frontiers in Bioinformatics

TYPE Review
PUBLISHED 19 August 2026
DOI 10.3389/fbinf.2026.1878981

Check for updates

OPEN ACCESS

EDITED BY
Jack M. Craig,
Temple University, United States

REVIEWED BY
Wan-Ping Lee,
University of Pennsylvania, United States
Alejandro Mejia Garcia,
McGill University, Canada

*CORRESPONDENCE
Patrick Fournier,
pf@patrickfournier.ca

RECEIVED 12 May 2026
REVISED 21 June 2026
ACCEPTED 30 June 2026
PUBLISHED 19 August 2026

CITATION
Fournier P and Larribe F (2026) Human ancestries simulation and inference: a review of ancestral recombination graph-based approaches.
Front. Bioinform. 6:1878981.
doi: 10.3389/fbinf.2026.1878981

COPYRIGHT
© 2026 Fournier and Larribe. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

Human ancestries simulation and inference: a review of ancestral recombination graph-based approaches

Patrick Fournier* and Fabrice Larribe

Département de Mathématiques, Université du Québec à Montréal, Montréal, QC, Canada

The importance of the ancestral recombination graph (ARG) in population genetics is undeniable. An important theoretical tool, the main obstacle to its widespread usage is the computational cost required to match the ever-increasing scale of the data being analyzed. Many of these difficulties have been overcome in the past 2 decades, which have consequently seen the development of increasingly sophisticated ARG simulation and inference software. Nonetheless, challenges remain, especially in the area of ancestry inference. This study is a comprehensive review of ARG simulation and inference programs that have emerged in the past 3 decades to meet the need for scalable and flexible ancestry simulation and inference solutions. It specifically focuses on their performance, usability, and the biological realism of the underlying algorithm and primarily aims to provide a technical overview of the field for researchers seeking to design and implement their own coalescent-with-recombination algorithm. As a complement to this article, we have compiled the links to software, source code, and documentation and made them available at https://patrickfournier.ca/publications/arg-software-review/graph.

KEYWORDS
ancestral recombination graph, ancestral recombination graph inference, ancestral recombination graph simulation, coalescent theory, software review

1 Introduction

The ancestral recombination graph (ARG) has been described by Hubisz and Siepel (2020) as the "holy grail" of statistical population genetics. It is, by all accounts, of great value to the analyst, and while less elusive than the proverbial Arthurian relic, many challenges have hindered its widespread adoption. Among these, the heavy computational burden associated with its simulation and inference from genetic data is probably the most noteworthy. It is true that many of these difficulties have been overcome in the past 2 decades, which have consequently seen the development of increasingly sophisticated and performant solutions. Nonetheless, challenges remain, especially in the area of ancestry inference.

This study reviews ARG simulation and inference software from the early 2000s to the present day. As such, it is not the first review of its kind. Previous work include:

- Carvajal-Rodriguez (2008): a short comparison of the features of various coalescent-with-recombination (CWR) simulators and some forward-in-time approaches not based on the coalescent.
- Liu et al. (2008): similar to Carvajal-Rodriguez (2008) but compares fewer features.

- Arenas (2012): a comparison of CWR, forward-time, and phylogenetic simulators, with a focus on evolutionary scenarios and mutation models.
- Yuan et al. (2012): a comparison of features, computational complexity, and execution time of CWR and forward-time simulators, along with the mathematical basis of the underlying models and a brief presentation of resampling approaches.
- Hoban et al. (2012): an extensive comparison of the features of forward-time and CWR simulators, along with practical application considerations.
- Yang et al. (2014): a comparison of five simulators revolving around the simulation of recombination hotspots.
- Brandt et al. (2024): a review of three ARG inference software applications, along with a table summarizing most of the available inference programs as of 2023.
- Nielsen et al. (2024): a short and to-the-point introduction to coalescent theory with a comparison of major inference software featuring a small-scale simulation study.

In comparison, our evaluation is differentiated by three aspects. First, it is very extensive, if not exhaustive. We review 33 programs in detail and mention eight others. We emphasize that our review is limited to programs capable of outputting ARGs. In particular, pieces of software designed to produce exclusively lower-dimensional objects such as genotype matrices and summary statistics are excluded. From this perspective, our work encompasses a broader range of tools than any of the above-listed articles. Second, it includes both simulation and inference software (see Section 1.1.4). Third, and most importantly, it primarily aims to provide a technical overview of the field for researchers seeking to write their own simulation software. It focuses on the performance, usability, and biological realism of the underlying algorithm. Specifically, we provide overviews of algorithms, highlight noteworthy innovations, and summarize their relation to other ancestral recombination graph simulation and inference procedures to provide a sense of the current state of the field and its evolution over the last 25 years. When appropriate, we describe the practical aspects of software packages, such as the application programming interface (API), input and output formats, and additional functionalities. As such, it is the only review of this kind and fundamentally different from those cited above, which are written with the user rather than the programmer in mind.

1.1 Vocabulary

Terminology in coalescent theory is extensive and sometimes confusing. To minimize misunderstanding, this article adheres to a strict vocabulary, which is explained in this section.

1.1.1 ARG vs. coalescent-with-recombination

As noted by Wu (2008) and Wong et al. (2024), the term "ARG" can be confusing, as it is not used consistently in the literature. To ensure clarity, we will reserve its usage to refer to a graph-theoretical object containing information about the evolutionary history of a sample; this corresponds to the "data structure" usage in Wong et al. (2024). For the other meaning, that of a stochastic process, we use the term *coalescent-with-recombination* or the acronym CWR.

1.1.2 Location vs. position

Recombination events are inherently two-dimensional. They are characterized by their location on the ARG and on the sequences. We refer to these quantities as the *location* and the *position* of the event, respectively. The meaning is preserved when we write that an event is *located* or *positioned* somewhere.

1.1.3 Recombination, chromosomal crossover, and gene conversion

For the purpose of this article, "recombination" denotes a class of genetic processes involving the exchange of material between chromosomes. Chromosomal crossover and gene conversion are both members of this class, with the former being the main focus of ARG-based approaches. This is not surprising, considering its biological significance and its relevance for genetic analysis. In fact, the meaning of the term "recombination" is often restricted to "crossover" in much of the literature on coalescent theory.

Chromosomal crossover is the process by which two chromosomes swap portions of their material. Gene conversion also involves two chromosomes, but is more unidirectional in comparison; it results in a single chromosome being modified. A segment of the donor simply replaces the homologous segment in the recipient chromosome.

1.1.4 Simulation vs. inference

ARG *simulation* aims to generate an ARG from a set of genetic parameters such as the recombination rate and effective population size. The set of haplotypes associated with vertices that have no children constitutes a sample of genome sequences and is often of primary interest. ARG *inference* is the inverse problem of producing ancestries that are "consistent" with a given sample of haplotypes. Similarities and differences between these two procedures are discussed in Section 2.3.

2 Typology

This section details the broad categories guiding our review.

2.1 Model vs. heuristic

Model-based algorithms generate events according to predetermined probability distributions. These include, without being limited to, coalescences, chromosomal crossover, mutations, and migration. They are mainly used to generate an ancestry using parameters estimated from genome data. The resulting graphs are in turn useful for estimating quantities, including the time to the most recent common ancestor (TMRCA) of positions of interest.

On the other hand, heuristic-based algorithms give little to no importance to probability distributions. A major heuristic is the parsimony of mutation and recombination events. Algorithms

designed accordingly follow the general idea that ancestries with fewer events are more likely to resemble real-world genealogies.

Although heuristic algorithms produce events according to broad guiding principles rather than probability distributions, they are not inherently deterministic. Finding an evolutionary network minimizing the number of recombinations, for instance, is NP-hard. Therefore, many ARG inference programs rely on greedy algorithms that can reach states where the next element must be chosen from a set in which all elements equally satisfy the heuristic. In such situations, it is not unusual for the software to choose its next state randomly. The distribution of that state, however, is expected to be uninformative, as it exists merely as a convenient way of dealing with nearsightedness rather than fitting data.

The level of performance achieved by parsimonious algorithms often constitutes their main appeal. Generating ARGs according to a probability distribution is, in itself, a computationally demanding task that they largely avoid. Even when necessary, random number generation is generally limited to continuous or discrete uniform distributions, which are the easiest to sample from. Perhaps more importantly, parsimony means that the number of generated events itself is likely reduced. Combined with careful implementation, this makes them difficult to beat on any time- or space-measuring benchmark.

Notwithstanding the intuitive appeal of the various heuristics implemented by this class of algorithms, model-based ones are the most natural from a statistical standpoint. At the end of the day, genetic data are data, and a major use for ARG simulation and inference in an analysis pipeline is to produce genealogies from a user-specified distribution.

In other contexts, the goal is to infer the ancestry of a sample of haplotypes. The choice between model-based and heuristic-based solutions is often dictated by methodological considerations and the complexity of the task. One's preference for the statistical rigor of a mathematical model of population evolution might be tempered by the reality that producing a genealogy for a large set of individuals with chromosome-scale numbers of genetic markers might be impossible with the computational resources at hand. There is also a case to be made for preferring heuristics when it comes to this kind of punctual estimation. That being said, CWR-based programs can also be used for purposes requiring the generation of multiple ancestries, such as interval estimation or numerical integration. Since both require generating multiple ancestries, heuristic-based approaches that invariably produce the same output given a particular input are not an option. Even those that introduce randomness are probably not ideal choices, as the quality of the genealogies is generally a primary concern for those tasks. This is, of course, unfortunate, as their number is secondary in importance.

2.2 Supported events

Perhaps the most common way of reducing the computational burden is to reduce the space of admissible events. Early programs were more or less direct implementations of mathematical developments, a list of which can be found in Lewanski et al. (2024). Before long, algorithms avoiding the simulation of non-informative events were developed for performance's sake. Starting in the mid-2000s, simulators prohibiting some informative events as well were developed. While this subclass of applications can only approximate the coalescent with recombination, they can achieve remarkable performance that is often orders of magnitude superior to that of their exact counterpart.

2.2.1 Coalescence events

It is customary to categorize coalescence events based on the ancestrality structure of the generic material associated with the coalescing sequences. We distinguish between material inherited by the sample and material that is not. The first kind is called *ancestral* material, while material of the second kind is called *non-ancestral*. Given two sequences, we classify their coalescence as a *type A* event if some of their ancestral material overlaps. Otherwise, the event is deemed to be of *type B*. Regardless of its type, the occurrence of any coalescence event has the potential to affect the composition of the sample. Consequently, the choice of disregarding a particular type is made primarily, if not solely, for performance reasons. Figure 1 shows examples of both types of coalescence events.

Type A events have never been disregarded in practice. There are no known performance gains to be achieved in this way. On the other hand, denying type B events led to the now-famous *sequential Markov coalescent* (SMC). This algorithm builds on a well-known article by Wiuf and Hein and overcomes one of its limitations. In Wiuf and Hein (1999), they conceptualize the ARG in a novel way as a coalescent tree to which edges and vertices representing recombination and recoalescence events are added iteratively. They call their construct the spatial algorithm graph (SAG) to contrast it with earlier *temporal* algorithms that build graphs backward in time from a sample. Since then, the adjective "spatial" has been used to refer to any ARG construction algorithm that moves along sequences rather than time.

This shift in perspective, however, does not provide any fundamental increase in performance. The main reason is that to correctly follow the CWR distribution, recombination locations must be modeled as a Poisson process on the complete graph at each step of the algorithm. Selecting the next event is consequently more demanding than the last, compounding the computational burden until the end of the sequences is reached. McVean and Cardin (2005) studied a slight modification of the SAG where the location of a recombination event is constrained to the sub-graph induced by vertices ancestral for a position. This induced sub-graph is a full binary tree and is called the *marginal tree* of the position. Their algorithm is Markovian insofar as, at any given step, the recombination event conditional on the current marginal tree is independent of the remainder of the graph. This slight modification has a marked impact on the algorithm's performance, as the compounding effect mentioned earlier no longer occurs. From a computational complexity standpoint, a Markovian model of the dependence structure of sequence segments opens the door to linear-time algorithms with respect to the sequence length. Of course, as noted by Wiuf and Hein, the distribution of ancestries produced in this way only approximates the "true" distribution of the ancestral recombination graph. As noted by Cardin and McVean, their scheme is equivalent to disallowing type B events. Nevertheless, the SMC algorithm, which exploits this approximation by operating on sequences of trees rather than graphs, is arguably the best-known

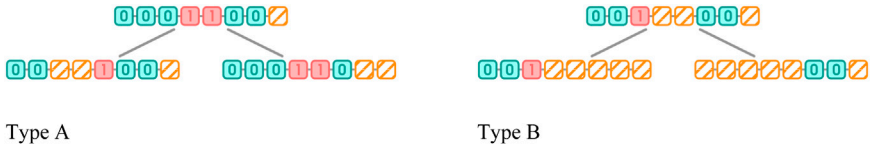


FIGURE 1
Examples of coalescence events.

result from the work of Cardin and McVean. More flexible SMC-type approximations allowing events to be located on the sub-graph induced by vertices ancestral for an interval rather than a singular position were developed later.

Theoretical results concerning the effects of the SMC approximation on parameter estimation are generally difficult to obtain, but some are available. Wilton et al. (2015) used a two-locus model to show equality in distribution between the pairwise joint coalescence time at two recombination sites under ARG and SMC' (Marjoram and Wall, 2006), a slightly modified version of the SMC algorithm, along with some more general numerical results. Chen et al. (2014) showed equality in distribution between ancestries simulated by what they call a *typical back-in-time algorithm* (`ms` with minor modifications) and their own generalized version of the SMC called SC. Their algorithm offers a spectrum of approximations ranging from the SMC, where events' locations are restricted to the current marginal tree, to no restriction at all. A brief overview of the main differences between SMC and SMC' is available in Section 3.8.2.

Simulation studies about SMC's accuracy are more readily available, starting with the original article McVean and Cardin (2005) and Cardin's own PhD thesis Cardin (2007). Eriksson et al. (2009) showed that the SMC' approximation is generally accurate under various demographic structures regarding the pairwise correlation between TMRCA and the linkage probability of two loci. Hobolth and Jensen (2014), among other things, studied the accuracy of recombination and coalescence rate estimation under SMC and SMC' approximations with respect to their own "natural" Markovian approximation of the CWR. They concluded that SMC' is overall quite accurate. One important caveat is that estimations of the recombination rate might be inflated when it is an order of magnitude greater than the coalescence rate. SMC, however, systematically underestimates the recombination rate. Shchur (2022) showed that the total variation distance between the pairwise joint distribution of the TMRCA for SMC' and the full CWR model is non-negligible under a demographic model of two populations with symmetric migration.

2.2.2 Recombination events

Similarly, recombination events can be classified with respect to their relation to the ancestral material of the sequence on which they occur. We distinguish between five types of events. The presence of ancestral material on both sides of the recombination position is the defining characteristic of the first two types. Such an event is of type 1 if it occurs within ancestral material and of type 2 otherwise. Type 3 and type 4 events correspond to cases in which ancestral material is present only to the left or the right side of the position, respectively. An event positioned on a sequence devoid of ancestral material is of type 5. Figure 2 illustrates these recombination event types.

Unlike coalescence events, some types of recombination events have no bearing on the sample. The outcomes of events of types 3, 4, and 5 are exclusively either non-ancestral sequences that can be safely disregarded or clones of existing haplotypes. These events are targeted for optimization by all reviewed software. Events of types 1 and 2, on the other hand, have the potential to be consequential for the sample and are desirable from the point of view of biological realism. Exact algorithms are largely indifferent to the type of recombination event; both are equally easy to simulate from a computational standpoint. This is not as straightforward for approximate spatial algorithms. Consider that from a spatial point of view, any recombination event located in the marginal tree of its position must almost, by definition, be of the first type since every edge of the tree is ancestral for that position. This is to be contrasted with the fact that an event of the second type cannot be located in the marginal tree associated with its position. Marginal tree-centric algorithms—such as the original SMC or variations like SMC'—are consequently unable to generate events of the second type. This is almost self-evident for SMC and SMC', as the kind of trapped non-ancestral material required for the occurrence of type 2 recombination can only be created through type B coalescence, which they deliberately omit. Trapped material and type 2 events may (Wang et al., 2014; Mahmoudi et al., 2022) or may not (Rasmussen et al., 2014) have a significant impact on the sample.

We summarize this section as follows: when optimizing model-based algorithms, there is a tradeoff to be made between realism and performance. The main optimization target concerns trapped non-ancestral material, which is a byproduct of type B coalescence events. Denying such events leads to a Markovian correlation structure and a linear-time algorithm. However, ancestry produced this way also lacks type 2 recombination events. Interestingly, it is possible to generate type B coalescence while excluding type 2 recombination events (see Section 3.5.1). Finally, we do not want to give the impression that approximating the CWR is the only possible avenue for optimizing algorithms for performance. While it may be the case that the correlation structure of sequence segments is not Markovian, precluding the existence of linear-time exact algorithms, there is more to the story than asymptotic observations, since sequences of genetic markers are, by their very nature, discrete and finite objects. While it is certainly of theoretical interest to reduce the computational complexity of ARG simulation and inference, the sole practical requirement (disregarding restrictions related to data storage) is to bring the duration of simulations below a

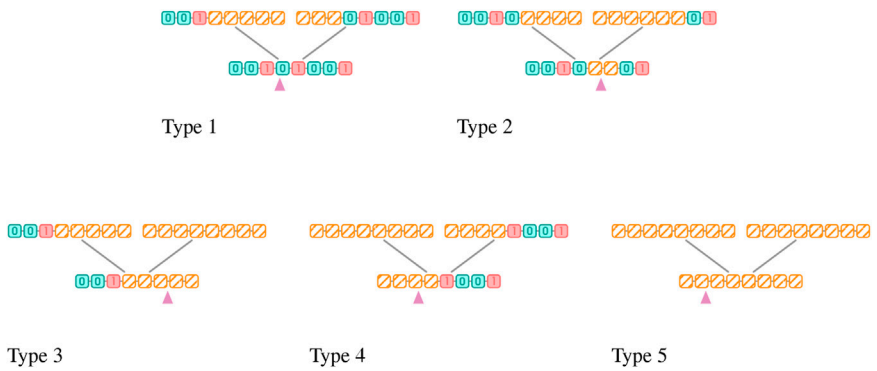


FIGURE 2
Examples of recombination events.

determined threshold for a given sample. These two objectives are of course related, but the importance of efficient implementation in reducing the time necessary for actual computations cannot be overstated (see Section 3.1.5).

2.3 Inferred vs. simulated

Although these adjectives have been used in the literature for quite some time, they have not been formally defined, to the best of our knowledge. Consequently, we begin this section with the following tentative definition: *ARGs simulation* aims to produce one or multiple ancestries according to a specific genetic model such as Wright–Fisher (WF) or CWR. *ARGs inference* aims to produce one or multiple ancestries that may have given rise, according to some criterion, to a sample of haplotypes.

In practice, parameters for ARG simulation are often estimated from a sample of the target population. These might include, but are not limited to, values such as recombination rate, mutation rate, effective population size, and demographic model. The output can be used to estimate various quantities of interest. Such an approach might seem overly complicated at first glance, but it is very natural when the quantity of interest is ancestry-related. A classic example is the TMRCA of a chromosomal segment.

Inputs for ARG inference software, especially model-based ones, generally include parameter estimates as well. All of them require an initial set of haplotypes to construct ancestries. Although they may aim to produce genealogies distributed according to a specific genetic model, at least approximately, this goal is secondary to consistency with the input haplotypes. The most common notion of consistency, at least for software focusing on human genealogies, stems from the *infinite-site model* (ISM) for mutations by Kimura (1969). The basic idea is that as long as the locations of the mutations are supported on the continuum, the probability of more than one mutation event occurring at the same site is extremely low. From this point of view, consistency can be informally defined as follows: an ancestry is consistent with a sample if there is a way to locate mutations on its edges that give rise to the set of input sequences while mutating each locus at most once. The ISM might appear as a strong assumption, but it may be plausible when working at the nucleotide level, as noted by Kimura. Considering that the human genome is characterized by a relatively large number of nucleotides and a relatively low mutation rate, it is arguably a reasonable model for nucleotide-level markers such as SNPs.

From this point of view, it may appear as though model-based approaches rely on a minimization heuristic for the number of mutation events, much like their parsimony-based counterparts. While there might be some truth in that statement, we would like to point out that the infinite sites model is a model, not a heuristic. It simply happens to entail parsimony of mutation events. Inference software applications based on a different set of assumptions are possible, at least in theory. The schism between the model and parsimony occurs at the level of recombination events.

2.4 Programming language

The programming language used to implement an algorithm may be of interest for certain use cases. One might, for instance, desire to modify a software application to suit their needs, a task that is easier done in a familiar language. The language a software application is coded in can also provide some hints about its characteristics. It is reasonable to expect a `C` program to be more efficient but also less user-friendly than a `Python` one, for instance.

It is becoming more common for software to be written in multiple languages. A given language is often designed to prioritize one or at most a few aspects of software development. Understandably, programmers want the best of every world, a desire that has become increasingly easy to satisfy. With that in mind, we will classify programs according to the language used to implement the actual algorithm. This decision is natural since, among all the concerns addressed by programming languages, computational performance usually comes first. Ease of use, addressed by the next criterion, is next on the list.

2.5 Interface

The interface of a software application is to be understood here in a broad sense. It includes any way a user or another software application might interact with the algorithm's implementation. For example, we consider a program for which there exist `Rust` bindings as having a `Rust` interface. Another example is software

implementing an algorithm in C while providing a Python application programming interface (API). We would classify such an application as being coded in C and providing a Python interface.

In order for an interface to be taken into account, two conditions must be met: (1) it must be documented, and (2) a *significant* part of the program must be exposed through it.

An interface does not need to be a programming language. A command-line interface (CLI), for instance, might be a valid interface.

3 Software review

Programs are grouped by family where appropriate and then listed in chronological order. Properties of the software are summarized in short tables. Column titles have the following meaning:

- M: model-based (vs. heuristic-based);
- C (B): supports type B coalescence events;
- R (2): supports type 2 recombination events;
- S: simulated (vs. inferred);
- L: programming language;
- I: interface.

When a property is binary (e.g., M), a checkmark (✔ or ✔) or a cross mark (✖ or ✖) indicates the presence or absence of the property, respectively. A green or red checkmark indicates desirable or undesirable features, respectively. The opposite is true for crosses. Only the principal interface of programs with many interfaces is listed. An asterisk "*" indicates that other interfaces exist and are discussed in the associated subsection. Unknown status is indicated by a yellow question mark (?).

3.1 ms family

Simulators in this family implement algorithms based on Hudson's original work (Hudson, 1983, Hudson, 1991). They are model-based temporal simulators that are able to generate all types of informative coalescence and recombination events. They generally provide a high degree of accuracy with respect to the CWR.

3.1.1 ms

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C	CLI

ms is widely regarded as the reference CWR implementation against which others' statistical performances are benchmarked. To avoid confusion, we reserve the name ms for the simulator itself and refer to the algorithm it implements as "*Hudson's algorithm*" (Hudson, 2002).

Every informative coalescence and recombination event is supported, and none of the non-informative ones are ever simulated. In addition, ms supports multiple extra features:

1. Gene conversion.
2. Demographic events.
3. Migration events.
4. Inclusion of ancient DNA.

Gene conversion functionalities were added in a 2001 update. When requested by the user, events are simulated according to Wiuf and Hein (2000), a model based on the Holliday Junction (Holliday, 1964). Simulations can be carried out with any of the four possible combinations of chromosomal crossover/gene conversion events. Migration events follow an island model. An arbitrary number of subpopulations with variable sample sizes can be specified. It is even possible to introduce population splits at predetermined times. The migration matrix is entirely customizable by the user and can even vary as a function of time. Similarly, growth rates and population sizes can vary dynamically. Ancient chromosomes can be placed anywhere back in time and in any subpopulation.

All in all, the flexibility of ms, caused by its extensive set of features, is impressive, especially at the time of its release. From a computational performance perspective, ms is generally regarded as efficient. This claim is largely based on the types of events it supports, which are limited to informative ones. Of course, its performance is not on par with programs implementing approximations to the CWR, but such a comparison would be unfair. Nevertheless, ms's performance is not sufficient to meet the requirements of most contemporary genetic analyses, which are often conducted with large samples and on a genome scale. For instance, Kelleher et al. (2016) exclude ms from their "performance analysis" section. The time required to complete simulations in the most demanding scenarios is much longer than for more recent benchmarked simulators.

Hudson's algorithm is implemented as a loop that continues as long as the sample's MRCA has not been reached; it is the archetypical backward-in-time simulator. At each iteration, a type of event (coalescence, recombination, mutation, or migration) is generated. The current graph is then altered in accordance with the outcome. This description is rather high-level, as the program itself does not store the ARG itself but rather the corresponding sequence of marginal trees.

Conveniently, ms can output this sequence, i.e., each tree encoded in Newick format, upon a user's request. As noted by Hudson, this is convenient when interaction with other tools is desired. On the other hand, as seen in Section 3.1.5, ms's data structure is also its Achilles heel; it simply does not scale, neither with the sample size nor the sequence length. Interestingly, parallelized versions have been developed (see Section 3.1.6).

3.1.2 msms

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	Java	CLI

msms aims to improve on ms by adding selection capabilities. It is, nonetheless, able to construct neutral ancestries (i.e., without selection) and is equivalent to ms when used in that way. Moreover, their command-line interfaces and text outputs are identical. All of this makes msms a drop-in replacement for ms. The authors claim that its computational performance is generally on par with that of ms. However, there are reports that it outperforms the original, especially in scenarios involving high recombination

rates and/or large sample sizes (Kelleher et al., 2016; Ewing and Hermisson, 2010).

If requested by the user, selection is integrated into the CWR model by simulating ancestries conditional on the major allele's frequency (MAF). MAF is computed prior to the actual ancestry simulation in a forward pass involving a structured Wright–Fisher model with selection. In the case of a spatially structured population, fitness values can be specified on a per-deme basis.

Although `msms` implements Hudson's algorithm, it differs substantially from `ms` internally. While both programs' interfaces and output are identical, `msms` is actually implemented in `Java` rather than `C`. As it turns out, the performance is not negatively impacted by this choice. In addition, since `Java` runs on a virtual machine, `msms`' portability is superior to that of its predecessor, which is a significant advantage for reproducible research and open science. Moreover, since `Java` is a higher-level language than `C`, `msms` code is arguably easier to understand and modify.

3.1.3 `cosi2`

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C++	text

Published over a decade after `ms`, `cosi2` innovates in its implementation of Hudson's algorithm. At the time of its release, simulating long sequences was exclusively the domain of software implementing approximations to the CWR, all of which lacked selection capabilities. Furthermore, no simulator, approximate or exact, had the ability to generate both selection and gene conversion events. An efficient implementation of Hudson's algorithm would allow simulation of these complex scenarios, which is precisely what `cosi2` aims to achieve. The software's main innovation compared to the original algorithm is its clever exploitation of the backward-in-time CWR Markovian property. At each step of the simulation, the next state can be chosen without knowledge of anything but the current one. Consequently, `cosi2` discards unnecessary information after each transition. Of course, just because some information can be disregarded at some point in the simulation does not mean that it is irrelevant to the final result. It turns out that the complete ARG can be reconstructed by recording, at every step and for every node n, the function that maps an interval x to the set of sequences in the sample that inherit material in x from n. This can be carried out efficiently using skip lists. Gains in terms of execution time are substantial, even more so for the approximate version of `cosi2`, which restricts coalescence events to type A (Shlyakhter et al., 2014).

Simulation parameters are set in a text file using a custom format. The output is compatible with that of `ms`'.

3.1.4 `discoal`

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C	CLI

The main feature of `discoal` is its support for fixation of neutral alleles caused by linkage with a beneficial mutation, which is a phenomenon known as *selective sweep*. Both hard (polymorphism elimination) and soft (polymorphism reduction) events are supported (see Pennings and Hermisson (2006) for a detailed and more nuanced presentation). Other features include recurrent mutation, gene conversion, fixation of arbitrary mutations, and support for complex demographic scenarios (Kern and Schrider, 2016).

`discoal` is the simulator used by `selscan` (Szpiech, 2024).

3.1.5 `msprime`

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C	Python*

It is hard to overstate the importance of `msprime` in the current landscape of CWR simulators. Before its publication, it was largely accepted that large-scale simulations were only possible through approximations. `msprime` demonstrated that a careful implementation of Hudson's algorithm could, for all intents and purposes, outperform sequential approximations even on genome-scale datasets (Baumdicker et al., 2021).

`msprime`, as its name indicates, aspires to be an improved version of `ms`. Given the consensus around Hudson's program's efficiency, it might appear almost impossible to achieve significant improvements. While this may well be true from a computational complexity standpoint, it may still be possible to realize practical gains by optimizing the underlying data structure. Ancestries produced by `ms` are stored as sequences of marginal trees, which is a natural choice given the nature of the data. This design is nonetheless inefficient: each recombination event requires the creation of a new marginal tree with a number of vertices and edges on the order of the sample size. The overhead is arguably negligible for modest sample sizes and recombination rates. However, on the genome scale, as evidenced by `ms`' poor performance in that context, the spatial and temporal demands prove to be a significant limitation.

`msprime`'s solution exploits the high degree of correlation between nearby marginal trees. The first key insight is that two adjacent trees are nearly identical. Given the left tree, little information is necessary to recover the right one. Earlier versions of `msprime` encoded this information in the form of a sequence of *coalescence records*, dubbed a *tree sequence* (Kelleher et al., 2016). Each record tracks a coalescence event, which is defined as the merging of two segments with ancestral material in a common interval into a single segment. *Common ancestor events* are distinct from *coalescence events*, with the former events including type A and B events and the latter being reserved for type A. In any case, let $c = (a, b)$ be a coalescing pair, w the segment they coalesce into, t the latitude of w, and $[l, r)$ the shared ancestral interval. The corresponding coalescence record is the 5-tuple (l, r, w, c, t). Notably, the merging of two segments with multiple disjoint common ancestral intervals generates multiple coalescence records, one for each shared interval. Hence, from a topological perspective, a tree sequence is a multi-graph. The second insight is as follows: if a sample has reached its MRCA for an interval $[l, r)$ at time t, there is no fundamental need to keep track of the associated marginal tree before t, as it cannot possibly contain information about $[l, r)$ relevant to the sample; all relevant information is contained in the records preceding the one corresponding to the MRCA of $[l, r)$. Consequently, there is no need to simulate the complete ARG to obtain a sample's ancestry; it is sufficient to generate a valid tree sequence. This can be achieved without

tracking "partially built trees," only by maintaining a set containing ancestral segments. This realization is the main source of improved performance of msprime over ms, as storing and managing these trees is a major source of overhead for the latter. Notably, this formulation of Hudson's algorithm allows for other optimizations as well, such as segment defragmentation with adjacent intervals. As of version 1.0 (Baumdicker et al., 2021), the tree sequence data structure is generalized, primarily to support supplementary information such as mutations. Nevertheless, the main ideas underlying msprime's improved performance remain unchanged.

Relying on tree sequences renders random access to marginal trees impossible. This is perhaps the main drawback of msprime's choice of data structure. Although, this is arguably not a major concern as many common operations only require *sequential* access, which can be achieved efficiently via Baumdicker et al. (2021)'s algorithm T.

msprime is distributed as a Python package and can be installed via Pip or Conda. This allows for seamless integration into a Python workflow, which is a welcome convenience given the widespread adoption of this language in bioinformatics and data science. Online documentation also mentions the availability of a container image on Dockerhub, streamlining execution via Docker, Podman, or even Kubernetes. However, as of March 2025, the image lagged behind msprime's last release by over 4 years. Simulations can also be run through a command-line interface. Two applications are provided for this purpose. The first, msp, provides full access to msprime's functionalities. The second, mspms, implements ms' interface and is intended as a drop-in replacement for Hudson's program.

Msprime's output is a tskit (Kelleher et al., 2018) object of type treeSequence. This allows for considerable flexibility. The tskit ecosystem is vast and offers a wide range of quality software. Many analyses can be conducted within its bounds. Should integration with other languages be needed, APIs are available for Python, C, and Rust. If integration with R is desired, tskit's authors recommend interfacing through reticulate (Ushey et al., 2025). Finally, tskit can output tree sequences as plain text files.

Even though msprime is first and foremost an implementation of Hudson's algorithm, other models are also available. A discrete backwards-in-time Wright–Fisher model (Nelson et al., 2020), which can be thought of as a discrete-time CWR simulator, is provided. Its inclusion aims to address biases caused by long-range correlation emphasized in ancestries constructed from large samples of long sequences. Hybrid schemes where recent history is simulated according to the discrete model and the remainder is simulated under classical CWR are possible. SMC and SMC' approximations are implemented via a simple rejection sampling. Although the distributions are correct, the underlying algorithm remains Hudson's algorithm and does not take advantage of optimization opportunities offered by Markov approximations. These are clearly intended to facilitate quick comparisons and experimentation and are not "production grade" solutions. Models including selective sweep events are supported via an implementation of discoal's algorithm (see Section 3.1.4). Moving beyond the standard CWR, backwards-in-time pedigree, beta coalescent, and Dirac coalescent models are also available.

3.1.6 Other programs

- msHOT (Hellenthal and Stephens, 2006): It allows for user-specified recombination hotspots. In Hudson's original algorithm, recombination breakpoints are generated from a homogeneous point process. msHOT gives users the ability to vary the process' intensity along sequences.
- mbs (Teshima and Innan, 2009): Flexible simulation of sophisticated demographic scenarios for bi-allelic sequences.
- msPar (Montemuiño et al., 2014): Parallel implementation of ms via MPI.
- msParSm (Montemuiño et al., 2016): Similar to msPar but includes various optimizations. msParSm is able to construct genealogies for "long" sequences under a high recombination rate.

3.2 SIMCOAL family

This family comprises two simulators developed primarily by Laurent Excoffier from 2000 onwards. In addition to simulation capabilities, these programs can output data in a format suitable for ARLEQUIN (Excoffier and Lischer, 2010), allowing for the computation of summary statistics and further analysis.

3.2.1 SIMCOAL2

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C++	CLI

Up until version 2.0, SIMCOAL did not support recombination events (Excoffier, 2000). It was essentially a coalescent simulator with broad support for demographic events. Unlike many similar programs, it did not rely on a continuous time approximation, instead using an admittedly slower discrete generation approach. While detrimental to performance, discrete time facilitated the implementation of support for multiple events per generation. Version one of SIMCOAL had the ability to generate multiple demographic events per generation, a capacity extended to coalescent and recombination events in version two. Laval and Excoffier (2004) argued for the necessity of multiple coalescence events by generation in a discrete framework on the basis of the potential for a rapidly increasing number of lineages under the effect of recombination. From that point of view, SIMCOAL2 is well-suited for samples whose size is not significantly below the effective population size (Laval and Excoffier, 2004).

Despite the article's lack of details about the algorithm, the source code reveals that the first two types of recombination events are supported exclusively; the function TNode::recombinationPosition is particularly revealing in that respect. Similarly, TDeme::singleCoalescentEvent and TDeme::multipleCoalescentEvents indicate that both types of coalescence events are supported.

SIMCOAL outputs summary statistics about both the sample's ancestry and the ancestry itself. The statistics are contained in an ARLEQUIN-formatted file. The ancestry is available in the form of a sequence of NEXUS-encoded marginal trees in a separate file.

Ancient DNA is supported via an extension. Serial SimCoal (Anderson et al., 2004) adds support

for the so-called serial coalescent model. In addition to simulation facilities, it includes tools for statistical analysis of generated data.

3.2.2 fastsimcoal

M	C (B)	R (2)	S	L	I
✔	✖	✖	✔	C++	CLI

fastsimcoal is a complete rewrite of SIMCOAL2. It shares its ability to deal with highly complex demographic scenarios, but its simulation approach is radically different. Its focus on performance contrasts with the more relaxed stance of its predecessor. It completely abandons SIMCOAL2's original time-discretized approach in favor of a classical continuous-time approximation. In fact, fastsimcoal's algorithm is an implementation of SMC' that is customized to support migration between demes (Excoffier and Foll, 2011).

Serial sampling capabilities, previously enabled by an extension, are now integrated into the main program. Another addition to fastsimcoal is the possibility of specifying the distributions of various genetic parameters instead of the parameters themselves, facilitating the application of statistical methodologies such as approximate Bayesian computation. Support for recombination hotspots completes the list of new functionalities.

Version 2.0 (Excoffier et al., 2021) has many new features, most of which relate to parameter estimation and inference. Regarding performance, the main improvement involves concurrent computing, with multithreading available for independent simulations. As the implementation is carried out via OpenMP, this is limited to shared-memory machines. Approximations to transcendental functions, along with some other unspecified optimizations, also contribute to reducing the total computing time.

Fastsimcoal is largely backward compatible with SIMCOAL2. They share a similar format for input files. In particular, their input files share the same format, with the exception that fastsimcoal requires the specification of polymorphic sites exclusively and the addition of a syntax for haplotype age specification, with omission interpreted as present time.

Simulation studies reported in the original applications note indicate that fastsimcoal produces data similar to ms and MaCS while being consistently faster. Documentation is extensive, with many screenshots and a wide selection of examples. Binaries for major operating systems are distributed directly on the author's website. Unlike SIMCOAL2 and most software oriented toward scientific research, the source code is not publicly available.

3.3 SHRUB family

This family comprises three related ARG inference pieces of software that exploit heuristics to construct parsimonious genealogies. Their algorithms share a similar aggressive cleaning/reduction step and a simplified recombination procedure. SHRUB and beagle primarily focus on computing bounds on the number of recombination events. However, ancestry topologies are simulated in the process and can be recovered afterward.

3.3.1 SHRUB

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	?	?

SHRUB is an acronym for "simulated history recombination upper bound." It is a heuristic algorithm revolving around a quantity called *recombination weight*. A haplotype r has a recombination weight $w(r\,|\,A-r)$ with respect to a set of haplotypes A if the smallest number of sequences in $A\backslash r$, that r is a mosaic of, is $w(r\,|\,A-r)+1$. From a biological standpoint, this quantity can be interpreted as follows: a haplotype with weight $w(r\,|\,A-r)$ can be coalesced into A after $w(r\,|\,A-r)$ recombinations and $w(r\,|\,A-r)$ coalescence events (Song et al., 2005).

SHRUB's primary function is to compute an upper bound on the minimum number of recombination events for a sample of haplotypes, which is equal to the minimum of the sums of the recombination weights associated with the ancestries it produces. This sum can be interpreted as the number of recombination events in the best genealogy that a naïve parsimonious algorithm can generate. There are two versions of SHRUB. In its full version, the exact upper bound is computed by branch and bound. Data are represented, at least conceptually, as a matrix, and biological events are equated to row and column deletions. Coalescence and recombination events correspond to the removal of a row, while mutation events correspond to column deletions. The algorithm alternates between two stages; the first greedily performs every currently possible coalescence and mutation events. A row can be removed if it has a recombination weight of zero, meaning that it is equal to some other row in the matrix. This corresponds to a coalescence event. For a column to be a candidate for deletion, it must have a recombination weight no more than one 0 or one 1, a requirement in line with the ISM. The second step involves removing a row and adding its weight to an accumulator before returning to the first stage. When only a single row remains, the accumulator contains the accumulated recombination weight of the ancestry. This quantity is stored in memory, and the algorithm restarts to explore a new ancestry.

The algorithm, as described, requires an exhaustive search to compute the upper bound. As the authors noted, minor branch and bound-type modifications significantly increase efficiency. If, at any point, the accumulated weight is greater than the current minimum, the ancestry can be discarded. Similarly, if the current state has been reached in another run of the algorithm, the partial accumulated weights should be compared, and the current run should be interrupted if its weight is greater. This approach markedly improves performance, reducing the time complexity from $\mathcal{O}(n!)$ to $\mathcal{O}(2^n)$. If this speedup is not sufficient, one can use the fast version of the algorithm, where the removed row is chosen to be a minimizer of $w(r\,|\,A-r)+L(A-r)$, with the right term being a (possibly approximate) lower bound on the minimal recombination number of $A\backslash r$.

Unfortunately, we were not able to find SHRUB's source code. Since the article makes no mention of the language used to program it, we have no way of determining it, although C would be a natural choice. Binaries are available on the author's website (the link provided in the original article is no longer valid). Similarly, unlike HapBound, the other program presented in the article, no details about SHRUB's usage are given, leaving its interface undetermined

for the reader. One could reasonably suppose, however, that it is similarly controlled via a command-line interface.

3.3.2 Beagle

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C	CLI

Beagle implements a heuristic designed to find parsimonious ancestries via a branch and bound algorithm. The constructed ancestries are truly parsimonious; they effectively attain the recombination lower bound. Beagle assumes the infinite-site model, bi-allelic markers, and polarized data—i.e., the wild allele is known for each marker. Missing data are supported and treated as non-ancestral material (Lyngsø et al., 2005).

The algorithm starts by computing a lower bound for the number of recombination events. This is carried out via a custom algorithm generalizing RecMin.c (Myers and Griffiths, 2003). The next step is to attempt to find an ancestry realizing this lower bound, a problem that beagle's algorithm proceeds to solve in a somewhat greedy fashion. First, a reduction step, during which every possible coalescence and mutation event is performed, is repeated until exhaustion. Only then are recombination events considered. A maximum of two consecutive recombination events, which are explicitly restricted to types 1 and 2, are selected, after which another reduction step is executed. This back-and-forth between reduction and recombination is repeated until either a complete ancestry is found, in which case the algorithm finishes, or the number of recombination events exceeds the lower bound. In that event, the bound is increased by one, and the algorithm backtracks. Importantly, beagle computes a new lower bound every time a new state is reached and backtracks early if the minimum number of recombination events required for completion is higher than currently allowed. To avoid re-exploring unfruitful paths, each state along with the associated lower bound is stored in a hash table.

Biological events are formulated as operations on the set of sampled haplotypes. A coalescence event corresponds to the removal of a sequence from that set, while a mutation event is realized by removing a given marker from every haplotype. Recombination events are slightly more complex, as they demand the removal of one haplotype followed by the addition of two sequences. These operations are subject to restrictions to enforce their biological soundness under the assumptions made by the algorithm. Sequence removal takes place if there is at least one other sequence identical to the one being removed in the set. A marker cannot be removed unless there is at most a single sequence carrying the derived allele. The new sequences generated from a recombination event must be equal to the old one to the left (respectively right) of the breakpoint and be entirely non-ancestral to its right (respectively left).

Beagle is more efficient than Song and Hein (2003) since its branch and bound approach avoids enumeration of marginal trees, a claim that is supported by the simulation studies reported by the authors. Unfortunately, we were unable to find the program's source code or binaries online (the link given in the article is no longer working).

Beagle should not be confused with the popular genotype phasing and imputation software BEAGLE (Browning et al., 2018, Browning et al., 2021).

3.3.3 KwARG

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C	CLI

KwARG is described by its developers as parsimony-based, greedy, and heuristic. At first glance, its most striking feature is perhaps its somewhat relaxed adherence to the ISM framework. Technically, it is perfectly capable of producing consistent ancestries in the usual sense. However, unlike the majority of ARG inference software, this is offered as an option; given an appropriate set of parameters, a recombination-free ancestry will be produced. KwARG offers users the possibility to interpolate, so to speak, these two radically different approaches (Ignatieva et al., 2021).

Under the hood, KwARG's algorithm relies on matrix-like data structures, interpreting biological events as operations on the rows and columns of matrices. Sampled haplotypes are stored in a ternary matrix (0, 1, and*) in order to account for non-ancestral or missing material, which are dealt with in the same way. Each iteration of the algorithm begins with repeated application of a cleaning step taken from Song and Hein (2003), which is nothing more than the reduction step of beagle with an additional rule allowing the removal of all zero columns. Subsequently, a set of candidate next states is created by generating either a recombination or a (recurrent) mutation event. Restrictions on admissible events are the same as for beagle. After cleaning, a score is attributed to each candidate state in a fashion resembling the A*algorithm (Hart et al., 1968). The cost associated with a given state is a function of user-defined per-event costs and a local estimate of the number of events to termination. The next state is chosen to minimize the cost, but a probabilistic component is introduced in an attempt to compromise between exploration and exploitation.

KwARG compares favorably to exact solutions when computing parsimonious ancestries, both under models that allow for and prohibit recurrent mutations. Remarkably, KwARG can be used to study organisms with a high mutation rate, such as viruses, which is an avenue explored by the author in a subsequent study (Ignatieva et al., 2022).

Source code, binaries for major operating systems, and small utility programs are distributed via GitHub.

3.4 Margarita family

This category includes programs aiming at inferring so-called "plausible" or "minimal" ARGs via a heuristic approach related to that implemented by Margarita. Margarita itself is similar to SHRUB in some respects. According to the authors of the former software, these resemblances are coincidental. Consequently, we have decided to split them into distinct families.

3.4.1 Margarita

M	C (B)	R (2)	S	L	I
✖	✖	✖	✖	Java	CLI

The primary motivation behind Margarita is the fine mapping of diseases via inferred ARGs. Even though the implemented algorithm is heuristic-based, the authors clearly state that their guiding principle is not the minimization of the number

of recombination events. They aim for constructed ARGs to be "plausible," a term that they unfortunately do not define further. In any case, their method is efficient, allowing for sample sizes in the thousands and sequences of hundreds of SNPs. The authors suggest that Margarita could be used on the genome scale, albeit in a windowed fashion (Minichiello and Durbin, 2006).

The heuristic is guided by a few principles. Coalescence events are restricted to type A. Regarding mutations, back or recurrent ones are prohibited. In consequence, mutation events for a given marker can only occur when there is a single sequence having the derived allele and must occur on that sequence's parent. Recombination events are chosen to allow immediate coalescence, which is indeed performed directly afterward. It may be necessary to actually perform two recombination events before coalescence is possible, a possibility that Margarita is designed to handle. Nevertheless, recombining sequences are chosen to maximize the shared material. Under these rules, coalescence and mutation events are produced greedily until exhaustion. At that point, a recombination event is sampled. This process is iterated until the sample's MRCA is reached.

From a practical standpoint, Margarita can deal with missing and unphased data. Absent SNPs are imputed *ad hoc* by basically being treated as non-ancestral material. The same idea, with additional restrictions, is used for unphased markers.

3.4.2 ARG4WG

M	C (B)	R (2)	S	L	I
✖	✖	✖	✖	C++	CLI

The main purpose of ARG4WG is to enable the analysis of genome-sized datasets. Its authors acknowledge the performance of Margarita while noting that it is still not fast enough to handle "thousands of whole chromosome samples." Their software is a reimplementation of Margarita that incorporates a modification to the recombination step (Nguyen et al., 2017).

The recombination step of Margarita's algorithm selects a haplotype to maximize the amount of shared ancestral material in the subsequent coalescence events. Since the shared segment can be flanked by unshared ancestral material, up to two recombination events may be necessary before coalescence is possible. ARG4WG's algorithm is similar but restricts recombination events even further: every single event must *immediately* lead to a coalescable sequence. This is enforced by selecting candidates on which the shared material is positioned at one of its ends or at least only separated from the end by a non-ancestral segment. ARG4WG's algorithm is justified by a lemma stating that given a set of reference haplotypes and an additional haplotype to be coalesced into it, a greedy left-to-right procedure repeatedly generating recombination events and coalescing the resulting sequence with ancestral material to the left of the breakpoint in a way that maximizes the amount of shared ancestral material minimizes the number of recombination events. This result is similar to the lemma 4.2 of Song et al. (2005). In addition, the search for haplotype pairs with shared ends can be conducted in parallel, which further speeds up execution.

Although biological realism is not a priority, Nguyen et al. (2017) shows results that are in line with Margarita, and the results of their application to the Gambia dataset agree with other analyses. Regarding performance, the results are undoubtedly impressive. For instance, ARG4WG was able to construct an ARG for a sample of 5,560 haplotypes containing data for the whole chromosome 11 in just over 3 hours using 16 threads.

ARG4WG's source code is distributed on GitHub. A short readme containing the compilation and execution instructions is available.

3.4.3 GAMARG

M	C (B)	R (2)	S	L	I
✖	✖	✖	✖	C++	CLI

GAMARG aims to reduce the number of recombination events even further than its predecessor ARG4WG, to which it is closely related. The main modification is the integration of the well-known *four gametes test* (Hudson and Kaplan, 1985) into its algorithm. Since this test operates pairwise on markers, it does not scale to the sample sizes that GAMARG is intended to handle. Consequently, it actually implements a windowed and weakened version of the complete test (Thao and Vinh, 2019).

This addition to ARG4WG's algorithm has important consequences. One of the important features of ARG4WG is that the number of live sequences never increases. Any recombination event added to the ancestry is chosen to enable a specific coalescence event that follows immediately. This is no longer the case for GAMARG. After every coalescence and mutation events have been exhausted, a "classic" recombination event, which is one not followed by a coalescence, is generated if there is at least one sequence bearing a pair of incompatible sites.

Simulation studies and application to real datasets indicate a slight reduction in the number of recombination events. However, the execution time is considerably above that of ARG4WG.

3.4.4 Other programs

- REARG (Phuong Thao and Vinh, 2017): It considers both the "similarity between sequences" and the "length of sequences" when generating recombination events. Since we could not access the article or the software's source code, our summary is based on Thao and Vinh (2019).

3.5 MaCS family

This section groups simulators implementing a flexible ARG construction algorithm related to that of MaCS.

3.5.1 MaCS

M	C (B)	R (2)	S	L	I
✔	✔	✖	✔	C++	CLI

The main appeal of exact CWR simulators such as those belonging to the ms family is the possibility of obtaining accurate samples from the distribution of interest. In addition, some of them even allow for rich demographic and even selective models. However, before msprime, they were more or less relegated to the rank of academic tools due to their difficulty in keeping up with the rapidly growing scale of real-world data. SMC approximations are definitely less limited in that regard, but trading biological realism backed by mathematical rigor for computational performance gains can be difficult. MaCS intends to be a middle ground between the

two approaches by implementing a variable approximation of the CWR. The main feature of the SMC algorithm is its Markovian nature: any marginal tree is independent of any previous one, given one in between. This allows for efficient simulation algorithms since only the current tree needs to be considered when simulating the next one. From the point of view of sequential approximations, MaCS' algorithm can be summarized as an SMC-type simulator that generalizes the concept of marginal *tree* to that of marginal *graph*. While the SMC generates the next tree conditional on the current position, MaCS has the ability to do so conditionally on the history of an *interval* of positions. The right endpoint of this interval is the current position, and its length is controlled by the user. Graphically, this interval's ancestry is described by the sub-graph induced by the subset of vertices having ancestral material within it. In general, for an interval, there is no guarantee that this sub-graph is a tree; hence, we use our expression "marginal graph." The number of marginal trees k included in any given marginal graph is determined by a sequence length h specified in the program's invocation. k is chosen to represent the ancestry of an interval of position with width h. Since the number of recombination events within a specified interval is random, this choice is approximate (Chen et al., 2008).

Based the fact that recombining sequences are allowed to recoalesce on the current marginal graph without further restriction, we can infer that both types of coalescence events are possible, providing a sufficiently large h. Recombination events' locations are constrained to the marginal tree associated with their position. Thus, type 2 events are impossible. By the same reasoning, non-informative events are not simulated either.

Regarding code, data structures constitute an interesting departure from ms, which encodes ARGs as sequences of trees. MaCS' approach is to store the complete ARG as a graph. The CLI is mostly similar to ms, and an auxiliary program called msformatter that turns MaCS' output into one compatible with Hudson's ms is provided.

3.5.2 SC

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C++	CLI

The only feature missing from MaCS in order to fully interpolate between Hudson's and SMC algorithms is the capability to generate type 2 recombination events. This is what SC brings to the table. Based on version 0.4e of MaCS, both software applications implement largely similar algorithms, with SC making one important addition to the recombination step. In line with other sequential simulators, MaCS selects recombination locations uniformly on some sub-graph of the ARG conditional on their position on the sequences. While this is also true of SC, this is not the whole story. After one such event has been produced, two things can happen depending on the location of the subsequent recoalescence event. If it is located on the current marginal tree, the algorithm iterates just like MaCS would. If, on the other hand, recoalescence occurs on an older marginal tree, a chain of successive recombination and recoalescence events is started. The next recombination is simulated from the previous recoalescence's location following a set of rules. This process continues until coalescence with the current marginal tree occurs (Wang et al., 2014).

3.5.3 SC-sample

M	C (B)	R (2)	S	L	I
✔	✔	✔	✖	C++	CLI

It is a modified version of SC integrating a minimum mutation number (MMN) algorithm and various constraints that allow ARG inference from a sample of haplotypes. Their MMN algorithm is an edge labeling scheme allowing efficient computation of the minimum number of mutation events that would have been necessary for a marginal tree to have given rise to a sample's specific marker. If this number is larger than one, then the marginal tree cannot represent this marker's history assuming the ISM. Otherwise, it is said to be *consistent* with the marker. A high-level description of the algorithm is as follows: it is started by constructing a consistent marginal tree for the leftmost marker, which is easily done. The MMN algorithm is applied sequentially to each following marker. When a marker m is found to be inconsistent with the last marginal tree, recombination events are generated between that marker and the last one (or the beginning of the sequence) until a new marginal tree consistent with m is built. It is iterated until every marker is associated with a consistent tree (Wang et al., 2014).

While the description of the algorithm in Wang et al. (2014) is slightly unclear with respect to the recombination positions' distribution, a glance at the code confirms that each one is uniformly distributed between the last one (or the position of marker $m-1$) and the m^{th} marker's position. This choice of a support-bounded distribution is a departure from standard theory necessary when inferring ARGs. It would be naive to hope for a sample to be produced in reasonable time otherwise. The distributions of recombination and recoalescence locations are similarly bounded versions of their classical counterparts. Recombination edges are constrained to be mutation edges and to recoalesce in a way that decreases the number of mutations.

3.6 Tree scan family

ARG inference based on the so-called tree scan (Song and Hein, 2005) method.

3.6.1 RENT

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C++/Java	CLI

RENT aims to improve on earlier methods. The "tree scan" method (Song and Hein, 2005), which is the focus of the first part of the study, can produce ancestries with several recombination events attaining its lower bound. This method works well, but it does not scale up. It is essentially an exhaustive search, making it practically impossible to deal with sample sizes of more than nine haplotypes. RecPars (Hein, 1993) attempts to overcome this limitation using heuristics. Some of its drawbacks include its assumption that marginal trees are rooted and binary. While this is a fairly common assumption, the author notes that such methods heavily depend on the unavoidable choice of an initial tree, which is generally arbitrary. It also limits the number of recombination events between two markers to one and is unable to infer more than one ancestry (Wu, 2011).

Central to RENT's algorithm is the phylogenetic concept of a split, which, in that context, is nothing more than a bipartition of a set of haplotypes. If each haplotype is associated with the leaf of a tree, a split is simply an edge; removing any edge of a tree results in a graph with two components. A split with a singleton component is deemed trivial and is non-informative because it appears in any tree. In general, the smaller the set of trees for which a given split appears, the more informative the split. For instance, a star graph is the least informative topology since every split is trivial. By contrast, a completely unbalanced binary tree is maximally informative since every partition is possible. RENT starts with a set of "trivial" marginal trees, one for each marker. A trivial tree has a single non-trivial split partitioning haplotypes based on their status with respect to a marker. Those trees are iteratively refined towards topologies with more and more non-trivial splits. This is done through heuristic rules that take into account not only the tree itself, but its neighbors as well.

A few years later, the author published an extension called RENT+ (Mirzaei and Wu, 2016), which builds on the same split refinement ideas. The main improvement concerns the initial set of marginal trees, which is constructed based on the distance between sampled haplotypes. The metric used is nothing more than a windowed and normalized version of Hamming's distance. In the original program, singleton markers—markers for which only one haplotype bearing the derived allele exists in a sample—were simply ignored; any information they may contain about inter-sequence similarities is not considered.

RENT is written in C++. We were unable to locate it online. RENT+ is written in Java and is available on GitHub.

3.6.2 ARBORES

M	C (B)	R (2)	S	L	I
✔	✖	✖	✖	C	CLI

The main idea of ARBORES is to scale up the tree scan procedure by applying it to overlapping sub-segments of haplotypes. We classify it as model-based regardless, as the heuristic serves as the foundation for a Metropolis–Hastings MCMC sampler designed to generate draws from the SMC distribution. Densities are derived from a hidden Markov model (HMM) with marginal trees and haplotypes as hidden and observed states, respectively (Heine et al., 2018).

Proposals are obtained by inferring ancestries for each sub-segment conditional on the marginal trees of its leftmost and rightmost markers, a procedure known as "bridging." Importantly, sub-segments can be processed concurrently. It begins with the computation of candidate topologies using a tree scan-inspired procedure (Song and Hein, 2005), which involves the iterative modification of a marginal tree by sub-tree-prune-and-regraft (SPR) operations. As the name indicates, an SPR operation consists of cutting a branch of the ancestry and reattaching it elsewhere. From a biological standpoint, it is equivalent to a pair of consecutive recombination and recoalescence events. As in Hein (1993), a parsimony constraint limiting the number of SPR operations between adjacent sites to at most one is enforced. The bridge is completed by drawing latitudes for free vertices, i.e., those absent from either terminal tree.

In theory, the search involved in the bridging procedure should be exhaustive. As it turns out, the computational burden is prohibitive, and actual implementation relies on a two-pass approximation. First, a bridge with the number of SPR operations between non-segregating and segregating sites restricted to a maximum of 0 and 2, respectively, is selected at random. Trajectories for which fulfillment of this constraint is not possible are simply discarded. To introduce variability, the set of candidates is augmented by subjecting topologies obtained in the first pass to additional SPR operations. Each is processed from left to right by performing an SPR operation at sites for which none were necessary in the first pass.

In addition to its inference algorithm, ARBORES also implements a tailor-made branch-and-bound consistency-checking procedure. It is provably exact, assuming no more than one SPR operation between segregating sites. Unfortunately, this proof does not hold for greater numbers of operations.

ARBORES's approach is certainly noteworthy, particularly due to its divide-and-conquer approach to model-based ARG inference, which enables concurrency while supporting a rich dependence structure. With that being said, the authors' own numerical experiments indicate that the current implementation of the algorithm does not scale.

ARBORES accepts input in a custom format documented in a PDF available in the GitHub repository from which it is distributed. The documentation contains concise information about other aspects of the software.

3.7 ARGweaver family

This family is composed of programs implementing threading-based algorithms.

3.7.1 ARGweaver

M	C (B)	R (2)	S	L	I
✔	✖	✖	✖	C++	CLI

ARGweaver's approach to ARG inference is radically different from that of its predecessors. The objective of inferring ancestries for a set of haplotypes is reframed as solving a sequence of *threading problems*. In essence, it involves constructing an ARG for n haplotypes conditional on an ARG inferred for $n-1$ other haplotypes under Markovian assumptions regarding the recombination process. Due to the notoriously large size of the space of possible ARGs, this is easier said than done. ARGweaver addresses this difficulty by implementing a tailor-made algorithm revolving around a space–time discretized SMC-type model called the *discretized sequentially Markov coalescent* (DSMC). Dependency between marginal trees is modeled as in the original SMC, but the locations of coalescence and recombination events, along with the positions of the latter, are constrained to a discrete grid. The DSMC can be viewed as an approximation to the SMC and is designed in such a way as to approach it (under some mild assumptions) as the grid is refined. The main drawback of this model is that it allows for zero-length branches with all related rounding concerns. These are addressed heuristically by ARGweaver. In any case, these assumptions allow for a tractable expression of the joint density of the ARG and the haplotypes as the complete data likelihood function of a HMM (Rasmussen et al., 2014).

A threading problem is solved by adding recombination and recoalescence events to the current graph. Concretely, this amounts to adding one branch between the sequence to be threaded and each marginal tree. The sequence of locations for these new branches on the current graph is called the *coalescence threading* of the new sequence. A *recombination threading* describing related variations between marginal trees is defined similarly. One important optimization is a consequence of the heavy restriction imposed on the recombination threading by the coalescence threading: it is considerably easier to sample the former conditional on the latter than to sample them jointly. This is the approach implemented in `ARGweaver`. Moreover, this procedure can be carried out in parallel because recombination events within this framework are conditionally independent.

Entire ARGs can be inferred through threading. After a first trivial graph containing a single sequence is constructed, the remaining sequences are threaded one at a time until completion. This is perhaps the most intuitive use case for `ARGweaver` and can be done efficiently. The main drawback of this approach is that the graph is built when threading sequence $k+1$ is sampled conditional on the first k sequences rather than on the whole sample, making the distribution of the resulting ARG invalid. The authors' recommended solution to this problem is to use sequence threading as the starting point for a Markov chain Monte Carlo (MCMC) sampler. The basic approach is to build an initial ARG using the method described previously and subsequently remove and rethread sequences. Again, this can be carried out efficiently. However, the impact of rethreading a single sequence on the graph is generally fairly local, leading to poor mixing. To improve the sampler's quality, `ARGweaver` implements sub-tree sampling. Sampling a whole sub-tree is not an easy task, mainly because sub-trees are not constant along sequences. Consequently, `ARGweaver`'s sub-tree sampler relies on a fairly involved sub-tree selection algorithm, which takes a toll on the performance.

In short, reframing ARG inference as a sequence of threading problems has the potential for significant improvement of performance over traditional methods, since such problems can be solved efficiently. Nevertheless, obtaining a satisfactory sample, i.e., one whose distribution is "close enough" to that of the CWR, from repeated rethreading is not straightforward.

`ARGweaver` is distributed with a handful of utilities in the form of individual programs. The sampler itself is a standalone piece of software called `ARG-sample`, which takes optionally compressed `fasta`-formatted files as input. A more compact custom file format called `sites` can also be used. Simulations will be started from scratch by default, but it is possible to specify an ARG as an initial value or even resume execution from a partially built graph. Simulation results are stored as Newick-encoded marginal trees along with data about recombination events. TMRCA and the recombination rate can be estimated from generated ancestries using `arg-extract-tmrca` and `arg-extract-recomb`. Facilities for ARG visualization are also provided. Conveniently, `ARGweaver`'s distribution also includes `arg-sim`, which can be used to generate samples of sequences under both SMC and classical CWR models along with DSMC. Implementations are custom.

`ARGweaver` has been subsequently modified to include demographic models in an extension known as `ARGweaver-D` (Hubisz et al., 2020).

3.7.2 `ARG-Needle`

M	C (B)	R (2)	S	L	I
✖	✖	✖	✖	C++	CLI*

`ARG-Needle` addresses `ARGweaver`'s limitations with a novel model for the TMRCA distribution, along with a marker-based distance heuristic. The algorithm is divided into three steps, the third of which involves threading a new sequence into the ARG. These steps are repeated until the complete ARG is built, i.e., until every sequence in the sample has been threaded. The heuristic is applied in the first step, identifying the most similar haplotypes among those already threaded. To account for recombination, the search is localized to subdivisions of sequences, the length of which is a user-defined parameter. This leads to a list of the most similar haplotypes for each subdivision. A hashing scheme is used to speed up this process. In the second step, the TMRCA between the sequence to be threaded s and each sequence computed in the previous step is estimated. Again, this estimation is performed per subdivision, using the ascertained sequentially Markovian coalescent algorithm (Palamara et al., 2018), which revolves around an HMM model of the haplotypes where the hidden state is made up of TMRCA intervals. Coalescent threading for s is computed in the third step using the minimum estimates from the previous step. Unlike `ARGweaver`, recombination threadings are not computed. In addition, `ARG-Needle`'s implementation of the threading procedure is "ARG-wide" (e.g., it does not consider each marginal tree individually), which increases its efficiency. A post-processing step called *ARG normalization*, which amounts to quantile renormalization on the heights of internal nodes based on trees sampled from the demographic prior, may allow for a reduction of bias in the TMRCA estimates (Zhang et al., 2023).

Despite both pieces of software being threading-oriented, `ARG-Needle` and `ARGweaver`'s approaches to ARG inference are radically different. The former prioritizes performance, heavily relying on heuristics and largely leaving `ARGweaver`'s distribution validity concerns unaddressed. That being said, Bisschop et al. (2025) indicated that satisfactory likelihoods can be formulated for ARGs that do not include recombination nodes under the SMC. However, the fundamental issue of the threading approach remains: the k^{th} threading is independent of the remaining $n-k$ unthreaded haplotypes. The existence of a likelihood function is unrelated to the statistical properties of the inferred ancestry.

In addition to the CLI interface, a `Python` API to the underlying library (called `ARG-Needle-lib`) is also available and distributed via PyPI.

3.7.3 `Threads`

M	C (B)	R (2)	S	L	I
✔	✖	✖	✖	C++	CLI

`Threads` is closely related to `ARG-Needle`. Both were designed in Pier Palamara's lab within a short time interval, and the articles presenting each software application have many authors in common. Both programs are designed to solve threading problems using heuristics and are very fast at doing so, enabling genome-scale ARG inference. However, a major

difference between the programs lies in Threads's use of the Li–Stephens model (LSM) (Li and Stephens, 2003). The popular framework, which is mentioned by ARGweaver's authors, is related to threading as it revolves around a factorization of the haplotypes' joint density of the form Gunnarsson et al. (2024).

$$f(h_1,\ldots,h_n \mid \rho) = \prod_{k=1}^{n} f(h_k \mid h_1,\ldots,h_{k-1},\rho)$$

Here, the h_is are haplotypes and ρ is a recombination parameter. Threads extends this expression slightly to include a mutation parameter, which we denote by μ in this section. In any case, the fundamental idea is to view the k^{th} haplotype as a *mosaic* of the first $k-1$ sequences that constitute a *reference panel*. Multiple problems, such as finding recombination breakpoints, can be solved using standard approaches for hidden Markov models within that framework.

The LSM model itself is sufficient to compute a coalescent threading, which is a fact noted by the authors. However, the large search space of possible threadings is disadvantageous from an efficiency standpoint. For this reason, the set of candidates for new coalescence events is first reduced in a pre-processing step via a tailor-made algorithm based on the positional Burrows–Wheeler transform (Durbin, 2014). Thanks to this heuristic, only candidates similar to the new haplotype, with respect to the Hamming distance, are considered. The implementation is efficient as it revolves around operations on red–black trees. Indeed, numerical experiments presented by the authors show that Threads is generally faster than ARG-Needle, often by an order of magnitude.

A new haplotype h_{n+1} is threaded into the graph according to the sequence of events maximizing $f(h_{n+1} \mid h_1,\ldots,h_n,\rho,\mu)$. h_{n+1} is associated with a word in $\{1,\ldots,n\}^m$, representing the pairwise similarity between two haplotypes for each marker. The number k at position p indicates that the new haplotype is most similar to sequence k at the p^{th} marker, given a pair of recombination and mutation rates. Of course, runs of similar letters are expected; these are called *Viterbi segments* and are conceptually related to the widespread notion of *IBD segment*, although they are not interchangeable. Each Viterbi segment making up a haplotype constitutes a threading unit, which acts as a proxy for recombination events (see Section 5 of the supplementary notes to Gunnarsson et al. (2024)). In any case, maximizing the conditional density of a new haplotype is computationally challenging, as the classical algorithm involves $\mathcal{O}(mn^2)$ evaluations. Threads significantly reduces this number. First, the pre-processing step described above is used to reduce the number of candidates to l, which is chosen to be small, possibly orders of magnitude smaller than m. While this is helpful, the heart of Threads's algorithm is a branch and bound approach based on certain characteristics of the recombination process, lowering the complexity of the so-called Threads–Viterbi step to linear in n. Since there are m haplotypes to be threaded, the complexity of this step is $\mathcal{O}(mln)$. A final source of speedup is that each of the m optimization problems is independent of the others, making computation perfectly parallelizable.

By default, Threads ancestries cannot be considered to be inferred from a sample in the usual sense. An optional post-processing step, which estimates the age of each mutation and alters the ARG topology to render it sample-consistent, is available to the user via a simple command-line switch. Optionally, mutation ages can be specified by the user.

Threads takes three files as input. The first two files should provide genotype data and genetic maps in the form of Plink2 and SHAPEIT formatted data, respectively. The third file should supply demographic information in a custom two-column format. The first column denotes the number of generations in the past, while the second column indicates the corresponding effective population size. The output is in a custom format (.threads file) but can be easily converted to the ARG-Needle format using a built-in utility. Notably, these files can later be used for genotype imputation by Threads itself. This is an important use case and another distinctive feature with respect to ARG-Needle.

Threads is coded in C++ and Python. It is made available through the PyPI index. The ARG construction process is controlled via a well-documented command-line interface. Despite a significant portion of the software being written in Python, there is currently no publicly documented Python API.

3.7.4 SINGER

M	C (B)	R (2)	S	L	I
✔	✖	✖	✖	C++	CLI

Similar to ARG-Needle, the main goal of SINGER is to improve upon ARGweaver by reducing the inference time, although it does not, for the most part, resort to heuristics for that purpose. As stated above, the main performance bottleneck in ARGweaver comes from its MCMC sampling step, which, while not mandatory, is necessary for full posterior sampling. It is, therefore, not surprising that one of the main innovations of SINGER is a simplification of the MCMC sampling step. The major source of complexity of ARGweaver's sub-tree sampling algorithm is the requirement for cuts to spatially span the totality of the ARG's genome positions. While mixing is improved, the procedure's sophistication is high, and its computational burden is heavy. SINGER's sub-graph pruning and regrafting operation represents a tradeoff, allowing efficient cuts on internal branches at the cost of reduced spatial span. At each step, the position of the next cut is sampled to the right of the interval covered by the last one. Its location on the associated marginal tree is sampled uniformly. This position is then stretched into an interval in both directions. Stretching in a given direction stops when either the end of the sequences is reached or a position deemed equivalent to the cut is encountered. Pruned sub-graphs are regrafted via threading by approximate sampling from the posterior distribution, which leads to high acceptance rates. This means that in addition to SINGER's sampling procedure being more efficient than ARGweaver's sub-tree sampling, a decrease in mixing times is expected. This is confirmed by simulation studies presented in Deng et al. (2025), which report a total speedup of approximately 400x, which is largely due to improved mixing (Deng et al., 2025).

Another source of improvement with respect to ARGweaver is its approach to the threading problem. Even though both software applications rely on hidden Markov models, SINGER once again offers a simpler solution by considerably reducing the number of hidden states. Instead of considering the complete set of discretized recombination and coalescence points, a two-step procedure is

implemented. These steps are known as *branch sampling* and *time sampling*. In the first step, a coalescence branch for the haplotype to be threaded is sampled. This is carried out efficiently via a deterministic approximation of coalescence times, which is used to compute two quantities. The first is a location on the coalescence branch called the *representative joining time*. It serves as a provisional coalescence location used to compute mutation probabilities. The second is the probability of coalescing with the branch under the coalescent process. It appears in expressions for recombination probabilities along with the representative joining time. In HMM parlance, recombination and mutation events correspond to transitions and emissions, respectively. However, this is not the whole story. SINGER's approach to the threading problem is unique in that single markers do not constitute the unit of analysis [referred to as *locus* in Deng et al. (2025)]. For probability computation purposes, they are grouped together in equally sized bins, effectively reducing the number of loci and, consequently, the computational burden. Additionally, the number of recombination events between two loci is restricted to be at most one, leading to a second transition type: instead of generating a new recombination event, the new haplotype can join a previous coalescence threading, a process called *recombination hitchhiking*.

SINGER's restriction on the number of recombination events is very strong. Indeed, in combination with its adherence to the SMC paradigm, it is too strong in the following sense: the set of possible threadings is not absolutely continuous with respect to that of possible ARGs. This is problematic because threadings not associated with a valid ARG could possibly be sampled. To remedy this issue, the authors introduce a distinction between *full* and *partial* branches. A branch is said to be partial if it has been bisected by a recombination or coalescence event; branches that are not partial are full branches. Strictly speaking, it is possible to generate a sequence of marginal trees under the SMC solely by keeping track of full branches. In fact, the previous statement is something of a threading-centric formulation of the SMC. Indeed, recombination and coalescence operations are often described in a somewhat metaphorical language. For instance, representing a coalescence event in an ARG is often described as bisecting the coalescence edge. This is generally helpful, but sometimes it obscures important details. From a graph theoretical standpoint, the actual operations involved are the addition and deletion of elements in the ARG's vertices and edges sets according to certain rules. It is possible to formulate the set of rules associated with SMC as a subset of that of standard CWR; these restrictions are associated with not adding new vertices or partial branches. In any case, in order to allow recombination hitchhiking, SINGER has to depart from strict SMC and keep track of partial branches. Theoretical complexity improvements granted by approximating the CWR are lost, but it can be mostly recovered in practice. By constant pruning of low-probability branches, the state space size can be kept close to its theoretical minimum, according to the authors.

Threading computation is followed by a rescaling step. Time is discretized into intervals. For each interval, a scaling factor based on the expected number of mutations is computed. The procedure is similar to ARG-Needle's normalization step; the key difference being that SINGER does not assume any demographic model.

SINGER is controlled via a CLI interface. It accepts input in the form of standard VCF files. Its output can easily be converted into the tskit format using an included Python script. Two scripts are also provided for output analysis. The first computes summary statistics indicative of the MCMC sampler's convergence. One of these, the number of non-uniquely-mapped sites, is reported for simulations in Deng et al. (2025). The other computes the pairwise coalescence times with respect to a focal haplotype. In addition, two scripts are provided for streamlining concurrent execution of SINGER on regions of input data. The first divides haplotypes into regions of a predetermined length; graphs are then constructed in parallel for each of these. The other allows the user to select a subset of regions for ancestry inference.

3.8 Others

3.8.1 SNPsim

M	C (B)	R (2)	S	L	I
✔	✔	✖	✔	C	CLI

SNPsim's primary feature is the ability to simulate recombination hotspots. The user must provide a base recombination rate, the expected number of hotspots, and a "hotspot" recombination rate. The program begins by selecting random locations for hot and cold spots, along with a specific rate for each. SNPsim then performs standard CWR simulation while accounting for the variable recombination rate (Posada and Wiuf, 2003).

Demographic scenarios with varying population sizes are supported. Complex mutation models are possible. Recurrent mutations can be simulated using a bi-allelic Jukes–Cantor model (Jukes and Cantor, 1969). Mutation models and rates can be mixed along haplotypes.

Although the article does not mention the types of events supported, the code in the function CalcIndividualG indicates support for type B recoalescence. Type 2 recombination events are out of the scope.

SNPsim is written in C and is publicly available through Google's Code Archive.

3.8.2 FastCoal

M	C (B)	R (2)	S	L	I
✔	✖	✖	✔	C++	?

This is the Marjoram and Wall C++ implementation of the SMC and SMC' algorithms. As a reminder, from an algorithmic perspective, SMC's main idea is to delete the recombination branch above the recombination location *before* generating a recoalescence event. SMC' slightly increases its support by deleting the branch *after* the recoalescence event, thus allowing recombination events to be effectively canceled (Marjoram and Wall, 2006).

Ancestries produced by FastCoal via SMC' are reportedly similar to those constructed via the original SMC algorithm, but with a distribution more similar to that of ms, used by the author as the reference CWR simulator. This conclusion is based on comparisons of the TMRCAs distributions and the patterns of linkage equilibrium between the two software applications. As expected, FastCoal is overwhelmingly faster than ms and more memory efficient.

We were unable to find either the FastCoal source code or binaries on the internet.

3.8.3 GENOME

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C++	CLI

Although it proceeds backward in time, GENOME is not, strictly speaking, a CWR simulator since it does not resort to continuous-time approximations. This design decision is motivated by diminishing returns as the length of the haplotypes to be simulated increases. Indeed, simulation studies reported in the article indicate runtimes inferior to those of ms and cosi for long chromosomes. The implemented scheme is to simulate every generation, working backward in time from the present. In addition to being more time-efficient, discrete generations allow for fine control over demographic events and simultaneous events such as coalescence between more than two sequences. ISM is assumed throughout the article. Variable recombination rates are supported, enabling the simulation of hotspots. The algorithm is implemented using sparse matrices, which are reused at each generation for efficient memory management. All informative events are supported, as revealed by the code of the SampleParent function (Liang et al., 2007).

The format of the output files is similar to that of ms. Simulation studies indicate that the output is in line with the theory and similar to that of ms. GENOME is programmed in C++ and is available online, although not at the address provided in the article. The interested reader can find its source code along with a manual and precompiled binaries for major operating systems at https://csg.sph.umich.edu/liang/genome/download.html.

3.8.4 TMARG

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C++	?

TMARG is based on lemma 4.1 of Wu (2008), which is an extension of lemma 3.1 of Song et al. (2005). An ARG may or may not contain haplotypes absent from its sample; those are known as *Steiner sequences*. An ARG without Steiner sequences is called *self-derived*. A sample for which a self-derived ARG exists is called *self-derivable*. Lemma 4.1 states that if a sample is self-derivable, every minARG for that sample is self-derived (Wu, 2008).

TMARG actually implements two different algorithms, and both are based on parsimony-centric heuristics. The first algorithm presented in the article relies on the fact that sampling uniformly over the space of self-derived ARGs can be carried out efficiently. Unfortunately, this approach has two issues that greatly reduce its usability in real-world scenarios. First, it requires counting the total number of possible minARGs for a given sample. This can be achieved using a fairly simple procedure, albeit in a time exponential in the size of the sample. An algorithm is described in the article and implemented in TMARG. Once this is accomplished, however, constructing an ARG is an operation with quadratic complexity.

The second flaw of this approach is perhaps the more obvious: it is only applicable to self-derivable samples. If the sample of interest does not meet this requirement, one strategy is to augment it with Steiner sequences to make it self-derivable. This is not practical in general, as the search space for Steiner sequences grows exponentially with the number of markers. Heuristics aiming at facilitating exploration of this space are presented in the articles, but for larger datasets, TMARG implements an alternative algorithm called *pathway sampling*. The idea is to first draw a permutation of $1, \ldots, n$, where n is the number of sampled haplotypes. Then, a sequence of events generating haplotype i involving only haplotypes 1 up to $i-1$ is computed for each i, and the ARG is built accordingly. Pathway sampling is related to SHRUB's algorithm.

TMARG is coded in C++. Unfortunately, its source code is not publicly available online.

3.8.5 SLiM

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C/C++	DSL*

SLiM, which stands for "selection on linked mutations," is an agent-based forward-in-time simulator based on the Wright–Fisher model. Its stated goal is "uniting genetics and evolutionary biology with ecology" (Haller and Messer, 2019). The primary motivation for its creation was the observation that an efficiency/flexibility trade-off is unavoidable when designing a population genetics simulator; programs relying on coalescent approximations often sacrifice the flexibility needed to model complex biological and population phenomena. Nevertheless, SLiM manages to bridge some of the performance gap while providing an impressive array of features. Indeed, its latest major release can even be used to simulate a complete human genome (Haller et al., 2025; Messer, 2013).

In addition to chromosomal crossover and gene conversion, selection, complex scenarios of demography and population substructure, selective sweeps, epistasis, continuous spatial models, multiple species, and multiple chromosomes are supported. A distinct characteristic of SLiM is that many of these features are not hard-coded but instead left for users to implement. This is accomplished through a domain-specific language (DSL) called Eidos, which was specifically designed for this purpose. Eidos is admittedly "very simple and unoriginal," with its syntax and semantics heavily inspired by C and R. Consequently, it is relatively easy to learn, even for users with little to no programming experience. One of the uses of Eidos is to specify the initial conditions for a simulation run, and in this respect, it is functionally equivalent to a well-designed, text file-based parameter specification system. What sets it apart is the possibility to finely tune the simulation workflow via a callback mechanism, allowing user-defined functions to be called at specific stages. This enables users to implement functionalities that were not previously envisioned by SLiM's authors without having to modify and recompile the program's source code.

SLiM is compatible with non-WF forward-in-time agent-based models, which are characterized by an emphasis on individual behavior and, as a result, increased biological realism. Much of the support for this class of models comes from explicitly modeling individual ages and from an enriched API and additional callbacks. The former allows for overlapping generations and varying population sizes by modeling fitness as survival

probabilities. The latter gives users fine control over migration and reproduction events.

The price for SLiM's flexibility is having to learn a new programming language, albeit a relatively easy one. However, this is partly offset by a comprehensive collection of "recipes" for common scenarios available in the manual. Additionally, a graphical interface called SLiMgui is provided to assist in the development of Eidos scripts. Many of the functionalities of an integrated development environment are available, including code completion and highlighting, along with static syntax checking. Simulations can be observed in real time, with the possibility of live graphical representations.

Another option is to use SLiM through R, facilitated by the slimr R package (Dinnage et al., 2023), which is available on GitHub. Note that this package is different from SlimR available on CRAN.

Much of the performance of SLiM comes from using optimized data structures, including treeSequences, whose implementation is discussed in Haller et al. (2019). SLiM is strongly integrated with the tskit ecosystem, which can be used for post-simulation analysis. Moreover, a neutral "burn-in" coalescent simulation from msprime can be prepended to a genealogy, an operation that is dubbed "recapitation." Users, therefore, benefit from the best of both worlds: the realism of forward-in-time simulation and the performance of a coalescent approximation. Recapitation, among other operations, can be performed using a small Python package called pyslim that is available on PyPI.

SLiM's documentation and the manual for the Eidos language are extensive. Both are available on the project's website. Due to its ability to output treeSequences, the simulation results are readily converted to any format supported by tskit. The source code is available on the project's GitHub repository, along with a macOS installer and instructions for installation on Linux and Windows.

3.8.6 scrm

M	C (B)	R (2)	S	L	I
✔	✔	✔	✔	C++	CLI

scrm, the software, is an implementation of SCRM, an approximation of the standard coalescent with recombination model. It is presented as an improvement on SMC and SMC', both of which may be unsuitable under certain conditions (Eriksson et al., 2009). From a performance standpoint, reported simulation studies indicate that scrm is more efficient than either MaCS or FastSimCoal and approximately 50 times faster than ms. Part of this speed-up is related to the data structure. Indeed, scrm avoids the complexities associated with graphs by storing information in an efficient tree-based data structure. From an algorithmic standpoint, it shares multiple similarities with simulators of the MaCS family in general and SC in particular. Type 2 recoalescence events are generated by allowing recoalescence with non-ancestral branches and generating a new recombination event on the recoalescence branch with a certain probability. Both programs can approximate the full CWR distribution by disregarding branches separated from the current marginal tree by a distance above a user-defined threshold. In that regard, scrm's justification for disregarding linkage past a certain distance is well-detailed. It is framed in terms of *accumulated recombination rate*. In simpler words, the rate at which recombination events occur on non-ancestral branches increases with its distance from the current marginal tree. A consequence of this observation is that recoalescence events with distant branches might as well be ignored, as they are unlikely to affect the history of the recoalescing tree. However, in practice, ignoring branches with a high recombination rate is equivalent to constraining events to occur on branches within a sliding window of ancestral material. Interestingly, Kelleher et al. (2016) reports that there does not appear to be a window size above which approximation of the CWR worsens, which is the case for MaCS (Staab et al., 2015).

3.8.7 Relate

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C++	CLI

Relate implements an ARG inference algorithm whose primary goal is to enable analysis of large samples on a genome scale. It can manage phased and polarized sequences of bi-allelic SNPs on the order of 10^4 and allows for some level of error in the input data. Another feature of interest is the possibility of constraining ancestries by specifying ancient sequences (Speidel et al., 2021; Speidel et al., 2019).

Relate's algorithm generates a sequence of trees that is consistent with the input sample in certain limiting scenarios. Fundamentally, marginal tree reconstruction is dealt with as a hierarchical clustering problem. Clustering is performed according to a distance related to the difference in the number of mutations on each sequence. The distance in question is not a metric since it is not symmetric; this is intentional to take advantage of the polarized nature of the data. In fact, it is constructed from the concept of *derived mutation*, which is the number of derived alleles carried by one of the haplotypes but not by the other. The number of derived mutations between each pair of haplotypes is computed from the posterior probabilities of a Li–Stephens model modified to account for polarization. The result is forwarded to the tree builder, which is the procedure tasked with performing the clustering. Ties are possible and are dealt with heuristically. Importantly, the branch length is not estimated at this stage.

Tree building is not performed at each SNP but rather when needed. The computationally demanding procedure is only performed when a marker cannot be *mapped* to the current marginal tree. The mapping rules follow the ISM model approximately. Ideally, it should be possible to locate a single mutation event on the marginal tree in a way that separates derived sequences from those carrying the wild allele. In practice, some flexibility is allowed to account for the various errors that may affect the data.

Branch length estimation is performed via MCMC once topology reconstruction is complete. A coalescent prior is used, and equivalent branches in different trees are reconciled before the beginning of the sampling procedure. As noted by the authors, this part of the algorithm is computationally demanding but allows for great flexibility.

Functionalities of Relate can be extended via add-ons. These add-ons provide capabilities such as effective population size, mutation rates, branch lengths estimation/reestimation, selection detection, tree extraction according to user-specified criteria, and

plotting. These add-ons are included in the standard `Relate` distribution as standalone scripts.

Conveniently, shared memory parallelization can be achieved via the included `RelateParallel.sh` script, which takes the same arguments as the main program, with the addition of the `--threads n` switch to specify the number of threads to use. Distributed memory parallelism is also possible, but it requires a more involved setup. A script to run simulations on a "sun grid engine" managed cluster is provided, which, while being written for a deprecated system, should be compatible with the open cluster scheduler.

Input haplotypes are to be provided in the form of haps/legend/sample files, which is the format used by SHAPEIT (Delaneau et al., 2011). When necessary, recombination maps can be provided via a custom format detailed in the online documentation.

3.8.8 `tsinfer`

M	C (B)	R (2)	S	L	I
✖	✔	✔	✖	C	Python

`tsinfer` handles the problem of ancestry inference backward. It starts by inferring ancestral haplotypes and then works its way back to the sample. Its algorithm is divided into three steps, the first being to determine the age of the derived allele for each marker. This is done using a simple heuristic relying on derived allele frequencies. The term "age" is used loosely as only the relative temporal ordering is estimated (Kelleher et al., 2019).

The ancestral haplotypes are inferred in the second step. Estimation is informed by the results of the preceding stage. Again, the process is entirely heuristic and very efficient, relying on a consensus algorithm rather than a genetic model. For each focal site, a set of descendants is constructed assuming the ISM. This set contains every haplotype with the derived allele. The state of nearby sites is determined based on consensus between haplotypes. Disagreement with the consensus is interpreted as indicative of recombination. Divergent haplotypes are removed from the set of descendants as inference propagates left and right. Inference stops once the cardinality of the sets decreases below a certain threshold. This process does not generally span the full length of the haplotypes; the remaining markers are deemed non-ancestral.

The final step of the algorithm is to associate each haplotype with a set of ancestors in order to produce an ARG in the form of a sequence of marginal trees. Ancestors are computed along haplotypes by the maximum likelihood under the LSM. One peculiarity of `tsinfer` compared to other programs using the LSM, such as `Threads` or `Relate`, is that the reference panel is made up of ancestral rather than sampled haplotypes. This is problematic when the number of markers m is large, since the size of the panel is on a similar scale, leading to a $\mathcal{O}(m^2)$ time complexity. Moreover, the reference panel is not static through this step. To circumvent these issues, the LSM is formulated in terms of tree sequences (see Section 3.1.5) in a way that makes computation and storage of the likelihood value unnecessary for certain ancestral haplotypes. To actually perform the inference, `tsinfer` implements a custom version of the Viterbi algorithm that is aware of the relationships between the elements of the reference panel. Due to the ancestry's topology, many sequences have the same likelihood as their parents. These haplotypes are simply marked with a reserved value instead of undergoing likelihood computation. Without surprise, the end result is output as a `treeSequence`.

Despite its reliance on numerous heuristics, reported simulation studies indicate good statistical performance in practice. `tsinfer` is distributed as a Python package and is available on PyPI.

3.8.9 SARGE

M	C (B)	R (2)	S	L	I
✖	✖	✖	✖	C++	CLI

SARGE is an ARG inference program that implements a heuristic that does not compromise when it comes to performance. It implements a greedy algorithm built upon the four gametes test. The author fully acknowledges the tendency of their program to systematically underestimate the number of recombination events and warns users against using it for tasks sensitive to such bias. Moreover, SARGE's main focus is topological. Its approach to branch length estimation is a heuristic involving the locations and positions of mutation events. Despite this, simulation studies presented in the article indicate that estimates are devoid of systematic biases. Additionally, recombination breakpoints are not simulated (Schaefer et al., 2021).

SARGE revolves around a simple procedure designed to determine a sequence of recombination events between two phylogenetic trees, which is made possible by a representation of binary trees as hierarchical clades of haplotypes. The main idea is to reorganize a subset of the clades of the first tree in a way that yields the second one. Since multiple choices are possible in general, the decision is guided by a parsimony heuristic: branch movements (i.e., SPR operation) explaining the maximum number of discordant clades are selected. Breakpoints are positioned halfway between the rightmost position associated with the left tree and the leftmost position associated with the right one. Discordance is defined as a failure of the four gamete test. As remarked earlier (Section 3.4.3), applying this test to every pair of markers is not feasible in practice. SARGE overcomes this difficulty by disregarding markers positioned outside of a moving *propagation window*, which is an approach similar to that of other algorithms such as `MaCS` or `scrm`.

SARGE uses a custom format for input files but is distributed with utilities for converting `VCF` filesand the output of a run of the `ms` simulator into it. Other notable utilities include `Python` scripts for Newick-formatted trees visualization, propagation window determination, and gap determination. The latter can be used to split haplotypes into subsequences that are suitable for parallel analysis, which can be performed by an included `bash` script relying on `GNU Parallel`.

A peculiarity of SARGE is that the sample size is a compile-time constant. This, combined with the fact that it is written in C++ and is hence ahead-of-time compiled, may lead to significant overhead, temporal or spatial, for the practitioner dealing with samples of variable sizes on a regular basis. The SARGE source code and a detailed readme are available on GitHub.

3.8.10 ARGinfer

M	C (B)	R (2)	S	L	I
✔	✔	✔	✖	Python	Python⋆

The impressive computational performance of software such as Relate and tsinfer allows them to handle large samples, but this comes at the price of various approximations. On the other hand, programs implementing an MCMC scheme such as ARGweaver and Arbores have good statistical properties but are limited to sample sizes a few orders of magnitude smaller than those of their recent heuristic counterparts. Importantly, these are based on the SMC model instead of the full CWR and cannot generate genealogies containing trapped non-ancestral material. Similar to these two programs, ARGinfer implements probabilistic ARG inference via an MCMC algorithm. However, it does so by targeting the CWR distribution instead of the SMC, thus hopefully improving accuracy in the process. From a computational performance perspective, ancestries are stored in a custom data structure called *augmented tree sequence* (ATS), which is a variation on tskit's tree sequence designed for ARG inference (Mahmoudi et al., 2022).

ARGinfer assumes a coalescent prior and uses a likelihood that takes branch lengths, genomic intervals, and number of mutations carried into account. The initial ARG is inferred using a heuristic algorithm inspired by ARG4WG (Section 3.4.2). The proposal distribution is constructed from six different types of moves, with the first four being relatively straightforward: an SPR operation, the addition/removal of a recombination, and the resampling of a recombination breakpoint. These are readily implemented but are chosen far less often in a typical run than the two other moves, which have superior mixing capabilities. These are the Kuhner move and the resampling of event times. A Kuhner move is similar to an SPR operation but follows the full CWR distribution. Specifically, the pruned edge can undergo a sequence of recombination and coalescence events before completely recoalescing with the ARG.

An ATS record is a six-tuple (s, p, c, b, m, t), where t and c are defined as in a coalescence record for a tree sequence (Section 3.1.5). b and m contains the eventual breakpoints and mutation events associated with the branch. The entries l and r of a coalescence record are replaced by s, a sequence of disjoint regions of the form $[l_1, r_1], \ldots, [l_k, r_k]$. p contains the parents of the edge. If the event at the bottom of the branch is a coalescence, c is a pair, just as it always is in the case of coalescence records. However, a major difference between these and ATS records is the latter's capacity to encode recombination events directly. When one such event is at the bottom of a branch, the associated c is a singleton. The reverse is true for p: it is a singleton when the top event is a coalescence and a pair when it is a recombination. Topologically, an ATS is a simple graph. Reportedly, storing mutation events along with edges helps speed up computations by allowing likelihood evaluation without site-by-site sequence comparisons. Despite this, numerical experiments on simulated data indicate that ARGinfer takes longer than ARGweaver to achieve convergence. This is especially true when the mutation rate is smaller than the recombination rate. According to the authors, this discrepancy is due in part to the poor mixing provided by ARGinfer's proposal distribution. On the flip side, parameter estimation performances appear approximately at least as good as that of ARGweaver's and is even superior for recombination rates, TMRCA, and allele age (i.e., mutation latitude).

In addition to a Python API, a CLI interface is available. Both interfaces are documented in the package documentation hosted on Read the Docs. Source code is available on GitHub, and the package is distributed through PyPI.

3.8.11 Espalier

M	C (B)	R (2)	S	L	I
✔	✖	✖	✖	Python	Python⋆

Admittedly, the primary target of the botanically-named Espalier is not human genomic data. However, we feel that the originality of its approach to inference, coupled with the fact that it can be applied to human data regardless of its creators' intent, justifies its inclusion in this review (Rasmussen and Guo, 2023).

On the surface, the most striking characteristic of Espalier is its ability to generate recurrent mutations. In this respect, it can be considered the model-based counterpart of Ignatieva et al. (2021). In addition, for reasons that will become clear in subsequent paragraphs, Espalier works best in scenarios where the mutation rate "significantly exceeds" the recombination rate. Overall, this makes it well-suited for ancestry inference of rapidly mutating organisms. Indeed, Rasmussen and Guo (2023) reported a simulation study performed on a sample of genomes of potyviruses, a plant RNA virus with a high mutation rate.

The algorithm revolves around a preexisting concept intimately related to SPR, known as a maximum agreement forest (MAF) (Hein et al., 1996). Removing edges from a tree creates a set of disconnected trees called a *forest*. An *agreement forest* for two trees is any forest obtainable in this way from both trees. Given two trees with n leaves, finding an agreement forest is trivial (by removing $n-1$ edges, for instance). However, computing a forest minimizing the number of cuts is, in general, provably NP-hard. Such a forest is a MAF. Consequently, Espalier relies on a custom approximate algorithm named the *4-cut MAF algorithm*, which is a modified version of the *3-cut MAF algorithm*.

Somewhat similarly to Heine et al. (2018), the sample of genetic material for which an ancestry is to be inferred needs to be divided into regions. Each of these regions should have a low probability of recombination events occurring within them and is determined by the user. Espalier's algorithm demands that a coalescent tree be associated with each region. These can either be user specified or constructed by the program. According to the authors, the accuracy of these initial trees is a relatively minor concern, as their algorithm leverages correlations between neighboring trees to improve them. In any case, this concludes the first stage of the algorithm. In the following stage, pairs of trees are iteratively reconciled via a branch-and-bounds type algorithm. Given two trees T_1 and T_2 and an (approximate) MAF $V = V_1, \ldots, V_m$ obtained from the 4-cut algorithm, a tree is reconstructed by iteratively regrafting V_k, $k = 2, \ldots, m$ to the trees generated in the previous steps of the procedure. At step k, 2^k trees are created by reattaching V_{k+1} to its parent edge in either T_1 or T_2. In practice, only trees for which a likelihood ratio is above a certain bound are considered, which contributes to making the algorithm tractable. Once the reconciliation procedure is completed, a sequence of trees is selected by maximizing the joint likelihood of marginal trees, as provided by the SMC distribution.

TABLE 1 Summary of ARGs simulation and inference software.

Name	Year	References	Model	Coal. (B)	Rec. (2)	Simulated	Language	Interface
ms family								
Ms	2002	Hudson (2002)	✓	✓	✓	✓	C	CLI
Msms	2010	Ewing and Hermisson (2010)	✓	✓	✓	✓	Java	CLI
cosi2	2014	Shlyakhter et al. (2014)	✓	✓	✓	✓	C++	Text
discoal	2016	Kern and Schrider (2016)	✓	✓	✓	✓	C	CLI
msprime	2021	Baumdicker et al. (2021)	✓	✓	✓	✓	C	Python*
SIMCOAL family								
SIMCOAL2	2004	Laval and Excoffier (2004)	✓	✓	✓	✓	C++	CLI
fastsimcoal	2011	Excoffier and Foll (2011)	✓	×	×	✓	C++	CLI
SHRUB family								
SHRUB	2005	Song et al. (2005)	×	✓	✓	×	?	?
beagle	2005	Lyngsø et al. (2005)	×	✓	✓	×	C	CLI
KwARG	2021	Ignatieva et al. (2021)	×	✓	✓	×	C	CLI
Margarita family								
Margarita	2006	Minichiello and Durbin (2006)	×	×	×	×	Java	CLI
ARG4WG	2017	Nguyen et al. (2017)	×	×	×	×	C++	CLI
GAMARG	2019	Thao and Vinh (2019)	×	×	×	×	C++	CLI
MaCS family								
MaCS	2008	Chen et al. (2008)	✓	✓	×	✓	C++	CLI
SC	2014	Wang et al. (2014)	✓	✓	✓	✓	C++	CLI
SC-sample	2014	Wang et al. (2014)	✓	✓	✓	×	C++	CLI
Tree scan family								
RENT	2011	Wu (2011)	×	✓	✓	×	C++/Java	CLI
ARBORES	2018	Heine et al. (2018)	✓	×	×	×	C	CLI
ARGweaver family								
ARGweaver	2014	Rasmussen et al. (2014)	✓	×	×	×	C++	CLI
ARG-Needle	2023	Zhang et al. (2023)	×	×	×	×	C++	CLI*
Threads	2024	Gunnarsson et al. (2024)	✓	×	×	×	C++	CLI
SINGER	2025	Deng et al. (2025)	✓	×	×	×	C++	CLI
Others								
SNPsim	2003	Posada and Wiuf (2003)	✓	✓	×	✓	C	CLI
FastCoal	2006	Marjoram and Wall (2006)	✓	×	×	✓	C++	?
GENOME	2007	Liang et al. (2007)	✓	✓	✓	✓	C++	CLI
TMARG	2008	Wu (2008)	×	✓	✓	×	C++	?
SLiM	2013	Messer (2013)	✓	✓	✓	✓	C/C++	DSL*
Scrm	2015	Staab et al. (2015)	✓	✓	✓	✓	C++	CLI
Relate	2019	Speidel et al. (2019)	×	✓	✓	×	C++	CLI
tsinfer	2019	Kelleher et al. (2019)	×	✓	✓	×	C	Python
SARGE	2021	Schaefer et al. (2021)	×	×	×	×	C++	CLI

(*Continued*)

TABLE 1 Continued

Name	Year	References	Model	Coal. (B)	Rec. (2)	Simulated	Language	Interface
ARGinfer	2022	Mahmoudi et al. (2022)	✓	✓	✓	×	Python	Python*
Espalier	2023	Rasmussen and Guo (2023)	✓	×	×	×	Python	Python*

Checkmarks (✔or ✔) and cross marks (✖or ✖) indicate the presence or absence of a property, respectively. A green or red checkmark indicates desirable or undesirable features, respectively. The opposite is true for crosses. Up-to-date links to source code and documentation are available at https://patrickfournier.ca/publications/arg-software-review/graph.

This task is framed as a HMM problem and carried out efficiently via the Viterbi algorithm. Ultimately, latitudes are assigned to vertices via a tailor-made algorithm, and recombination events are imputed with the help of previously computed MAFs. The final result is stored as a `tskit` object. As an aside, the authors provide a method for constructing multiple ancestries from the same sample.

The simulation experiments reported in the article cover a broad spectrum of scenarios and are discussed at length. The reported running time of `Espalier` is at least an order of magnitude faster than the convergence time of `ARGweaver`. Despite this, it remains slow in comparison to other ARG inference software. Its author suggests that the pure `Python` implementation of the 4-cut MAF algorithm may lead to improved performance.

`Espalier` is distributed as a `Python` package on PyPI and is documented on Read the Docs. Its source code is publicly available on GitHub. A CLI is available in addition to the `Python` interface.

4 Discussion

We have reviewed 32 ancestral recombination graph simulation and inference software. Our findings are summarized in Table 1. Among them, approximately two-thirds are designed to infer ancestries, while the others implement simulation algorithms. The most striking features of our dataset are as follows:

- Every heuristic-based program is an inference program. Equivalently, ARG simulators are model-based.
- Almost half of the algorithms described disregard either type B coalescence events or type 2 recombination events. Most of them are inference algorithms.
- `C` and `C++` are by far the most common programming languages.
- Most programs have a single interface, and that interface is a command line.

ARG inference programs' heavy dependence on heuristics reflects the challenges involved in reconstructing genealogies. Formulating a statistical model for sample-consistent ARGs is difficult. Designing scalable inference algorithms can prove even more challenging, and implementing them correctly is not a task to be underestimated. These observations help explain why, to a large extent, all is fair in ARG inference. Consistent ancestries are valuable, and many are prepared to sacrifice some statistical correctness to obtain them, more so than in other inference contexts. Similar observations can be made regarding the widespread exclusion of certain categories of events by this class of software. In this regard, the variable level of approximation offered by `SCsample` is an interesting feature, allowing users to finely tune the trade-off between performance and realism to suit their needs.

The ubiquitous use of `C` and `C++` for implementation is hardly surprising. Both are mature and well-established programming languages with a largely justified reputation for delivering robust performance. Hence, they are natural choices for tackling the computational challenges inherent to ARG simulation and inference. The main drawback of these languages is perhaps the awkwardness involved in integrating them into a modern workflow. There are no formal packaging standards for these languages, and no standardized way of sharing code, at least nothing with a scope comparable to that of `Python` for instance. Even though platforms such as GitHub help, distribution remains challenging. This is compounded by the fairly involved process of compiling `C/C++` source code. On that matter, the ahead-of-time compilation required by these languages makes designing interactive software a highly non-trivial task. As a matter of fact, the vast majority of the programs reviewed are limited to a command-line interface and plain-text input/output. This is not inherently bad: CLIs are ubiquitous, and plain text is a famously universal interface. That being said, universality is not a replacement for convenience. In many contexts, the ability to interact with a library via a well-designed API is invaluable in streamlining workflows. Arguably, a polished interface can consolidate adoption among practitioners. In that regard, `msprime` is one of a kind. It is by far the most widespread simulator. While its performance and flexibility are certainly a major contributing factor to its popularity, the choice of its designers to make it available as a rich `Python` library is undoubtedly loved by its numerous users.

Despite this success story, let us not forget that developing solutions that are both fast and user-friendly is difficult: `msprime` is a major undertaking. For one thing, while it is mostly written in `Python`, `C` accounts for almost 40% of the codebase. Interfacing both languages is not trivial, and coding in `C` is certainly harder than doing so in a high-level language such as `Python`. There is hope that the recent popularity of recent languages such as `Julia`, which promise to simplify scientific programming by, among other things, solving the two-language problem, may speed up development and increase the quality of future software.

Author contributions

PF: Writing – original draft, Writing – review and editing. FL: Writing – review and editing.

Funding

The author(s) declared that financial support was not received for this work and/or its publication.

Conflict of interest

The author(s) declared that this work was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declared that generative AI was not used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Anderson, C. N. K., Ramakrishnan, U., Chan, Y. L., and Hadly, E. A. (2004). Serial simcoal: a population genetics model for data from multiple populations and points in time. *Bioinformatics* 21, 1733–1734. doi: 10.1093/bioinformatics/bti154

Arenas, M. (2012). Simulation of molecular data under diverse evolutionary scenarios. *PLoS Comput. Biol.* 8, e1002495. doi: 10.1371/journal.pcbi.1002495

Baumdicker, F., Bisschop, G., Goldstein, D., Gower, G., Ragsdale, A. P., Tsambos, G., et al. (2021). Efficient ancestry and mutation simulation with msprime 1.0. *Genetics* 220, iyab229. doi: 10.1093/genetics/iyab229

Bisschop, G., Kelleher, J., and Ralph, P. (2025). Likelihoods for a general class of ARGs under the SMC. *Genetics*. 232 (1), iyaf103. doi: 10.1093/genetics/iyaf103

Brandt, D. Y. C., Huber, C. D., Chiang, C. W. K., and Ortega-Del Vecchyo, D. (2024). The promise of inferring the past using the ancestral recombination graph. *Genome Biol. Evol.* 16, evae005. doi: 10.1093/gbe/evae005

Browning, B. L., Zhou, Y., and Browning, S. R. (2018). A one-penny imputed genome from next-generation reference panels. *Am. J. Hum. Genet.* 103, 338–348. doi: 10.1016/j.ajhg.2018.07.015

Browning, B. L., Tian, X., Zhou, Y., and Browning, S. R. (2021). Fast two-stage phasing of large-scale sequence data. *Am. J. Hum. Genet.* 108, 1880–1890. doi: 10.1016/j.ajhg.2021.08.005

Cardin, N. (2007). *Approximating the Coalescent with Recombination*. phdthesis: University of Oxford.

Carvajal-Rodriguez, A. (2008). Simulation of genomes: a review. *Curr. Genomics* 9, 155–159. doi: 10.2174/138920208784340759

Chen, G. K., Marjoram, P., and Wall, J. D. (2008). Fast and flexible simulation of DNA sequence data. *Genome Res.* 19 (1), 136–142. doi: 10.1101/gr.083634.108

Chen, X., Ma, Z.-M., and Wang, Y. (2014). Markov jump processes in modeling coalescent with recombination. *Ann. Statistics* 42 (4), 1361–1393. doi: 10.1214/14-aos1227

Delaneau, O., Marchini, J., and Zagury, J.-F. (2011). A linear complexity phasing method for thousands of genomes. *Nat. Methods* 9, 179–181. doi: 10.1038/nmeth.1785

Deng, Y., Nielsen, R., and Song, Y. S. (2025). Robust and accurate bayesian inference of genome-wide genealogies for hundreds of genomes. *Nat. Genet.* 57, 2124–2135. doi: 10.1038/s41588-025-02317-9

Dinnage, R., Sarre, S. D., Duncan, R. P., Dickman, C. R., Edwards, S. V., Greenville, A. C., et al. (2023). slimr: An R package for tailor-made integrations of data in population genomic simulations over space and time. *Mol. Ecol. Resour.* 24 (3), e13916. doi: 10.1111/1755-0998.13916

Durbin, R. (2014). Efficient haplotype matching and storage using the positional burrows-wheeler transform (pbwt). *Bioinformatics* 30, 1266–1272. doi: 10.1093/bioinformatics/btu014

Eriksson, A., Mahjani, B., and Mehlig, B. (2009). Sequential markov coalescent algorithms for population models with demographic structure. *Theor. Popul. Biol.* 76, 84–91. doi: 10.1016/j.tpb.2009.05.002

Ewing, G., and Hermisson, J. (2010). MSMS: a coalescent simulation program including recombination, demographic structure and selection at a single locus. *Bioinformatics* 26, 2064–2065. doi: 10.1093/bioinformatics/btq322

Excoffier, L., and Foll, M. (2011). Fastsimcoal: a continuous-time coalescent simulator of genomic diversity under arbitrarily complex evolutionary scenarios. *Bioinformatics* 27, 1332–1334. doi: 10.1093/bioinformatics/btr124

Excoffier, L., and Lischer, H. E. L. (2010). Arlequin suite ver 3.5: a new series of programs to perform population genetics analyses under Linux and Windows. *Mol. Ecol. Resour.* 10, 564–567. doi: 10.1111/j.1755-0998.2010.02847.x

Excoffier, L., Novembre, J., and Schneider, S. (2000). Computer note. simcoal: a general coalescent program for the simulation of molecular data in interconnected populations with arbitrary demography. *J. Hered.* 91, 506–509. doi: 10.1093/jhered/91.6.506

Excoffier, L., Marchi, N., Marques, D. A., Matthey-Doret, R., Gouy, A., and Sousa, V. C. (2021). Fastsimcoal2: demographic inference under complex evolutionary scenarios. *Bioinformatics* 37, 4882–4885. doi: 10.1093/bioinformatics/btab468

Gunnarsson, A. F., Zhu, J., Zhang, B. C., Tsangalidou, Z., Allmont, A., and Palamara, P. F. (2024). A scalable approach for genome-wide inference of ancestral recombination graphs. *bioRxiv*. doi: 10.1101/2024.08.31.610248

Haller, B. C., and Messer, P. W. (2019). Slim 3: forward genetic simulations beyond the wright–fisher model. *Mol. Biol. Evol.* 36, 632–637. doi: 10.1093/molbev/msy228

Haller, B. C., Galloway, J., Kelleher, J., Messer, P. W., and Ralph, P. L. (2019). Tree-sequence recording in SLiM opens new horizons for forward-time simulation of whole genomes. *Mol. Ecol. Resour.* 19, 552–566. doi: 10.1111/1755-0998.12968

Haller, B. C., Nelson, C. W., Rodrigues, M. F., and Messer, P. W. (2025). SimHumanity: using SLiM 5.0 to run whole-genome simulations of human evolution. *Hum. Popul. Genet. Genomics* 5 (4), 0006. doi: 10.47248/hpgg2505040006

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.* 4, 100–107. doi: 10.1109/tssc.1968.300136

Hein, J. (1993). A heuristic method to reconstruct the history of sequences subject to recombination. *J. Mol. Evol.* 36, 396–405. doi: 10.1007/bf00182187

Hein, J., Jiang, T., Wang, L., and Zhang, K. (1996). On the complexity of comparing evolutionary trees. *Discrete Appl. Math.* 71, 153–169. doi: 10.1016/s0166-218x(96)00062-5

Heine, K., Beskos, A., Jasra, A., Balding, D., and De Iorio, M. (2018). Bridging trees for posterior inference on ancestral recombination graphs. *Proc. R. Soc. A Math. Phys. Eng. Sci.* 474, 20180568. doi: 10.1098/rspa.2018.0568

Hellenthal, G., and Stephens, M. (2006). msHOT: modifying Hudson's ms simulator to incorporate crossover and gene conversion hotspots. *Bioinformatics* 23, 520–521. doi: 10.1093/bioinformatics/btl622

Hoban, S., Bertorelle, G., and Gaggiotti, O. E. (2012). Computer simulations: tools for population and evolutionary genetics. *Nat. Rev. Genet.* 13, 110–122. doi: 10.1038/nrg3130

Hobolth, A., and Jensen, J. L. (2014). Markovian approximation to the finite loci coalescent with recombination along multiple sequences. *Theor. Popul. Biol.* 98, 48–58. doi: 10.1016/j.tpb.2014.01.002

Holliday, R. (1964). A mechanism for gene conversion in fungi. *Genet. Res.* 5, 282–304. doi: 10.1017/s0016672300001233

Hubisz, M., and Siepel, A. (2020). *Inference of Ancestral Recombination Graphs Using Argweaver*. Springer US, 231–266. doi: 10.1007/978-1-0716-0199-0_10

Hubisz, M. J., Williams, A. L., and Siepel, A. (2020). Mapping gene flow between ancient hominins through demography-aware inference of the ancestral recombination graph. *PLOS Genet.* 16, e1008895. doi: 10.1371/journal.pgen.1008895

Hudson, R. R. (1983). Properties of a neutral allele model with intragenic recombination. *Theor. Popul. Biol.* 23, 183–201. doi: 10.1016/0040-5809(83)90013-8

Hudson, R. R. (1991). Gene genealogies and the coalescent process. *Oxf. Surveys Evolutionary Biology* 7, 1–44.

Hudson, R. R. (2002). Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics* 18, 337–338. doi: 10.1093/bioinformatics/18.2.337

Hudson, R. R., and Kaplan, N. L. (1985). Statistical properties of the number of recombination events in the history of a sample of DNA sequences. *Genetics* 111, 147–164. doi: 10.1093/genetics/111.1.147

Ignatieva, A., Lyngsø, R. B., Jenkins, P. A., and Hein, J. (2021). Kwarg: parsimonious reconstruction of ancestral recombination graphs with recurrent mutation. *Bioinformatics* 37, 3277–3284. doi: 10.1093/bioinformatics/btab351

Ignatieva, A., Hein, J., and Jenkins, P. A. (2022). Ongoing recombination in sars-cov-2 revealed through genealogical reconstruction. *Mol. Biol. Evol.* 39, msac028. doi: 10.1093/molbev/msac028

Jukes, T. H., and Cantor, C. R. (1969). *Evolution of Protein Molecules*. Elsevier, 21–132. doi: 10.1016/b978-1-4832-3211-9.50009-7

Kelleher, J., Etheridge, A. M., and McVean, G. (2016). Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLOS Comput. Biol.* 12, e1004842. doi: 10.1371/journal.pcbi.1004842

Kelleher, J., Thornton, K. R., Ashander, J., and Ralph, P. L. (2018). Efficient pedigree recording for fast population genetics simulation. *PLOS Comput. Biol.* 14, e1006581. doi: 10.1371/journal.pcbi.1006581

Kelleher, J., Wong, Y., Wohns, A. W., Fadil, C., Albers, P. K., and McVean, G. (2019). Inferring whole-genome histories in large population datasets. *Nat. Genet.* 51, 1330–1338. doi: 10.1038/s41588-019-0483-y

Kern, A. D., and Schrider, D. R. (2016). Discoal: flexible coalescent simulations with selection. *Bioinformatics* 32, 3839–3841. doi: 10.1093/bioinformatics/btw556

Kimura, M. (1969). The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations. *Genetics* 61, 893–903. doi: 10.1093/genetics/61.4.893

Laval, G., and Excoffier, L. (2004). Simcoal 2.0: a program to simulate genomic diversity over large recombining regions in a subdivided population with a complex history. *Bioinformatics* 20, 2485–2487. doi: 10.1093/bioinformatics/bth264

Lewanski, A. L., Grundler, M. C., and Bradburd, G. S. (2024). The era of the arg: an introduction to ancestral recombination graphs and their significance in empirical evolutionary genomics. *PLOS Genet.* 20, e1011110. doi: 10.1371/journal.pgen.1011110

Li, N., and Stephens, M. (2003). Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics* 165, 2213–2233. doi: 10.1093/genetics/165.4.2213

Liang, L., Zöllner, S., and Abecasis, G. R. (2007). Genome: a rapid coalescent-based whole genome simulator. *Bioinformatics* 23, 1565–1567. doi: 10.1093/bioinformatics/btm138

Liu, Y., Athanasiadis, G., and Weale, M. E. (2008). A survey of genetic simulation software for population and epidemiological studies. *Hum. Genomics* 3, 79–86. doi: 10.1186/1479-7364-3-1-79

Lyngsø, R. B., Song, Y. S., and Hein, J. (2005). "Minimum recombination histories by branch and bound," in *Algorithms in Bioinformatics* (Springer Berlin Heidelberg), 239–250. doi: 10.1007/11557067_20

Mahmoudi, A., Koskela, J., Kelleher, J., Chan, Y.-b., and Balding, D. (2022). Bayesian inference of ancestral recombination graphs. *PLOS Comput. Biol.* 18, e1009960. doi: 10.1371/journal.pcbi.1009960

Marjoram, P., and Wall, J. D. (2006). Fast "coalescent" simulation. *BMC Genet.* 7, 1–9. doi: 10.1186/1471-2156-7-16

McVean, G. A. T., and Cardin, N. J. (2005). Approximating the coalescent with recombination. *Philosophical Trans. R. Soc. B Biol. Sci.* 360, 1387–1393. doi: 10.1098/rstb.2005.1673

Messer, P. W. (2013). SLiM: simulating evolution with selection and linkage. *Genetics* 194, 1037–1039. doi: 10.1534/genetics.113.152181

Minichiello, M. J., and Durbin, R. (2006). Mapping trait loci by use of inferred ancestral recombination graphs. *Am. J. Hum. Genet.* 79, 910–922. doi: 10.1086/508901

Mirzaei, S., and Wu, Y. (2016). Rent+: an improved method for inferring local genealogical trees from haplotypes with recombination. *Bioinformatics* 33, 1021–1030. doi: 10.1093/bioinformatics/btw735

Montemuiño, C., Espinosa, A., Moure, J.-C., Vera-Rodríguez, G., Ramos-Onsins, S., and Budé, P. H. (2014). "msPar: a parallel coalescent simulator," in *Euro-Par 2013: Parallel Processing Workshops* (Berlin Heidelberg: Springer), 321–330. doi: 10.1007/978-3-642-54420-0_32

Montemuiño, C., Espinosa, A., Moure, J. C., Vera, G., Hernández, P., and Ramos-Onsins, S. (2016). Approaching long genomic regions and large recombination rates with msParSm as an alternative to MaCS. *Evol. Bioinforma.* 12, 223–228. doi: 10.4137/ebo.s40268

Myers, S. R., and Griffiths, R. C. (2003). Bounds on the minimum number of recombination events in a sample history. *Genetics* 163, 375–394. doi: 10.1093/genetics/163.1.375

Nelson, D., Kelleher, J., Ragsdale, A. P., Moreau, C., McVean, G., and Gravel, S. (2020). Accounting for long-range correlations in genome-wide simulations of large cohorts. *PLOS Genet.* 16, e1008619. doi: 10.1371/journal.pgen.1008619

Nguyen, T. T. P., Le, V. S., Ho, H. B., and Si Le, Q. (2017). Building ancestral recombination graphs for whole genomes. *IEEE/ACM Trans. Comput. Biol. Bioinforma.* 14, 478–483. doi: 10.1109/tcbb.2016.2542801

Nielsen, R., Vaughn, A. H., and Deng, Y. (2024). Inference and applications of ancestral recombination graphs. *Nat. Rev. Genet.* 26, 47–58. doi: 10.1038/s41576-024-00772-4

Palamara, P. F., Terhorst, J., Song, Y. S., and Price, A. L. (2018). High-throughput inference of pairwise coalescence times identifies signals of selection and enriched disease heritability. *Nat. Genet.* 50, 1311–1317. doi: 10.1038/s41588-018-0177-x

Pennings, P. S., and Hermisson, J. (2006). Soft sweeps ii—molecular population genetics of adaptation from recurrent mutation or migration. *Mol. Biol. Evol.* 23, 1076–1084. doi: 10.1093/molbev/msj117

Phuong Thao, N. T., and Vinh, L. S. (2017). "Building minimum recombination ancestral recombination graphs for whole genomes," in *2017 4th NAFOSTED Conference on Information and Computer Science* (IEEE), 248–253. doi: 10.1109/nafosted.2017.8108072

Posada, D., and Wiuf, C. (2003). Simulating haplotype blocks in the human genome. *Bioinformatics* 19, 289–290. doi: 10.1093/bioinformatics/19.2.289

Rasmussen, D. A., and Guo, F. (2023). Espalier: efficient tree reconciliation and ancestral recombination graphs reconstruction using maximum agreement forests. *Syst. Biol.* 72, 1154–1170. doi: 10.1093/sysbio/syad040

Rasmussen, M. D., Hubisz, M. J., Gronau, I., and Siepel, A. (2014). Genome-wide inference of ancestral recombination graphs. *PLoS Genet.* 10, e1004342. doi: 10.1371/journal.pgen.1004342

Schaefer, N. K., Shapiro, B., and Green, R. E. (2021). An ancestral recombination graph of human, neanderthal, and denisovan genomes. *Sci. Adv.* 7, eabc0776. doi: 10.1126/sciadv.abc0776

Shchur, V. (2022). Accuracy of the SMC' approximation of structured coalescent. *Lobachevskii J. Math.* 43, 3626–3630. doi: 10.1134/s1995080222150264

Shlyakhter, I., Sabeti, P. C., and Schaffner, S. F. (2014). Cosi2: an efficient simulator of exact and approximate coalescent with selection. *Bioinformatics* 30, 3427–3429. doi: 10.1093/bioinformatics/btu562

Song, Y. S., and Hein, J. (2003). *Parsimonious Reconstruction of Sequence Evolution and Haplotype Blocks: Finding the Minimum Number of Recombination Events*. Berlin Heidelberg: Springer, 287–302. doi: 10.1007/978-3-540-39763-2_22

Song, Y. S., and Hein, J. (2005). Constructing minimal ancestral recombination graphs. *J. Comput. Biol.* 12, 147–169. doi: 10.1089/cmb.2005.12.147

Song, Y. S., Wu, Y., and Gusfield, D. (2005). Efficient computation of close lower and upper bounds on the minimum number of recombinations in biological sequence evolution. *Bioinformatics* 21, i413–i422. doi: 10.1093/bioinformatics/bti1033

Speidel, L., Forest, M., Shi, S., and Myers, S. R. (2019). A method for genome-wide genealogy estimation for thousands of samples. *Nat. Genet.* 51, 1321–1329. doi: 10.1038/s41588-019-0484-x

Speidel, L., Cassidy, L., Davies, R. W., Hellenthal, G., Skoglund, P., and Myers, S. R. (2021). Inferring population histories for ancient genomes using genome-wide genealogies. *Mol. Biol. Evol.* 38, 3497–3511. doi: 10.1093/molbev/msab174

Staab, P. R., Zhu, S., Metzler, D., and Lunter, G. (2015). Scrm: efficiently simulating long sequences using the approximated coalescent with recombination. *Bioinformatics* 31, 1680–1682. doi: 10.1093/bioinformatics/btu861

Szpiech, Z. A. (2024). Selscan 2.0: scanning for sweeps in unphased data. *Bioinformatics* 40, btae006. doi: 10.1093/bioinformatics/btae006

Teshima, K. M., and Innan, H. (2009). Mbs: modifying Hudson's ms software to generate samples of DNA sequences with a biallelic site under selection. *BMC Bioinforma.* 10, 1–4. doi: 10.1186/1471-2105-10-166

Thao, N. T. P., and Vinh, L. S. (2019). A hybrid approach to optimize the number of recombinations in ancestral recombination graphs. *Proc. of 2019 9th Int. Conf. on Biosci. Biochem. and Bioinforma. (ACM)* 19, 36–42. doi: 10.1145/3314367.3314385

Ushey, K., Allaire, J. J., and Tang, Y. (2025). *Reticulate: interface to 'Python'*. R package. Available online at: https://github.com/rstudio/reticulate (Accessed May 12, 2026).

Wang, Y., Zhou, Y., Li, L., Chen, X., Liu, Y., Ma, Z.-M., et al. (2014). A new method for modeling coalescent processes with recombination. *BMC Bioinforma.* 15, 273. doi: 10.1186/1471-2105-15-273

Wilton, P. R., Carmi, S., and Hobolth, A. (2015). The SMC' is a highly accurate approximation to the ancestral recombination graph. *Genetics* 200, 343–355. doi: 10.1534/genetics.114.173898

Wiuf, C., and Hein, J. (1999). Recombination as a point process along sequences. *Theor. Popul. Biol.* 55, 248–259. doi: 10.1006/tpbi.1998.1403

Wiuf, C., and Hein, J. (2000). The coalescent with gene conversion. *Genetics* 155, 451–462. doi: 10.1093/genetics/155.1.451

Wong, Y., Ignatieva, A., Koskela, J., Gorjanc, G., Wohns, A. W., and Kelleher, J. (2024). A general and efficient representation of ancestral recombination graphs. *Genetics* 228, iyae100. doi: 10.1093/genetics/iyae100

Wu, Y. (2008). Association mapping of complex diseases with ancestral recombination graphs: models and efficient algorithms. *J. Comput. Biol.* 15, 667–684. doi: 10.1089/cmb.2007.0116

Wu, Y. (2011). New methods for inference of local tree topologies with recombinant snp sequences in populations. *IEEE/ACM Trans. Comput. Biol. Bioinforma.* 8, 182–193. doi: 10.1109/tcbb.2009.27

Yang, T., Deng, H.-W., and Niu, T. (2014). Critical assessment of coalescent simulators in modeling recombination hotspots in genomic sequences. *BMC Bioinforma.* 15, 1–14. doi: 10.1186/1471-2105-15-3

Yuan, X., Miller, D. J., Zhang, J., Herrington, D., and Wang, Y. (2012). An overview of population genetic data simulation. *J. Comput. Biol.* 19, 42–54. doi: 10.1089/cmb.2010.0188

Zhang, B. C., Biddanda, A., Gunnarsson, A. F., Cooper, F., and Palamara, P. F. (2023). Biobank-scale inference of ancestral recombination graphs enables genealogical analysis of complex traits. *Nat. Genet.* 55, 768–776. doi: 10.1038/s41588-023-01379-x