the group that splits and the location of the split. The first step is to weight the `counter` with split rates. This is done by making a deepcopy[1] of `counter`, and then multiplying the entries by the corresponding split rate. Now a group (n,k) is split with a probability proportional to the (n,k) entry of the weighted-counter (C_w). This can be achieved using the function `custom_distr`. First n is chosen by using a probability mass function

   $$P(n) = \frac{\sum_{i=k_{min}(n)}^{k_{max}(n)} C_w(n,i)}{\sum_{j=0}^{\infty} \sum_{i=k_{min}(j)}^{k_{max}(j)} C_w(j,i)}$$

   where $k_{min}(j)$ and $k_{max}(j)$ are the minimum and maximum possible number of type-1 individuals in a group of size j.

   Now k is chosen by using a probability mass function:

   $$P(k) = \frac{C_w(n,k)}{\sum_{i=k_{min}(n)}^{k_{max}(n)} C_w(n,i)}$$

   Once a group (n,k) is chosen to be split, the sizes and compositions daughter groups are to be determined, i.e. the number of type-1 and type-2 individuals in the daughter groups. This is achieved using a uniform distribution, ensuring that no daughter group has 0 individuals.

   After this, the count for group (n,k), in `counter` is decremented and the counts for the daughter groups (n_1, k_1) and (n_2, k_2) is incremented.

6. `void MergeSplit::merge():`

   Two groups have to be chosen to merge. As in the split routine, we use a similar probability mass function (except that we don't need to weight the counter). Once the first group is chosen, the count for that group has to be decremented before chosing the other group. The two groups are combined to give a daughter group, and the counter for the new group is incremented. Since the program cleans the counter in regular time-steps (deletion of unused cells in the counter), some resizing of the counter maybe required to accommodate the new, larger group.

All the member variables and functions of the class `MergeSplit` have been described in this section. Before proceeding to the descriptions of the main and auxiliary functions, a main point has to be addressed which is hashing.

3 Hashing

For the memory and time efficiency of the program, the `counter` in any object of `MergeSplit` is dynamic, i.e. its size and shape do not remain the same throughout. For example, any trailing empty slots in the counter are deallocated. Also, for large groups (in comparison to the total population size), it is inefficient to maintain counters for all group compositions (number of type-1 individuals). Therefore we need to map the counter indices (that always start from 0) to the group

[1] A copy that preserves the 2d array structure.

composition, 'k'.
e.g: For a population of N=100, N1=50, a group of size n=65 cannot have anywhere between 0-14 individuals of type-1 (since that would require more that 50 type-2 individuals to make 65), i.e. $k \geq 15$. So instead of numbering k from 0, we number it from 15. Also the group cannot have more that 50 individuals of type-1, so we stop numbering at k=50. In the counter however the numbering is k' from 0 to 35.

The hash function, `hash_counter` maps k to k' and vice-versa, depending on the state of a boolean parameter.

4 Main

`main` takes user input for the parameters: `s`, `N`, `N1`, `Ps0`, `Pm`, `d`, and `events_max`, the last one being the total number of events before the simulation stops. An object (henceforth referred to as `m`) of the class `MergeSplit` and initiates `group_counter`, a 2d array (The entries of this counter are of type `uint64_t`, 64-bit integers to accommodate large counts) that samples the `m.counter` at regular intervals. `group_counter` keeps track of the number of instances of a group of size n, and composition k. There is also a variable called `sampling_interval`, which is the interval at which sampling is done. Three file names are also declared here, one for printing the total group counts into, one for printing the group frequencies into and another that serves as a temporary back up. The first two files are printed into after the simulation, but the group count is printed into the third file at regular intervals throughout the simulation, and serves as a back up in case the program is interrupted.

The main timer is a `for` loop running from `0` to `events_max`. Inside this loop there are three `if` conditions: one for writing the back up file, one for displaying progress on the terminal, and a third for sampling `m.counter`. The third `if` is entered only when the timer is greater than 10%[2] of `events_max` and a multiple of `sampling_interval`. After sampling, `group_counter` is 'cleaned' using `clean_counter`. Due to this, every time sampling is done the program may need to resize `group_counter`.

The last step in each iteration of the main timer is the calling of the member function, `next_event`. The last few lines of `main` involves initialising a frequency counter that converts group counts into frequencies by dividing them by total number of groups that were counted. The contents of the two counters are then printed into files using the function `write_into_file`.

5 Auxiliary functions

All the other functions in the program will be explained in the section.

1. `int get_index(vector<int> x,int key):`

   Given a vector and a key, the function returns the index of the key's first occurrence in the vector.

2. `void clean_vector(vector<int>& x,int key=0):`

[2] 10% of the time is given to the system so that it may reach steady state, before sampling.

Given a vector, the function removes all the trailing elements that are the same as `key` (which by default is 0). It does so by popping the vector until the last element is not the same as `key`.
There is a copy of this function for vectors whose entries are `uint64_t`.

3. `void clean_counter(vector<vector<int> >& x):`

   This function takes in a counter, which is has 2d array-like structure (a vector of vectors) and applies `clean_counter` to each row. The function also removes all the trailing empty rows.
   There is a copy of this function for counters whose entries are `uint64_t`.

4. `vector<int> seq(int start,int end,int step=1):`

   Generates a vector containing a sequence starting at `start` and ending at `end` in steps of size `step`. The function only works for increasing sequences.

5. `int hash_counter(int N,int N1,int i,int j,bool unhash = false):`

   As explained in the section about Hashing, this function maps the composition of a group (number of type-1 individuals) to an appropriate index in the counter. If the boolean `unhash` is `true`, then it does the reverse mapping.

6. `void print_vec(vector<int>& x):`

   Since C++ does not provide a direct way to print vectors, this function is used to print them. The function iterates throught the elements of the input vector and prints them one by one.
   There is a copy of this function that prints vectors whose entries are `double`.

7. `void print_vec_of_vec(vector<vector<int> >& x):`

   For similar reasons as the ones stated above, a function is required to print the entries of a counter, a vector of vectors.
   There are two more copies of this function that print vectors whose entries are `double` and `uint64_t`.

8. `void random_distribution(vector<vector<int> >& counter,int N,int N1,int s, bool print=false):`

   This function is used only once while initialising the `MergeSplit counter`. It works by creating a 2d array of shape $s \times 2$, s rows representing the number of sites, and the two columns representing the number of type-1 and type-2 individuals at every site. Now, N individuals of which N1 are type-1 are placed one-by-one, randomly (unniformly) into these sites. The vector is then iterated over and for each occurrence of a group of size (n,k) the corresponding count is increased in the empty counter that is passed into the function. This produces a random distribution. If the boolean parameter `print` is `true`, the newly produced random distribution is printed.
   This function is essential to show that the steady state distribution is independent of the initial conditions.

9. `vector<vector<double> > vector_deepcopy(vector<vector<int> > x):`

   This function makes a copy of the input counter and returns it to the caller. It is called a deepcopy because it copy preserves the structure of the original counter.

10. `int custom_distr(vector<double> x,int start = 1):`

    Simulates a probability mass function dictated by the input vector `x`,

    $$P(i) = \frac{x[i]}{\sum x[i]}$$

    . The function works by creating a vector that is a cumulative sum of the `x`. The last element of this vector is the total sum of the elements of x. It then simulates a uniform random variable $U(0,1)$ and then compares it to the elements of the cumulative sum divided by the total. The index of the first element of the cumulative that exceeds the random number which is a realisation of the probability mass function described above, is returned by the function. The function has a copy that takes `int` vectors as input.

11. `vector<double> row_sum(vector<vector<double> > x):`

    Returns a vector whose n^{th} element is the sum of all the elements in the n^{th} row of the vector of vectors that is passed as an argument.
    This function has a copy that is compatible with counters whose elements are integers.

12. `double het_split_rate(double Ps0,double d,int n,int k):`

    Returns the split rate of the group of n individuals, of which k are of type-1 given by the formula $Ps(n,k) = Ps0 + \frac{k(n-k)}{n^2}d$.

13. `double sum_vector(vector<double>& x,int start,int end):`

    Calculates the sum of all the elements of a vector. There is an integer counterpart to this function.

14. `int longest_group(vector<vector<int> >& x):`

    Returns the row number of the longest row in a counter/vector of vectors. Has a `uint64_t` counterpart. This function is useful while writing the data from a counter into a file,

15. `int rand_int(int low,int high):`

    Simulates a uniform distribution in the range [low,high].

16. `bool ber(double p):`

    Simulates a Bernoulli random variable with parameter p.

17. `vector<int> randomly_split(int n,int k):`

    This is a function that was in use in the older versions of the program. It takes a group of size n with k type-1 individuals randomly arranges the members of the group on a line and splits the group at a random location on the line. This mechanism of splitting is not used in the simulation.

18. `const string currentDateTime():`

    Return the current date and time formatted as a string. Used in naming files produced by the program as output. Prevents overwriting of previously written files.

19. `void write_into_file(string file_name1,vector<vector<uint64_t> > group_counter,int N,int N1,int s,double Ps0,double Pm,double d,int events_max):`

    Opens a file by the name of `file_name1`, writes the values of `s, N, N1, Ps0, Pm, d, events_max` and the contents of `group_counter` into the file.

Notes on Random Number Generation:
The program uses random variables from the library `random` with the clock as seed value. `default_random_engine` is the random engine used by the program. The code also uses probability distributions such as Bernoulli and uniform which are part of the same library.

6 Compilation and Output

The source code is written in C++11 and has to be compiled using the appropriate compiler. One can compile the source code using the following line in the terminal:
`g++ -std=c++11 merge_split_xx.cpp -o MergeSplitxx`

The program creates a folder called 'Output' in the same directory that it is in. In the 'Output' directory it creates a sub-directory named: 'events_max_Ps0_Pm_d'. Within this subdirectory the program opens three files: a hidden back-up file named '.groups_distr_time_and_date.csv', a file containing the final steady state counts 'groups_distr_time_and_date.csv' and a file containing the frequencies (distribution) of different groups 'freq_distr_time_and_date.csv'.

On opening one of these files one will see that the first row contains the values of the parameters used for the simulation. The second row contains the group composition (number of type-1 individuals) corresponding to the column directly below it. The first column lists the size of the group corresponding to the row to its right. The rest of the file contains the actual data. In the 'groups_distr' file the data is the total number of instances of a group of that size and composition, while in the 'freq_distr' file it is the frequency.