Generating Simulated SNP Array Data

**Simulating the Tumor Genome**

First, before we can generate simulated data, we must simulate a ‘truth.’ Specifically, we must simulate a tumor, or a tumor genome. Given that we are not thinking of sequencing data we don’t need to simulate the tumor with a granularity at the level of individual bases; rather, we are only interested in simulating ‘true’ copy number states across the genome at the level of genomic ‘segments’, where a segment is defined as a contiguous set of markers (of which there are about 600,000 in the entire genome for this array, or an average of about 20-30,000 per chromosome). Because we want our simulated data to resemble real data as much as possible, rather than simulating marker positions across the genome, we use the marker positions from a real SNP array. This way, the density of markers (or variation in the density of markers) across the genome recapitulates what one might expect to see in actual data.

Next, we divide the genome into segments. We are assuming that, as far as copy number alterations are concerned, genomes can be characterized as consisting of segments varying in length (up to the size of the length of the chromosome on which the segment lay, of course). Each segment has a copy number value associated with it, where segments that are normal have two copies present (we are ignoring the sex chromosomes for now). Again, in order to generate realistic simulations, we have used the number of segments imputed in actual patient samples (usually between 100 and 400 segments in a typical CLL sample’s genome) to determine how many ‘pieces’ to divide the genome into.

Another important point worth noting is that, although if all SNP markers were located at points of heterogeneity we should expect to see one copy of each allele, this is not in fact the case. Many markers (even in normal patient samples) are at homogeneous loci, i.e., there are two copies of one allele. For this reason, in order to make the simulations realistic, we randomly select some fraction of the markers (perhaps a third or half) to be made ‘homogeneous’, meaning that in the normal case, there are two copies of one allele rather than one copy each of two alleles.

So, at this point in the simulation process what we have is a data frame consisting of SNP locations (locations on the chromosome, that is), chromosome numbers (the chromosome on which each SNP is found), a ‘segment number’ (e.g., 2.5 would mean the 5th segment on chromosome 5), number of A-copies and number of B-copies (given that we are still describing a normal genome, A + B equals 2 for all SNPs). From here, we generate the aberrant subclones. This is done by randomly selecting a segment (or multiple contiguous segments) to be assigned a new (non-normal, with equal probability of loss and gain of copy) copy number value. It should be noted that with each ‘iteration’ (really, each new clone generated), the new clone can be generated either by creating an aberration from the normal genome, or by adding new copy number alterations to an existing subclones, so if there are more than two subclones, one subclone can be a direct descendant of another. Also, it should be noted that one can generate a large population of clones, then select only some of them to be used to comprise the simulated ‘tumor’, so as to simulate the effect of clones dying off.

The last element of the ‘truth’ to be simulated is the ‘psi’ vector, which denotes the fraction of tumor cells belonging to each clone. The ‘psi’ values are generated via a K-dimensional Dirichlet distribution (where K is the number of subclones of which the tumor is comprised) with alpha parameter 1.

**Adding Noise**

To make simulated data from a simulated truth, we must add noise. For simplicity’s sake, we first generate what might be called ‘pre-data’ or ‘noiseless data.’ Or, to put it another way, we compute the weighted average allelic copy number across all clones at each marker. Essentially, for each SNP, we take all K ‘A-values’, multiply them by the associated ‘psi values’ and take the sum, then do the same with the B-allele copy number values at each SNP. So, if there are two clones, with 1 copy of A and 2 copies of A, respectively, at a particular allele, while psi1 = 1/3 and psi2 = 2/3, the weighted average A-allelic copy number is 1.667.

Next, we add noise to the weighted average copy number values for each marker. Currently, the noise is generated from a normal distribution with a predetermined value for sigma (usually ~1). This poses the obvious issue that, in theory, the addition of noise could lead to an observed negative copy number value. Since the variance used in these simulations is small enough that only a minute fraction (if any) of SNPs end up with negative copy number values after noise addition, it can be easily dealt with ad hoc without significantly affecting the distribution of the ‘observed’ copy number values.

**Converting Allelic Copy Number into Log R Ration and B Allele Frequency**

Next, we compute the Log R Ratio and B Allele Frequency at each SNP from the noise-added copy number values. The LRR and BAF are what is actually observed in the real data. Finally, for the purpose of analysis, we compute the segment median LRR and BAF for each segment (while also recording the length of each segment), which is also done for actual patient samples.

[Compare plot of actual SNP array data with simulated SNP array data]

[Plot of simulated LRR-BAF plots and corresponding X-Y plot]

**Preparation: Filtering**

Before running one of our algorithms, we must filter the data. This step is important because the vast majority of genomic segments in the segmented SNP array data do not contain useful information about heterogeneity because their copy number status is shared across all subclones. It would therefore be a waste of computational expense to include them in the algorithm. The goal of the filtering step is to eliminate such non-informative segments as thoroughly as possible while retaining all segments at which there is heterogeneity. For this reason, we conduct the filtering so as to allow marginal segments into the filtered data as it is better to let some homogeneous segments through than to lose valuable information by accidentally excluding a heterogeneous segment.

The filtering here is dome by rather simply: we keep all segments whose median copy number values’ distance to the nearest integer is >= epsilon. Typically we would set epsilon equal to .05. The value of epsilon should reflect one’s estimation of the sensitivity of the assay. In this case, we believe any clone smaller than .05 will be effectively impossible to discern from the noise inherent in the data.

[Plot: segment medians, with ‘filtered’ ones colored red (?)]

**Preparation: Parameter Selection**

The parameters of the posterior probability distribution should be fairly robust…

It is also worth noting that each algorithm has its own parameters that determine how it is run…

**Population of Samples to Generate**

-Population characteristics

The simulated tumors possess anywhere from 1 to 6 (?) subclones, with the fraction of cells belonging to each clone determined by a dirichlet distribution. The simulated tumor can be represented by an n X k matrix showing the copy number at each genomic segment for each subclone (which we allow to vary from 0 to 5), a vector of length k of the fractions of tumor cells belonging to each subclone (the vector therefore sums to 1), and a vector of integer values representing the size of each segment in number of markers.

We start out generating a random population of n X kmax matrices and a population of kmax-long psi vectors. We use a matrix and psi set that has already been saved, as it is unnecessary to generate a new one with each iteration of an algorithm. In the first step of the algorithm, these matrices and their associated psi vectors are then filtered according to the sum of their residuals (that is, we compute the N ‘expected’ copy number value, given the copy number matrix and psi vector, and sum the difference between the expected values and the data vector). This filtering can be done either by keeping all copy number matrix-psi vector pairs whose residual sum is less than some value, or by keeping those belonging to the top N residual sums. These ‘kept’ matrices and psi vectors then constitute the starting population for genetic algorithm, while the matrix and psi with the lowest residual is the starting point for the EM algorithm and the customized algorithm.

**Assessing Accuracy**

There are several ways to assess the accuracy of the imputed clonal architecture of the tumor. The most basic way is to compute the absolute accuracy of clonal copy number assignments across all genomic segments. Essentially, the metric would equal the total number of accurate copy number assignments across the ‘metagenome’ divided by the total number of assignments to be made (which equals the product of the number of segments, the number of subclones, and the number of possible alleles, which we assume to be 2). This metric is the strictest metric. It suffers from the shortcoming that it does not measure the severity of an assignment error. For example, if clone A has 0 copies of an allele at a given segment, and an algorithm imputes 1 copy, this error is less severe than if it were to impute 4 copies. The absolute accuracy metric considers both errors to be equal.

Another problem with the simple version of the absolute accuracy metric is that it can be misleading if not considered in the context of the right ‘null hypothesis.’ For example, even for most cancer genomes, if one assumes the entire genome to be normal in terms of copy number, one will still consistently get as much as 80-90% or higher of the segments accurate. In this light, an accuracy rate of 80%, though superficially much better than one would expect ‘by chance’ is not necessarily particularly impressive.

A more sophisticated measure of accuracy is what we call the weighted Euclidean distance.

**Algorithm Testing Strategy**

Algorithm classes:

-EM

-MCMC (segment-wise?)

-Genetic Alg.

-Unnamed/custom 4th algorithm

**Code Structure**

Simulations are generated by a function called ‘simulate’, which takes as arguments a set of parameters that determine the following: the true psi values (which also implies the number of subclones), … The function’s output consists of simulated SNP array data (i.e., the LRR and BAF values for the ~600,000 markers); the segmented data (that is, the median LRR and BAF values for each segment, as well as the converted ‘X’ and ‘Y’ imputed allelic copy number values, and the number of markers in each segment); the true clonal/allelic copy number values for each segment (in a 2\*k X n matrix, where there are n segments and k subclones); and lastly the (k) true psi values.

Each of the algorithm functions is written to take in as arguments the segmented SNP array data and a host of parameters dictating how the algorithm should function. For example, the EM algorithm takes the following parameters: .

Though the parameters vary from one algorithm to the next, the output of each algorithm has the same format. Each contains the following objects: 1) a list of arrays (called ‘arrays’) containing… 2) a list of final (or best) parameter picks for the psi vector and copy number matrix; and 3) the runtime of the algorithm. The algorithms can each be run through a generic function (which we call runAlg) that takes the name of an algorithm and a set of parameters that the particular algorithm should take as arguments. This function then puts out an object containing the results.

The results of each algorithm follow the same template, which is an S4 class containing the following slots: .

Lastly, we have created a function called ‘eval’ which evaluates the performance of an algorithm on a data set and illustrates performance via several plots... To demonstrate, we ran eval on a list containing results from the EM algorithm on three simulated test cases.

(plot)

Since are also interested in assessing and comparing the performance of the various algorithms, the ‘eval’ function can be used to generate metrics (and plots) for a given data set to allow for comparison of accuracy or rapidity of convergence between algorithms. (plot)

**Current Issues:**

There are several current issues with algorithm performance inhibiting the algorithms from converging to the global maximum.

1. GA: Local optimum convergence?
   1. Still some tendency to converge to apparentl local optima near the global optima

2. MCMC: posterior probability for matrices are exceeding the posterior for the true matrix/psi combination.

3. EM: Posteriors are better than true posterior. On looking at the expected cn vals, they look to be genuinely closer to the data than what we’d expect from the true params.