User Manual, Version 1.3

Sara Sheehan*†, Kelley Harris*, Yun S. Song†

January 10, 2014

The diCal (Demographic Inference using Composite Approximate Likeilhood) software can be used to estimate the effective population sizes of a population based on haplotype sequence data. It can be used to compute the conditional sampling probability of a haplotype, given a set of previously sampled haplotypes and a population size history. Various hidden Markov model decodings can also be computed. This code implements the method described in "Estimating variable effective population sizes from multiple genomes: A sequentially Markov conditional sampling distribution approach" [5]. Some of this code is based on the implementation of the method described in [4].

1 License

diCal is distributed open source under the FreeBSD (Berkeley Software Distribution) license. See `license.txt` for details.

2 Installation and Quick Start

First unzip `diCal-v1.3.tar.gz` and go into the folder `diCal-v1.3`. diCal has three jar-file dependencies, which must be downloaded and placed in the folder `diCal_lib` (due to the java classpath):

- Java Simple Argument Parser v2.1, `http://www.martiansoftware.com/jsap/`, JSAP-2.1.jar
- Java Matrix Package v1.0.2, `http://math.nist.gov/javanumerics/jama/`), Jama-1.0.2.jar
- Apache Commons Math v2.2 (for the Nelder-Mead optimization routine), `http://commons.apache.org/proper/commons-math/`, commons-math-2.2.jar (only the jar file is needed in the `diCal_lib` folder)

Note that the last jar is not the most recent Apache Commons Math version, but it contains a version of the Nelder-Mead optimization routine that is easy to use. The Java Matrix Package is also not the most recent version; diCal requires the version specified above. This code was developed using Java Runtime Environment (JRE), version 1.6.

To run the program on the example dataset and parameters, try out the command:

```
java -jar diCal.jar -F ex/data.fasta -I ex/params.txt -n 4 -p "3+2+2+3" -t 2 -N 1
```

After a few minutes, this should produce very similar output to the file `ex/output.txt`. See below for a full descriptions of the program options, which have changed since version 1.1 to be more consistent with PSMC [2] options. They can be displayed using the `--help` command.

*These authors contributed equally to this work
†To whom correspondence should be addressed. E-mails: ssheehan@eecs.berkeley.edu and yss@eecs.berkeley.edu

3 Required Parameters

There are two required parameters, the input fasta file(s), and the input text file of parameters. Alternatively, version 1.3 supports a simplified VCF file and a reference fasta file instead of fasta files of haplotypes. The converter `stripVcf.py` will convert a VCF file into the appropriate format (it will create a new file).

- `-F / --fastaFiles`: the path to the input fasta file(s).

  Within each fasta file, all haplotypes must be phased, of the same length, and on a single line. Each chromosome or scaffold should be in a separate file. To specify multiple files on the commandline, use quotes, such as:

  ```
  -F "chrom1.fasta chrom2.fasta"
  ```

  Missing bases are supported (denote with N/n); all other bases should be A/a, C/c, G/g, or T/t. For accurate population size estimates, it is recommended that haplotypes be least 1Mb long. Currently binning of haplotypes is not supported.

- `-R / --referenceFile`: the path to the reference fasta file.

  Alternatively, a reference file (fasta format) and a simplified VCF file can be specified. The labels of the chromosomes/scaffolds in the reference file should match those of the VCF, and the indexing in the VCF file should match the lengths of the corresponding chromosomes. Note that reference sequences across multiple lines will take a while to read in, but this only needs to be done once per run.

- `-V / --strippedVcfFile`: the path to the stripped VCF file.

  To create a stripped VCF file from a regular VCF file, use the provided python script:

  ```
  python stripVcf.py -v input_file.vcf -o output_file.txt
  ```

  The stripped VCF format contains three columns, the first for the chromosome name (string), the second for the variant base within the chromosome (indexed from 1, like VCF), and the third for the haplotype alleles. The data should be phased. Heterozygous unphased sites will be represented as two unknown bases. Many unphased bases will make the inference less accurate. An example line from a stripped VCF file would look like:

  ```
  chrom1 395049 ACAAAACAAA
  ```

  The python converter was designed for VCF version 4.1, but can be modified for different versions.

- `-I / --paramFile`: the path to the input parameter file.

  See the file `ex/params.txt` for an example parameter file. The first line should be the population scaled mutation rate $\theta = 4N_{\text{ref}}\mu$. The value of μ should ideally be based on some prior knowledge of the organism under consideration. The value of the reference effective population size N_{ref} can be chosen arbitrarily, but the inferred population size scaling factors can be multiplied by N_{ref} to obtain effective population sizes. It is best for the numerics of the program if these scaling factors are of order 1.

  The second line should be a 2×2 or 4×4 mutation matrix describing the probability allele a mutates to allele b, for all a, b. Each row should sum to 1. The example mutation matrix was estimated from human data in [1]. If a 4×4 matrix is specified, alleles A,C,G,T can be used.

If a 2×2 matrix is used, alleles A and C can be used.

Finally, the third line should be the population scaled recombination rate $\rho = 4N_{\text{ref}}r$. Note that the mutation rate, mutation matrix, and recombination rate are assumed to be the same for all loci. Blank lines and comments (`#`) are fine. In summary, the three lines should be:

1. $\theta = 4N_{\text{ref}}\mu$
2. mutation matrix, example format in `params.txt`
3. $\rho = 4N_{\text{ref}}r$

4 Recommended Parameters

If only the input data and input parameter file are specified, the program will use only the first $n = 2$ haplotypes, 8 time discretization intervals, and infer 3 population size parameters. Typically one will want to infer more parameters for a wider range of possible demographic histories. It is recommended that users also specify the following parameters (the first two are the most important).

- `-n / --nTotal`: the number of haplotypes to use (default = 2).

  The first n haplotypes will be used for inference. In the leave-one-out approach (see flag `-l`), each haplotype will be left out in turn, the hidden state being the time it joins the genealogy (of size $n - 1$), and the haplotype it joins onto (similar connotation as coalescence). More haplotypes will improve inference, but will also increase the runtime, which is quadratic in n for the leave-one-out approach. However, the computation can also be parallelized across haplotypes (see flag `-c`); ideally the number of cores used would equal n. Also note, if $n = 2$ is used, along with a symmetric mutation matrix, diCal will only leave out the first haplotype to reduce the runtime (since leaving out the second haplotype will produce the same results).

- `-p / --paramPattern`: the pattern of parameters spanning time intervals, given as a string (default = "3+2+3").

  Each population size parameter should ideally span at least two time intervals. The first (recent past) and last (ancient past) size parameters should in general span more intervals, since fewer coalescent events will be observed during these epochs. A reasonable parameter pattern would be

  ```
  -p "3+2+2+2+2+3"
  ```

  which means that the first size parameter spans 3 time intervals, the second spans 2 intervals, etc. In this example, time will be discretized into 14 intervals, and 6 population sizes will be inferred. Note that the adaptive parameter pattern method is not supported in version 1.3 since it does not work well for all scenarios, but it might be supported in later versions.

  If runaway behavior is observed (usually in the form of very large population sizes), fewer sizes should be used (so more intervals are spanned by each parameter). Runaway behavior is often the result of not enough events (i.e. coalescent events) occurring during a particular interval.

- `-t / --endTime`: start point of the last discretization interval, in coalescent units (default: not used).

  If the user knows the approximate data range of interest, they can specify and end time in coalescent units (i.e. units of $2N_{\text{ref}}$ generations). This will then scale the default discretization such that the last time point is the specified end time. This is also useful in the case when the default discretization (which is based on the data) looks unreasonable, which happens sometimes. See the Appendix of [5] for the default discretization procedure.

5 Java Virtual Machine Parameters

For most datasets, the default java heap memory size will not be sufficient and should be increased:

- `-Xmx`: use this flag to increase the memory for the java heap. For sequence length 1Mb, $n = 10$, and 16 discretization intervals, we used `-Xmx5G`.
- `-d64`: use this flag to specify a 64-bit machine.
- `-ea`: use this flag to "enable asserts" if desired or if any problems arise (will increase runtime).

6 Optional Parameters

- `-l / --leaveOneOut`: composite likelihood flag (default = 1/leave-one-out).

  By default the composite likelihood approach will be leave-one-out. If desired, users can specify `-l 0` for a traditional PAC (product of approximate conditionals) method with random permutations. This method has not been thoroughly tested. See [5] for more details.

- `-s / --seed`: the seed for the random number generator that initializes the permutations for PAC (default = default random number generator).

  This seed can be any long, used only for PAC.

- `-g / --numPerms`: the number of permutations for PAC (default = 5).

  The number of permutations of the haplotypes, used only for PAC.

- `-u / --linear`: linear runtime in the number of discretization points (default = 1/linear). One of the main updates of version 1.2 is that the runtime is linear in the number of time discretization intervals. This requires some new data structures, which can increase the memory requirements for small numbers of intervals. Overall though, this option will improve the runtime and is recommended over setting this flag to 0, which makes the runtime quadratic in the number of intervals. The optimization routine for the linear scenario is different in that it finds the maximum likelihood estimator for each size separately, so some differences in output are expected.

- `-c / --numCores`: number of cores to use (default = 1).

  If memory allows, it is optimal to set the number of cores equal to the number of haplotypes for the leave-one-out approach (then each one will be left out in parallel), or even higher for PAC, since each permutation can be done in parallel as well.

- `-d / --decodingInt`: what type of decoding to output (default = 0, no decoding).

  See the Posterior Decodings section below for more information.

- `-N / --numIter`: the number of EM iterations (default = 20).

  The default is reasonable for more most scenarios. This flag can be set to 0 in combination with one of the decoding options - then no inference will be performed. This can be useful in the case when the population size history is known and one wants to see how coalescence times are affected. Setting this flag too high can lead to overfitting.

- `-T / --timesFile`: the path to a file of times (and sizes, optional) (default = not used).

  This option is for specifying the user's own discretization intervals. See the example file `ex/times_sizes.txt`. If these are specified, the user can also specify a set of population sizes. This last option is useful mainly in the decoding scenario, without population size inference.

- `-e / --printExpectedSegs`: print the expected number of coalescent events (default = 0).

  This option can be set to 1 to print the expected number of coalescent events in each discretization interval. This computation is not linear in the number of time intervals, but it does not take long and is recommended so that a good discretization can be chosen. Intervals with very few coalescent events will contribute to runaway behavior.

7 Posterior Decodings

The hidden state in our model consists of an absorption haplotype and an absorption time (together describing a possible coalescence event). Computing a decoding through the path of hidden states can be very interesting, especially when the true demographic history is known and input to the method (see the `-T` flag above). In this case, the number of EM iterations can be set to 0 (`-N` flag), so no parameter inference will occur, but a decoding will be printed. Decodings can also be printed after parameter inference.

The following types of decoding can be specified (using the `-d` flag and the corresponding integer below). For all decodings, the times are taken as the midpoint of each interval (except the last interval, where the start time is chosen since the end time is ∞).

1. *posterior decoding*: Each line of output contains the locus, the posterior absorption time, and the posterior absorption haplotype index (with respect to the order the haplotypes appear in the fasta or VCF file).
2. *Viterbi decoding*: Similar format to the posterior decoding, with the best path absorption time and haplotype printed.
3. *posterior mean time*: For each time interval, the haplotype is marginalized out, then the posterior mean time (i.e. time a lineage joins the genealogy) is calculated.
4. *posterior decoding time*: Marginalize out the haplotypes, then choose the join-on time with the highest probability.
5. *posterior decoding time and probability*: Same as 4, but also prints out the highest probability.

8 Interpreting the Output

An example output file is provided in the `ex` folder. The main output is the line starting with "final sizes", after the EM procedure has finished. E-step likelihoods are also printed, along with individual haplotype likelihoods. The expected segments (in the case of the quadratic approach) are worth paying attention to - these show how many segments (separated by recombination breakpoints) coalesced during each time interval. Equal numbers of segments in each interval is desirable. Very imbalanced distributions are reg flags.

To translate the size scaling factors to real effective population sizes, multiply them all by the chosen N_{ref} corresponding to the input θ and ρ. To translate the discretization times to years, multiply them all by $2N_{\text{ref}}g$, where g is the number of years per generation. To compare the output with PSMC [2], see the Supplementary Material of [5].

9 Future Work

We are currently working on several areas of improvement to our method and implementation.

1. The runtime of our leave-one-out method is $O(Ldn^2)$, where L is the number of bases in each sequence, d is the number of time discretization intervals, and n is the number of sequences. The runtime could be significantly improved if we used a binning approach like PSMC [2]. However, creating a reasonable binning scheme for multiple sequences is less clear than for pairwise sequences. We are currently exploring this avenue, which could potentially enable whole-genome analysis.

2. Currently we have only tested the leave-one-out approach, but a more traditional PAC approach is implemented. For this method, it is unclear if the wedding cake genealogy is still a good approximation when the number of lineages is small, and it is also more difficult to choose a good discretization procedure. We are exploring ways to handle these issues and believe a well-implemented PAC approach should be able to improve the accuracy of our inference.

3. The choice of time discretization is critical to the performance of diCal. It is better to subdivide time more finely during periods with small population size than during periods with large population size when few coalescences occur. However, since the demography is what we are trying to infer, selecting an initial discretization is very difficult. An adaptive discretization scheme could potential improve inference across a wide range of demographic scenarios.

4. We have not fully investigated the impact (on both accuracy and speed) of different optimization routines. We currently use the Apache Commons Math implementation of Nelder-Mead [3], although the results in the paper used a different Nelder-Mead implementation that is no longer open source. For the linear method we use the Brent optimization routine from Apache Commons Math.

5. Our framework has been recently extended to incorporate structured populations with migration [6]. We are currently working on combining this extension with variable population size to create an integrated inference tool for general demographic models.

References

[1] Chan, A. H., Jenkins, P. A., and Song, Y. S. 2012. Genome-wide fine-scale recombination rate variation in *Drosophila melanogaster*. *PLoS Genet.*, **8**,(12) e1003090.

[2] Li, H. and Durbin, R. 2011. Inference of human population history from individual whole-genome sequences. *Nature*, **10**, 15.

[3] Nelder, J. A. and Mead, R. 1965. A simplex method for function minimization. *Computer Journal*, **7**,(4) 308–313.

[4] Paul, J. S., Steinrücken, M., and Song, Y. S. 2011. An accurate sequentially Markov conditional sampling distribution for the coalescent with recombination. *Genetics*, **187**, 1115–1128.

[5] Sheehan, S., Harris, K., and Song Y. S. 2013. Estimating variable effective population sizes from multiple genomes: A sequentially Markov conditional sampling distribution approach. *Genetics*.

[6] Steinrücken, M., Paul, J. S., and Song, Y. S. 2013. A sequentially Markov conditional sampling distribution for structured populations with migration and recombination. *Theor. Popul. Biol.*