Agemo: an open-source library for Laplace transformed coalescence time distributions.

Gertjan Bisschop *
Institute of Evolutionary Biology, University of Edinburgh, EH9 3FL Edinburgh, Scotland

Abstract

`agemo` is an open-source tool with a python API that allows users to generate the Laplace transform of the coalescence time distribution of a sample under any structured coalescent model. In addition, `agemo` provides ways to efficiently query that distribution, by using the fact that its generating function can be represented most simply as a directed graph with all possible ancestral states of the sample as nodes. Past implementations have not made full use of this, relying on computer algebra systems instead of graph traversal to process these recursive expressions. `agemo` can be used to compute the probabilities of the joint site frequency spectrum for blocks of a given size, under the specified model. Calculating these probabilities requires repeated differentiation of the generating function which suffers from an explosion in the number of terms when implemented naively. Using a closed-form expression for the coefficients of a series expansion of the equations associated with each edge, we can efficiently propagate these coefficients through the graph avoiding redundant operations.

1 Introduction

Laplace transforms have been used extensively in coalescent theory since the introduction of the structured coalescent (Takahata, 1988; Notohara, 1990; Barton and Wilson, 1995). Initially it was a

*g.bisschop@sms.ed.ac.uk

formal tool to analyse continuous-time Markov chains (Takahata, 1988; Griffiths, 1991; Wilkinson-Herbots, 1998). But as noted by Wilkinson-Herbots (1998) the Laplace transforms of the distributions of coalescence times are also interesting in their own right. The Laplace transform of a random variable has a clear probabilistic interpretation. One can think of that random variable as describing the length of an interval. If a Poisson marking process with intensity ω marks the interval, then the Laplace transform $f^*(\omega)$ is the probability of not observing any marks in the considered interval (Råde, 1972). Let the random variable be the stochastic process modelling the (sum of the) exponentially distributed waiting times describing the ancestry of a random sample, then the marking process can describe the arrival of mutations. Formulated as such, the connection to the coalescent (Kingman, 1982; Hudson, 1983; Tajima, 1983) becomes obvious, making the Laplace transform a natural choice (Weissman and Hallatschek, 2017).

Moreover, used in mathematical fields like queueing theory, the Laplace transform often provides us with an alternative description of the behaviour of a system, often simplifying its analysis in the process. In essence, this is what the generating function (GF) method as described in Lohse et al. (2011) does. Using the probabilistic interpretation of the Laplace and the fact that the transform turns convolution into multiplication (see 2.1), one can come up with a much simpler, recursive and automated, description of the distribution of the (sum of) interevent times given a set of samples and a (structured coalescent) model. Phrased differently, being given a flow diagram or coalescent state space graph describing all possible transitions during the coalescent process (see fig. 2.1.2), one can readily write down all associated expressions in the Laplace domain. This is not the case for the time domain.

This framework has been used to develop likelihood-based inference methods both for IM-type demographic histories (Lohse et al., 2011; Bunnefeld et al., 2015; Lohse et al., 2016) as well as estimating sweep parameters (Bisschop et al., 2021). Given that the GF relies on generating the coalescent state space graph, and that this state space graph will grow superexponentially with sample size, there are limitations on the sample size and number of events for which the GF can be generated and more importantly evaluated within reasonable (memory) space and time. These two bottlenecks, the ability to make mathematically tractable descriptions of a problem as well the ability to efficiently implement the solution hold for all inference problems. The first issue is generally solved by making simplifying assumptions about recombination, either ignoring the information contained within closely linked variants completely (Gutenkunst et al., 2009; Excoffier et al., 2013), or by only ignoring recombination within very short blocks (Yang, 2002; Hey, 2004). Additionally, one can limit the description to a small number of lineages or only consider very basic underlying (demographic) models.

The second issue is often very closely connected to the type of mathematical description used.

In general, one will benefit from the potential to implement a particular solution in terms of data structures for which efficient algorithms exist for each of the required basic operations. For example, preserving matrix structure up to the point of evaluation, as used in phase-type theory (Hobolth et al., 2019), will tend to have a positive impact on performance due to the existence of many efficient algorithms for linear algebra operations.

So far, all inference approaches using the GF, rely on deriving the probability of observing each of the different block-wise site frequency spectrum (bSFS) vectors. These combinations of mutation configurations along the distinguished branch types within blocks of a certain length summarise the joint distribution of linked polymorphisms (Bunnefeld et al., 2015). However, depending on both the number of observed mutations and the number of distinguished branch types, they require repeated differentiation of the entire GF. As a consequence, any naive implementation of the GF with higher-order derivatives using a computer algebra system (CAS) will quickly run into computational bottlenecks, essentially limiting the extendability of the framework.

Previous efforts to make the framework more efficient have targeted the first bottleneck category by limiting the number of distinguishable branchtypes, limiting the number of occurrences of particular events, and using symmetries within the recursive description (Lohse et al., 2016). No attempts have been made to tackle the second bottleneck and describe the problem in a way that allows for more efficient computation.

Here we present the first open-source implementation of the GF framework and lay out the key algorithms that allow us to efficiently calculate the bSFS by using the correspondence between the recursive formulation of the Laplace transform and the coalescent state space graph. The focus on the bSFS is only superficial however. Any future extension of the library will make use of the concepts outlined here.

First, we will outline how the GF can be represented more succinctly. Secondly, we will lay out the graph traversal algorithm allowing us to efficiently evaluate the GF. We will then apply this to the bSFS by using the algorithm to propagate the coefficients of a truncated Taylor series through the graph. Finally, we show how to extend bSFS type calculations to the fully phased and rooted case by making use of mutation type symmetries to limit the amount of computation needed.

2 Methods

2.1 Recursive description of the Laplace transform

2.1.1 single population

Given a sample of n uniquely labelled lineages $\Omega = \{a, ..., n\}$ coming from a single population, we can represent all possible coalescent histories of that sample as a series of trees. In each of these ranked topologies nodes represent the lineage ancestral to the subtended nodes. Alternatively, the same information can be captured by a single rooted directed graph describing the state space of all (coalescence) events affecting the history of the sampled lineages. In this graph, each node is uniquely labelled by one of the partitions of Ω. As lineages coalesce, we move through the graph from the source node, representing the set of all sampled lineages, to the root or most recent common ancestor (mrca). In the case of the neutral coalescent and a single population each path through the graph is equivalent to a single ranked topology. However, in general the state space graph will contain information on all events that affect the lineage configuration, not only those that join to lineages.

For k uncoalesced lineages, the rate of coalescence is $\binom{k}{2}$ (in units of $2N_e$ generations). Because each step is conditionally independent of the previous and as touched upon in the introduction, the Laplace transform of the sum of inter-event times or the time to the most recent common ancestor (t_{mrca}) is a simple product of all Laplace transforms of the random variable describing the waiting time to go from k to $k-1$ lineages. As such, we can associate each edge of the directed graph with a single equation describing this transition. The Laplace transform of the t_{mrca} for a single (ranked) topology can then be retrieved by multiplying the equations associated with a single path of the graph. Likewise, the entire GF is the sum of all expressions retrieved by a depth-first traversal of the graph (see 2.1.2).

Knowing that the equation associated with a single edge in the coalescent state space graph as outlined above, gives the probability of observing the coalescence event prior to any other event happening at rate ω, for any two out of k lineages, the Laplace transform is $f^*(\omega) = \frac{1}{\binom{k}{2}+\omega}$. Note that in general we can label each branch connecting two lineages in a coalescent tree by the samples that will be affected by a mutation arising during the time represented by that branch. And in general, we will associated each of these branch types with a unique dummy variable. Replacing ω by $\sum_{i=1}^{b} \omega_i$, we can retrieve the coalescence time distribution for each of the b distinguished branch types (Lohse et al., 2011).

2.1.2 general form

So far, we have dealt with a single population, tracking only coalescence events. In a structured population, we require at least one type of (migration) event to ensure lineages eventually all end up in the same population to allow for coalescence. These events can be incorporated in the state space graph by adding a node for each of these events and labelling that node by their impact on the tracked set of lineages. Looking at any node in the graph, the set of all possible events is determined by all outgoing edges and the set of all observable branchtypes during the time represented by each edge will be given by the label of the node from which the edges are leaving. The LT associated with a single edge still describes the probability of observing a particular event prior to all other events. So as long as the events that we have added in are associated with exponentially distributed waiting times and given that $min(X, Y) \sim exp(\omega_1 + \omega_2)$ when both $X \sim exp(\omega_1)$ and $Y \sim exp(\omega_2)$, we can still readily associate each edge with the correct equation.

$$f[\boldsymbol{\omega}] = \frac{c}{\sum c_i \kappa_i + \sum l_j \lambda_j + \sum o_k \omega_k} \tag{1}$$

Here, Roman letters represent integers counting the number of ways a certain (coalescence or other) event can take place (c_i, l_j) given the current state space, or the number of branches of a particular type (o_i). The Greek letters represent the rate of the associated competing processes. Note that in the case of coalescence with multiple populations, rates are relative and given by $\kappa_i = N_{e_i}/N_{e_{ref}}$.

Each equation can be encoded as a matrix with two rows containing the integer coefficients (c, c_i, l_j, o_k). The first row represents the numerator and will only contain a single non-zero value. The second row then represents the denominator. Storing the equations this way ensures we can efficiently substitute in parameter values by taking the dot product with a vector representing a point in parameter space ($\kappa_i, \lambda_j, \omega_k$) once the Laplace transform needs to be evaluated. Also, storing equation coefficients in array form allows us to efficiently perform operations on the equations (see 2.1.3). A minimal representation of the GF therefore consists of an array containing all unique equations, and an array of arrays with equation indices describing all paths through the graph.

2.1.3 discrete events

For the general description of the GF we have assumed that all competing processes have exponentially distributed waiting times. It is possible however to incorporate events that only happen once. Initially treating these discrete events as a competing exponential process with rate δ, we can recover the GF parametrised by the discrete time T of the event, by taking the inverse transform of the GF divided by its associated dummy variable (δ). This has been used to incorporate population

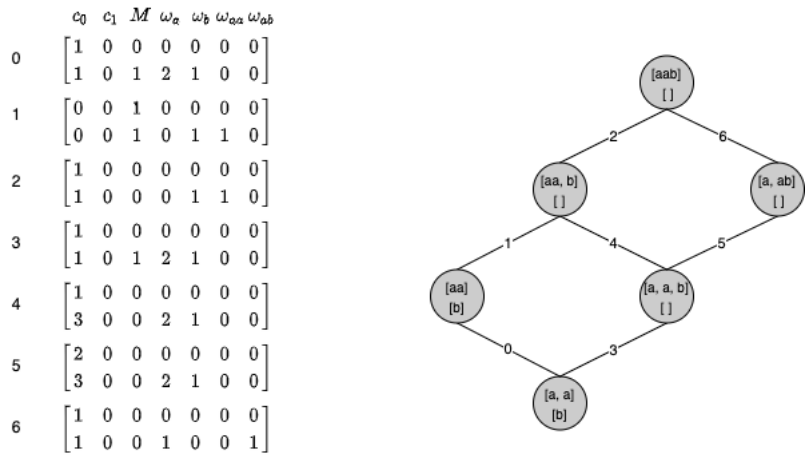

Figure 1: **From coalescent state space to equation array:** coalescent state space graph for two populations A and B with 2 and 1 unphased sample(s) respectively. The demographic model assumes a single mass migration event from population B to A back in time.

splits and bottlenecks (Lohse et al., 2011; Bunnefeld et al., 2015) as well as hard sweeps (Bisschop et al., 2021).

Previous implementations have relied on a CAS to get an analytic solution for the inverse transform upon each evaluation of the GF. However, it is clearly overkill to determine the inverse Laplace symbolically repeatedly given that the GF will always be a sum of products of factors of the form $c/(d+\delta)$ with c and d constants, independent of the complexity of the state space graph. That is, at least as long as we limit ourselves to a single discrete event. Having stored all equation coefficients as an array, we can formulate a closed-form solution to the inverse Laplace that allows for efficient substitution of all parameter values (see S3).

If we want to maintain the outlined correspondence between the state space graph and the GF as a sum of the products of the expressions associated with each path then we have to modify the state space graph. This can be done by creating a new node for each section of all paths leading up to a node associated with a discrete event. Each of these new nodes is connected to the root of the graph by a single edge associated with its respective inverse (see fig. 2.1.3).

In essence this means that the part of the path downstream of the discrete event remains unaltered, each edge still associated with a simple equation of the form 1. While each path upstream of that node is now collapsed into a single node. Once this operation has been performed, each path of the graph represents a product of factors again. Under the hood however, we perform a

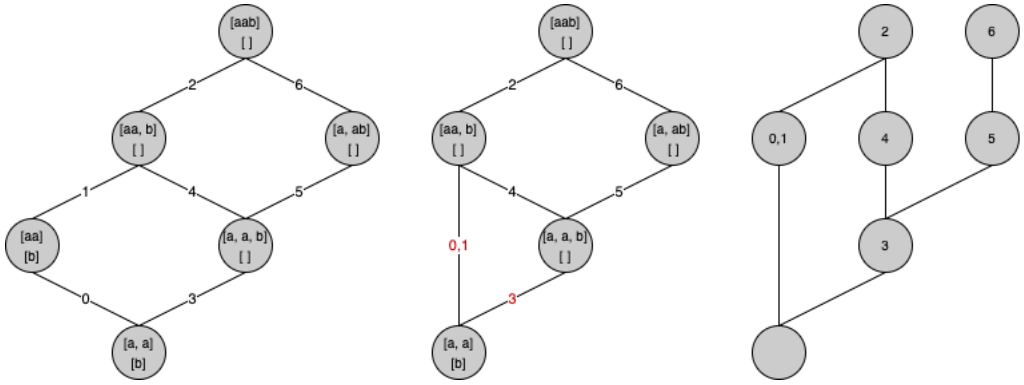

Figure 2: **From coalescent state space to equation graph:** setup identical as fig. 2.1.2. A shows the initial coalescent state space graph, while B shows the collapsed form grouping all parts of each path that require inverting with respect to the dummy variable associated with the discrete event. C takes this logic one step further turning the (sets of) equations previously associated with the edges into nodes, simplifying algorithms that do not require state space information.

graph inversion while collapsing turning edges into nodes and vice versa (see fig. 2.1.3 C). This simplifies the graph traversal algorithm (see 2.2).

2.1.4 adding new events

Currently, besides coalescence, two types of events have been implemented: uni-directional migration at a constant rate and population splits (forwards in time). Other events can readily be incorporated by adding in a method that provides a description of what the event does to the configuration of lineages (or node label) while generating the state space graph. There are two limitations however. First of all, the defined events should not lead to cycles in the directed graph. For example, in line with Lohse et al. (2016) migration is limited to being uni-directional such that lineages cannot cycle indefinitely between populations without ever reaching the fully coalesced state. Finally, note that so far our approach is only compatible with a single discrete event. Adding in more discrete events would require the use of a CAS to determine the inverse transform.

2.2 graph traversal

Given an equation graph (as described in 2.1.3) and a dependency sequence describing the evaluation order of the graph, a general algorithm to propagate any evaluation of the equations associated with each node is given by alg. 1. It relies on the fact that implicitly the edges of the graph rep-

resent multiplication. Once both multiplication and addition have been defined for the propagated values we can rely on the commutative property to efficiently traverse the graph towards the root. Especially in the case where addition is a less costly operation than multiplication (as is the case for polynomials, see 2.3), it will pay off to add the values associated with the children of a node prior to multiplication. We can now apply this algorithm to efficiently calculate the bSFS.

Algorithm 1: Propagate values through graph.

```
function PROPAGATE;
  Input: adjacency list of graph, node_values, evaluation_order
foreach parent in evaluation_order do
    children = graph[parent];
    temp = 0;
    if children then
        foreach child in children do
            temp+=node_values[child];
        end
        if parent not root then
            node_values[parent] = PRODUCT(temp, node_values[parent]);
        else
            return temp;
        end
    end
end
```

2.3 blockwise site frequency spectrum

2.3.1 graph traversal

The block-wise site frequency spectrum or bSFS is the vector of site frequency spectrum counts in short blocks of a fixed length (Bunnefeld et al., 2015). It is the probability of seeing k_i mutations on each of the i distinguished branchtypes. For an unphased sample of size 3 the bSFS would be a vector of the form $\{k_1, k_2\}$ with k_1 and k_2 representing the number of singletons and doubletons respectively. Each count is an element within the interval $[0, 1, 2, ..., k_i^{max} + 1]$ where $k_i^{max} + 1$ is used to bin all mutation configurations with more than k_i^{max} mutations.

Deriving the probability for each of the $(\prod_{i=1}^{c} k_i^{max} + 2)$ possible bSFS configurations, with c the length of the bSFS vector, has thus far been the way in which the recursively generated Laplace

transform of coalescence time distributions has been used. The probability of observing mutation configuration $\boldsymbol{k}$ under a specified model can be obtained by taking the ${k_i}^{\text{th}}$ derivative with respect to ω_i (see eq. (1) in Lohse et al. (2011) for details).

Any naive approach, based on calculating all higher order derivatives using a CAS, will suffer from an explosion in the number of terms due to the Leibniz or product rule when differentiating. Generally a CAS will fail to take into account the fact that the same partial derivatives are computed multiple times. This problem has been well-studied for the purpose of automatic differentiation algorithms (Neidinger, 1992, 1995; Griewank et al., 2000; Bettencourt et al., 2019). One possible approach to overcome this issue consists of implementing recurrence relations on how to build and combine the coefficients of truncated Taylor series. For such a series, the coefficient of $x^{\mathbf{k}}$ is indeed proportionate to the desired partial derivative. By defining the series coefficients for all elementary functions and defining the recurrence relations for arithmetic operators as well as function compositions one can build higher-order derivatives from the ground up while avoiding having to recalculate the same derivative multiple times(Neidinger, 2013).

Translating this to the graph traversal algorithm outlined above requires us to first obtain all coefficients for a truncated Taylor series of the equation associated with each node in the equation graph. We can then use the algorithm defining the product of two truncated Taylor series (see alg. (8) in Neidinger (2013) and alg. 2) to propagate the coefficients of the series associated with each node. Note that adding two truncated Taylor series simply amounts to the pairwise addition of all corresponding coefficients. To get at the higher order derivatives needed for the first step, we could use the recurrence relations as defined in Neidinger (2013). However, all equations associated with each of the nodes of the equation graph are well characterised. They fall into one of two categories depending on wether an inversion step was needed. For each of the two categories we can define a closed-form implementation of the derivatives with respect to the distinguished branchtypes (see eqs. S1 and S2).

Remark that alg. 2 contains an explicit `ADD` function. Care needs to be taken when summing (a subset of) the coefficients of a Taylor series. These will be both positive and negative, and as such catastrophic cancellation might occur leading to accuracy loss. The compensated summation algorithm of Ogita-Rump-Oishi (Ogita et al., 2005) is implemented as an efficient way to bound the potential loss in accuracy. Another way of handling this would be to temporarily increase numeric precision at the crucial steps. Lastly, an advantage of using Taylor series coefficients rather than the derivatives is that the coefficients will always be smaller by a factor $(\sum \boldsymbol{k})!$ leading to less cumulative roundoff error.

Algorithm 2: Product of two truncated Taylor series (Neidinger, 2013).

function SERIES_PRODUCT;
Input : Two same-size arrays of floats a and b
Output: array c of same size as a and b
foreach *multi-index* $\boldsymbol{k}$ **do**
    $sum = 0$;
    foreach *multi-index* $\boldsymbol{j} \leq$ *multi-index* $\boldsymbol{k}$ **do**
        sum = ADD(sum, $a[\boldsymbol{j}] * b[\boldsymbol{k} - \boldsymbol{j}]$);
    end
    $c[\boldsymbol{k}]$ = sum;
end
return $c[\ ]$;

2.3.2 Mutation type equivalences

When labelling each branch by the labels of the lineages it subtends, the number of branch types and therefor the number of mutation types (a single bSFS vector), will quickly go up. Calculating the probability for observing each one of these will become unfeasible very fast. To alleviate this we can make two types of simplifications. Following Lohse et al. (2016), we can remove phase and/or root information by a relabelling of the branches. This reduces the number of branch types and mutation types one distinguishes. Phase is removed by labelling all branches according to their number of descendants in each subpopulation. Root information on the other hand can be left out by consistently labelling each branch by the set of lineages it subtends or by the complement of that set. Note that applying these simplifications might mean one cannot fully use all information present in a dataset.

Additionally, we can make use of the fact that some mutation types are inherently equiprobable. For example, given a sample from a single population labelled $\{\{a, b, c\}\}$ mutation type $\{a, ab\}$ will have the same probability as 5 other mutation types. To determine this equivalence it suffices to see that each mutation type can be represented in two ways. Either each mutation type is labelled as an integer vector indicating presence or absence of one or more mutations along branch types $\{a, b, c, ab, ac, bc\}$. In our example this equates to $\{1, 0, 0, 1, 0, 0\}$. We can apply a similar representation rule to all branch types as well resulting in in a boolean vector containing 1's for the samples the branch subtends. Because each mutation type consist of no, one or multiple branch type(s), a single matrix combining the vectors for the branch types present can represent a mutation type as well. For this representation, all permutations of columns representing samples from the

same population represent equiprobable configurations.

Besides all this, a large fraction of mutation types is simply impossible to observe. Realising that the bSFS is a sparse array can reduce computation time even further by limiting computation to those mutation types that can actually be observed.

3 Results and Discussion

Currently, `agemo` is implemented in `python 3`, relying on `numba` jit-compilation to speed up the critical parts of the code. Compiling the code using `numba` has a few consequences. Firstly, compilation happens each time the code is run and will require a few seconds. This is generally not an issue given that usually for a single model many points in parameter space will be evaluated, for example when calculating the bSFS. Secondly, some numerical operations are implemented differently in `numba` than in `numpy`. This is particularly true for summation and has required us to implement a compensated sum algorithm. A potentially faster solution would be to temporarily increase machine precision for the evaluation of particular sums. This is not possible using `numba` however and would require us to transfer part of the code to `C`.

Because we have opted to no longer depend on a CAS to calculate derivatives, no longer placing the evaluation order of any expression out of user's reach allowing us to alleviate potential floating point precision issues. This was not possible previously forcing any implementation to use rationals rather than floats increasing computation time. However, there might be a point whenever one might want to analyse more complex models (e.g. containing more than one discrete event) that the need for relying on symbolic algebra or automatic differentiation becomes necessary. This does not change the general idea outlined in this paper. It will still pay off to evaluate the building block equations associated with the coalescent state space graph first, and then rely on graph traversal to get to the correct results. This could be achieved by putting in place a function to propagate a user-provided array (with first dimension equal to the number of equation nodes) through the graph using user-defined addition and multiplication rules. This way, we would not have the additional burden of having a CAS as a direct dependency.

There are more efficient algorithms to multiply (truncated) multivariate polynomials. The fast Fourier transform (FFT) would allow us to perform the operation at $\mathcal{O}(nlog(n))$ with n the order of the polynomial instead of $\mathcal{O}(n^2)$ for the current multiplication. However, gains would only be noticeable for a large number of distinguished branchtypes, or a really large k_{max} values.

These algorithms will not suddenly make it possible to extent the usage of the GF to any arbitrary model and sample size. The fact remains that the number of sample configurations grows

superexponentially with sample size (Lohse et al., 2016).

Any incorporate more demographic events (bottleneck,) and incorporate hard sweep approximations (Bisschop et al., 2021). Although most people still process vcfs when it comes to mutation data. There is the more succinct treesequence format containing both topology information as well as mutations. It would therefore make a lot of sense to add on the ability to calculate the likelihood of a mutation configuration under a particular coalescent model as given by a marginal tree from a treesequence. This boils down to restricting the GF for that particular model to the observed topology, count the number of mutations of each distinguished branchtype and compute its probability.

In summary, the work presented here constitutes a CAS-independent, open-source implementation of the GF. A general outline has been given on how to rely on the correspondence between the coalescent state space graph and the GF to query the distribution of Laplace transformed coalescence times efficiently. In particular, algorithms have been laid out to efficiently calculate the bSFS making this package ideal as a backend for likelihood calculations.

acknowledgements

This work was supported by an ERC starting grant (ModelGenomLand, 757648)

References

Barton, N. H. and Wilson, I. (1995). Genealogies and geography. *Philosophical Transactions of the Royal Society B*, 349(1327):49–59.

Bettencourt, J., Johnson, M. J., and Duvenaud, B. D. (2019). Taylor-Mode Automatic Differentiation for Higher-Order Derivatives in JAX. 10.

Bisschop, G., Lohse, K., and Setter, D. (2021). Sweeps in time: Leveraging the joint distribution of branch lengths. *Genetics*, 219(2).

Bunnefeld, L., Frantz, L. A., and Lohse, K. (2015). Inferring bottlenecks from genome-wide samples of short sequence blocks. *Genetics*, 201(3):1157–1169.

Excoffier, L., Dupanloup, I., Huerta-Sánchez, E., Sousa, V. C., and Foll, M. (2013). Robust Demographic Inference from Genomic and SNP Data. *PLoS Genetics*, 9(10).

Griewank, A., Utke, J., and Walther, A. (2000). Evaluating Higher Derivative Tensors by Forward Propagation of Univariate Taylor Series Source : Mathematics of Computation , Jul ., 2000 , Vol . 69 , No . 231 (Jul ., 2000), pp . 1117- Publi. *Mathematics of Computation*, 69(231):1117–1130.

Griffiths, R. C. (1991). The Two-Locus Ancestral Graph. *Selected proceedings of the Sheffield symposium on applied probability*, 18(1991):100–117.

Gutenkunst, R. N., Hernandez, R. D., Williamson, S. H., and Bustamante, C. D. (2009). Inferring the Joint Demographic History of Multiple Populations from Multidimensional SNP Frequency Data. *PLoS Genetics*, 5(10):e1000695.

Hey, J. (2004). Multilocus Methods for Estimating Population Sizes, Migration Rates and Divergence Time, With Applications to the Divergence of Drosophila pseudoobscura and D. persimilis. *Genetics*, 167(2):747–760.

Hobolth, A., Siri-Jégousse, A., and Bladt, M. (2019). Phase-type distributions in population genetics. *Theoretical Population Biology*, 127:16–32.

Hudson, R. R. (1983). Testing the constant-rate neutral allele model with protein sequence data. *Evolution*, 37(1):203–217.

Kingman, J. F. (1982). The coalescent. *Stochastic Processes and their Applications*, 13(3):235–248.

Lohse, K., Chmelik, M., Martin, S. H., and Barton, N. H. (2016). Efficient strategies for calculating blockwise likelihoods under the coalescent. *Genetics*, 202(2):775–786.

Lohse, K., Harrison, R. J., and Barton, N. H. (2011). A general method for calculating likelihoods under the coalescent process. *Genetics*, 189(3):977–987.

Neidinger, R. D. (1992). An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order. *ACM Transactions on Mathematical Software (TOMS)*, 18(2):159–173.

Neidinger, R. D. (1995). Computing multivariable Taylor series to arbitrary order. In *Proceedings of the international conference on Applied programming languages - APL '95*, pages 134–144, New York, New York, USA. ACM Press.

Neidinger, R. D. (2013). Efficient recurrence relations for univariate and multivariate Taylor series coefficients. *Conference Publications*, pages 587–596.

Notohara, M. (1990). The coalescent and the genealogical process in geographically structured population. *Journal of mathematical biology*, 29:59–75.

Ogita, T., Rump, S. M., and Oishi, S. (2005). Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988.

Råde, L. (1972). On the use of generating functions and laplace transforms in applied probability theory. *International Journal of Mathematical Education in Science and Technology*, 3(1):25–33.

Tajima, F. (1983). Evolutionary relationship of DNA sequences in finite populations. *Genetics*, 105(2):437–460.

Takahata, N. (1988). The coalescent in two partially isolated diffusion populations. *Genetical Research*, 52(3):213–222.

Weissman, D. B. and Hallatschek, O. (2017). Minimal-assumption inference from population-genomic data. *eLife*, 6:1–21.

Wilkinson-Herbots, H. M. (1998). Genealogy and subpopulation differentiation under various models of population structure. *Journal of Mathematical Biology*, 37(6):535–585.

Yang, Z. (2002). Likelihood and Bayes estimation of ancestral population sizes in hominoids using data from multiple loci. *Genetics*, 162(4):1811–1823.

Supplementary Information

closed-form derivatives

Given a first degree multivariate polynomial of the form $f(\mathbf{x}) = \sum c_i x_i + b$, then the following higher order derivatives have a closed form. With and $s = \sum k_i$

$$\frac{\partial f(\mathbf{x})^{-1}}{\partial^{k_i} x_i} = (-1)^s s! \frac{c_i^{k_i}}{f(\mathbf{x})^2} \tag{S1}$$

$$\frac{\partial e^{cf(\mathbf{x})}}{\partial^{k_i} x_i} = c^s c_i^{k_i} e^{cf(\mathbf{x})} \tag{S2}$$

These two base case derivatives together with the defined Taylor series product allow us to calculate all required derivatives for the entire GF assuming at most only a single inverse Laplace with respect to a discrete event is required.

closed-form inverse Laplace transform

Given $f(c) = \frac{1}{c+\delta}$, the inverse Laplace transforms (wrt δ) can be defined as follows. Here T represents the time to the discrete event.

$$g(T) = \mathscr{L}^{-1}(\frac{1}{\prod_{i=0} f_i(\mathbf{x})}) = \sum_i \frac{(-1)^{i+1} e^{-cT}}{\prod_{i \neq j} (f_{min(i,j)}(\mathbf{x}) - f_{max(i,j)}(\mathbf{x}))} \tag{S3}$$

Note that all derivatives of S3 can be determined using the equations from the section above.