An Efficient Algorithm for Estimating Population History from Genetic Data

Alan R. Rogers

May 3, 2021

Abstract

~~Legofit is a statistical package that~~ The Legofit statistical package uses genetic data to estimate ~~the history of population size, subdivision, and admixture~~parameters describing population history. Previous versions used computer simulations to estimate probabilities, an approach that limited both speed and accuracy. This article describes a new deterministic algorithm, which makes Legofit ~~orders of magnitude~~ faster and more accurate. The speed of this algorithm declines as model complexity increases. With very complex models, the deterministic algorithm is slower than the stochastic one. In an application to simulated data sets, the estimates produced by the deterministic and stochastic algorithms were essentially identical. Reanalysis of a human data set replicated the findings of a previous study and provided increased support for the hypotheses that (a) early modern humans contributed genes to Neanderthals, and (b) a "superarchaic" population (which separated from all other humans early in the Pleistocene) was either large or deeply subdivided.

1 Introduction

~~Legofit [20–22] estimates parameters~~ Legofit is a publicly-available statistical package that uses genetic data to estimate the history of size, subdivision, and gene flow within a set of populations.[1] Because it ignores the within-population component of genetic variation, it avoids the need to estimate parameters describing recent population history and is able to focus on a deeper time scale. It operates by fitting models of history to the frequencies of "nucleotide site patterns," which describe the sharing of derived alleles by subsets of populations. ~~The estimation process searches for a set of parameter values that maximize the~~ In recent publications, it has shown that Neanderthals and Denisovans separated earlier than previously thought, that their ancestors endured a bottleneck in population size, and that these ancestors interbred with a preexisting "superarchaic" population, which had inhabited Eurasia since early in the Pleistocene. It has also confirmed a variety of results first obtained by other methods [20–22].

Legofit's estimation procedure evaluates the fit of model to data ~~. This involves evaluating~~ at many sets of ~~values, and in~~ parameter values. In previous versions of Legofit, each evaluation required a lengthy computer simulation. These calculations were feasible ~~,~~because they could be done in parallel. Nonetheless, Legofit was ~~practical only~~ most useful on high-performance computing clusters. This stochastic algorithm also limited the accuracy with with models could be fit to data.

This article describes a new deterministic algorithm, which ~~provides an enormous increase in~~ increases both speed and accuracy. With the simulated data discussed below, the deterministic

[1] `https://github.com/alanrogers/legofit`

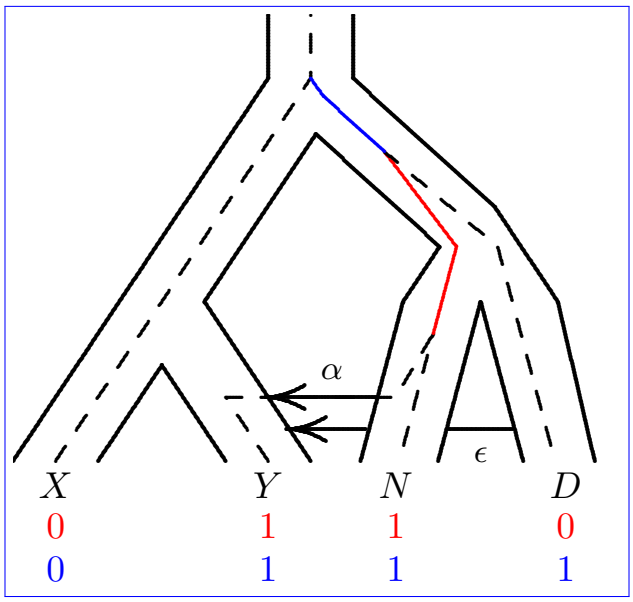

Figure 1: Population network with embedded gene tree. A mutation on the solid red branch would generate site pattern *yn* (shown in red at the base of the tree). One on the solid blue branch would generate *ynd*. "0" and "1" represent the ancestral and derived alleles. Key: X, Africa; Y, Eurasia; N, Neanderthal; D, Denisovan. After Rogers [20, Fig. 1].

algorithm is over 1600 times as fast as the stochastic one. ~~And because~~ Because of its greater accuracy, it also provides a better fit of model to data and improves Legofit's ability to discriminate among models.

2 Methods

The new algorithm involves two novel components. The first of these involves a well-known Markov chain [8, 23, 26] that is seldom used because of the numerical difficulties. Below, section 2.3 shows a way around these difficulties. The new algorithm also relies on two results describing how descendants are partitioned among ancestors. One of these (Eqn. 7) is old and the other (Eqn. 8) new. Before discussing these, however, let us review the basics of Legofit. As in previous publications, I use capitalization to distinguish the Legofit package from the legofit program within that package.

2.1 Model of population history

Fig. 1 shows a gene tree embedded within a network of populations. In Legofit, the population network is ~~modelled~~ modeled as a set of connected segments, each with a simple history. Each segment describes a single randomly-mating population, during an interval of constant population size. The root segment has no parent, and tip segments have no children. All other segments have at least one parent and one child. Segments that receive gene flow have two parents: one for native ancestors and the other for immigrants. Most segments have finite length, but the root segment is infinite.

The population history in Fig. 1 could be modeled using the network of segments in Fig. 2. Note that the branch ending at Y in Fig. 1 has three segments (y, y1, and y2) in Fig. 2. This is

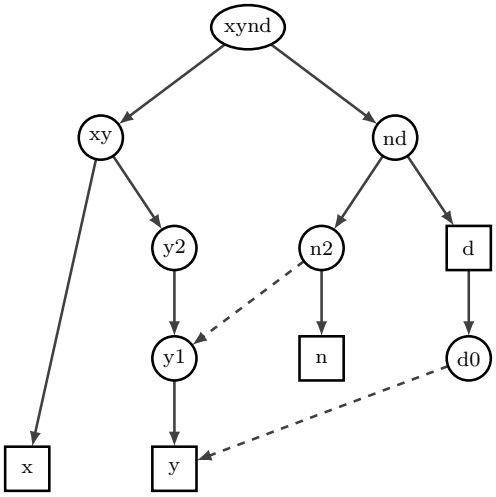

Figure 2: Network of segments used in legofit analysis. Squares represent segments from which we have "observed" (i.e. simulated) data. Arrows indicate ancestor-descendant relationships, and dashed lines represent gene flow. Segments in the same row need not be contemporary.

because that branch is interrupted by two episodes of gene flow, and gene flow can occur only at the ancient end of a segment. Thus, segment y extends from the present back to the first episode of gene flow, y1 extends from the first episode to the second, and y2 extends from the second episode back to the separation of populations X and Y.

The size of population Y cannot be estimated, because there is never more than a single lineage within Y. At time zero, there is a single haploid sample, because Y is a population that has been sampled. This lineage may derive from segment d0, from n2, or from y2. But there is no way, under this model of history, for any of the segments that compose Y to contain more than one lineage. Consequently, no coalescent events are possible within Y, and its population size does not affect site pattern frequencies. This population size is therefore treated as a fixed constant rather than a parameter to be estimated.

On the other hand, segment n2 may contain either 1 or 2 lineages. It will always contain at least 1 lineage, which is ancestral to the lineage sampled in segment n. In addition, it may contain the lineage sampled in segment y. Consequently, population size in segment n2 is an estimable parameter.

In order to reduce the parameter count, it is possible to specify that several segments share a single population-size parameter.

2.2 Nucleotide site patterns

Legofit works with the frequencies of *nucleotide site patterns*, which are illustrated in Fig. 1. A nucleotide site exhibits the *yn* site pattern if random nucleotides drawn from populations Y and N carry the derived allele, but those drawn from other populations carry the ancestral allele. Fig. 1 shows the gene genealogy of a particular nucleotide site, embedded within the network of populations. A mutation on the red branch would generate site pattern *yn*, whereas one on the blue branch would generate *ynd*. Mutations elsewhere would generate other site patterns. The gene genealogy will vary from locus to locus, so averaging across the genome involves averaging across gene genealogies. We are interested in the properties of such averages.

Let B_i represent the length in generations of the branch generating site pattern i. I employ the

"infinite sites" model of mutation [10], which assumes that the mutation rate is small enough that we can ignore the possibility of multiple mutations on any given branch. Under this assumption, a polymorphic site exhibits pattern i with probability

$$P_i = \frac{E[B_i]}{\sum_{j\in\Omega} E[B_j]} \tag{1}$$

where $E[B_i]$ is the expected length of the branch generating site pattern i, and Ω is the set of site patterns under study [20, Eqn. 1]. Previous versions of Legofit used coalescent simulations to estimate these expectations. The sections that follow describe a deterministic algorithm.

2.3 The matrix coalescent

The new algorithm is based on a model that calculates the probability that there are k ancestral lineages at the ancient end of a segment, given that there are n descendant lineages at the recent end. This model also calculates the expected length of the interval within the segment during which there are k lineages, where $1 \le k \le n$. The model employs a continuous-time Markov chain[23, appendix I; 8; 26], which begins with n haploid lineages at the recent end of the segment. As we trace the ancestry of this sample into the past, the original sample of n lineages falls to $n-1$, then $n-2$, and so on until only a single lineage is left, or we reach the end of the segment.

The number, n, of descendants equals 1 for tip segments. For ancestral segments, n may take several values with different probabilities. The legofit program sums across these possibilities, weighting by probability.

This Markov chain is well known [23, appendix I; 8; 26] but seldom used, because accurate calculations are difficult with samples of even modest size. Legofit, however, is designed for use with small samples. Furthermore, it is possible (as shown below) to factor the calculations into two steps, one of which can be done in exact arithmetic ,and only needs to be done once at the beginning of the computer program. Numerical error arises only in the second step, and as we shall see, that error is small.

Within a segment, the population has constant haploid size $2N$, although $2N$ can vary among segments. ("Haploid" population size is twice the number of diploid individuals.) It will be convenient to measure time backwards from the recent end of each segment in units of $2N$ generations. On this scale, time is $v = t/2N$, where t is time in generations. Let $\mathbf{x}(v)$ denote the column vector whose ith entry, $x_i(v)$, is the probability of observing i lineages at time v, where $1 \le i \le n$. I ignore the absorbing state x_1, so that indices of arrays and matrices range from 2 to n. Because there are n lineages at time zero (the recent end of the segment), the initial vector equals $\mathbf{x}(0) = [0, \ldots, 0, 1]^T$. At time v [26, Eqn. 8],

$$\mathbf{x}(v) = \mathbf{C}\mathbf{E}(v)\mathbf{R}\mathbf{x}(0) \tag{2}$$

Here, $\mathbf{E}(v)$ is a diagonal matrix of eigenvalues whose ith diagonal entry is $e^{-\beta_i v}$, where $\beta_i = i(i-1)/2$. $\mathbf{C} = [c_{ij}]$ and $\mathbf{R} = [r_{ij}]$ are matrices of column eigenvectors and row eigenvectors, both of which are upper triangular. They are calculated by setting diagonal entries equal to unity, and then applying [26, p. 1642],

$$\begin{aligned} c_{i,j} &= c_{i+1,j} \times \left(\frac{i(i+1)}{i(i-1) - j(j-1)} \right), \quad i = j-1, \ldots, 2 \\ r_{i,j} &= r_{i,j-1} \times \left(\frac{j(j-1)}{j(j-1) - i(i-1)} \right), \quad j = i+1, \ldots, n \end{aligned}$$

Let $\mathbf{m}(v)$ denote the vector whose kth entry, $m_k(v)$, is the expected duration (in units of $2N$ generations) of the interval during which the segment contains k lineages, within a segment of length v. This vector equals

$$\mathbf{m}(v) = \mathbf{B}^{-1}(\mathbf{x}(v) - \mathbf{x}(0)) \tag{3}$$

where

$$\mathbf{B} = \begin{pmatrix} -\beta_2 & \beta_3 & & \\ & -\beta_3 & \ddots & \\ & & \ddots & \beta_n \\ & & & -\beta_n \end{pmatrix}$$

Eqn. 3 holds not only for finite segments, but also when $v \to \infty$. In the infinite case, $\mathbf{x}(\infty) = 0$, because we are considering only the transient states $(x_2, \ldots, x_n)$, which disappear in the long run. Eqn. 3 is easy to calculate, because $\mathbf{B}^{-1}$ has a simple form. For the case of $n = 4$,

$$\mathbf{B}^{-1} = \begin{pmatrix} -1/\beta_2 & -1/\beta_2 & -1/\beta_2 \\ & -1/\beta_3 & -1/\beta_3 \\ & & -1/\beta_4 \end{pmatrix}.$$

This model presents challenging numerical issues. To deal with these, let us re-organize the calculations to do as much as possible in exact arithmetic. I illustrate this re-organization using the case of $n = 3$, for which Eqn. 2 becomes

$$\begin{aligned} \mathbf{x}(v) &= \begin{pmatrix} 1 & -3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{-\beta_2 v} & 0 \\ 0 & e^{-\beta_3 v} \end{pmatrix} \begin{pmatrix} 1 & 3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & -3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{-\beta_2 v} & 0 \\ 0 & e^{-\beta_3 v} \end{pmatrix} \begin{pmatrix} 3/2 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & -3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 3/2 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{-\beta_2 v} \\ e^{-\beta_3 v} \end{pmatrix} \\ &= \begin{pmatrix} 3/2 & -3/2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} e^{-\beta_2 v} \\ e^{-\beta_3 v} \end{pmatrix} \\ &= \mathbf{G}\mathbf{w}(v) \end{aligned} \tag{4}$$

where $\mathbf{w}(v) = (e^{-\beta_2 v}, e^{-\beta_3 v})^T$ is a vector of eigenvalues, $\mathbf{G} = \mathbf{C}\,\mathrm{diag}(\mathbf{R}\mathbf{x})$ is a matrix of column eigenvectors with columns scaled by the entries of vector $\mathbf{R}\mathbf{x}(0)$, and $\mathrm{diag}(\mathbf{R}\mathbf{x}(0))$ is a diagonal matrix whose main diagonal equals the vector $\mathbf{R}\mathbf{x}(0)$. The matrix $\mathbf{G}$ can be calculated in exact rational arithmetic. This is done at the beginning of the computer program for each possible value of n, and the resulting values are stored for later use.

Next, substitute (4) into (3) to obtain

$$\mathbf{m}(v) = \mathbf{z} + \mathbf{H}\mathbf{w}(v) \tag{5}$$

where $\mathbf{z} = -\mathbf{B}^{-1}\mathbf{x}(0) = (1/\beta_2, \ldots, 1/\beta_n)^T$, and $\mathbf{H} = \mathbf{B}^{-1}\mathbf{G}$, both of which can be calculated in advance for each possible value of n, using exact arithmetic. For example, if $n = 3$,

$$\mathbf{m}(v) = \begin{pmatrix} 1 \\ 1/3 \end{pmatrix} + \begin{pmatrix} -3/2 & 1/2 \\ 0 & -1/3 \end{pmatrix} \begin{pmatrix} e^{-\beta_2 v} \\ e^{-\beta_3 v} \end{pmatrix}$$

In an infinite segment, Eqn. 5 is simply $\mathbf{m}(\infty) = \mathbf{z}$.

This algorithm calculates $x_k(v)$ and $m_k(v)$ only for $k = 2, 3, \ldots, n$. Values for $k = 1$ are obtained by subtraction: $x_1(v) = 1 - \sum_{k=2}^{n} x_k(v)$, and $m_1(v) = v - \sum_{k=2}^{n} m_k(v)$. Finally, to re-express $m_k(v)$ in units of generations, define

$$L_k(t, 2N) = 2N m_k(t/2N) \tag{6}$$

where t is the length of the current segment in generations, and $2N$ is its haploid population size. $L_k(t, 2N)$ is the expected duration in generations of the interval during which the current segment contains k lineages.

Several of the quantities in this algorithm—$\mathbf{G}$, $\mathbf{H}$, and $\mathbf{z}$—are calculated in exact rational arithmetic. Although there is no roundoff error, these calculations will overflow if n is too large. With 32-bit signed integers, there is no overflow until $n > 35$. This is more than enough for Legofit, which requires that $n \leq 32$, so that site patterns can be represented by the bits of a 32-bit integer.

Roundoff error does occur in this algorithm, because all quantities are eventually converted to double-precision floating point during the calculation of Eqns. 4 and 5. To assess the magnitude of this error, I compared results to calculations done in 256-bit floating-point arithmetic, using the Gnu MPFR library [7]. I considered values of v ranging from 0 to 9.5 in steps of 0.5, and also $v \to \infty$. The maximum absolute error is 3.553×10^{-15} when $n = 8$; 2.700×10^{-13} when $n = 16$; and 1.543×10^{-8} when $n = 32$. These errors are all much smaller than those of Legofit's stochastic algorithm.

The theory just described allows us to calculate the probability that n descendants have $k \leq n$ ancestors in some previous generation. To relate this theory to the frequencies of site patterns, we must discuss how the coalescent process partitions descendants among ancestors.

2.4 Partitioning ~~samples~~ descendants among ancestors

A "segment" is an interval within the history of one subpopulation. Let n represent the number of descendant lineages at the recent end of the segment, and let $k \leq n$ represent the number of ancestral lineages at some earlier point within the segment. The theory in section 2.3 calculates the probability of k ~~given n, v, and $2N$. It~~ at any time within the segment and also provides the expected length of the ~~interval during which there are~~ subinterval containing k lines of descent.

For all segments except the root, we need both of these quantities. We need the expected lengths of subintervals, because these lengths measure the opportunity for mutation. In addition, we need to assign a probability to each of the ways in which the set of descendants can be partitioned among ancestors at the ancient end of the segment. These partitions and probabilities are used in calculations on earlier segments within the network.

For the root segment, we still need the expected ~~length of the subinterval within which there are k lineages~~lengths of subintervals. But because there are no earlier segments to worry about, we don't need to assign probabilities to partitions. This is fortunate, because the number of set partitions increases rapidly with the size of the set [11, p. 418], and the set of descendants is largest in the root segment.

To address these needs, I present two algorithms. One sums across partitions of the set of descendants and is used in all segments except the root. The other avoids this sum and is used only at the root.

2.4.1 Summing across set partitions

Section 2.3 calculated the expected length of the interval during which there are k ancestors, given that there are n descendants at the recent end of the segment. If a mutation strikes one ancestor,

Table 1: Set partitions, integer partitions, and their probabilities, for the case in which $n = 4$ and $k = 2$. Under "set partitions," the value in position j of each string is the index of the ancestor of descendant j. Thus, "1122" means that descendants 1 and 2 descend from one ancestor, whereas 3 and 4 descend from another. Ancestors are numbered in order of their appearance in the list of descendants. Integer partitions are discussed in section A.2 of the appendix.

Set partitions	Pr	Integer partitions	Pr
1112	1/6	$3 + 1$	2/3
1121	1/6		
1211	1/6		
1222	1/6		
1122	1/9	$2 + 2$	1/3
1212	1/9		
1221	1/9		

it will be shared by all descendants of that ancestor. The subset comprising these descendants corresponds to a nucleotide site pattern.

Suppose that at some time in the past there were k ancestors. These ancestors partition the set of descendants into k subsets. Let $x_1, x_2, \ldots, x_k$ denote the sizes of the k subsets, i.e., the numbers of descendants of the k ancestors. The conditional probability, given k, of such a partition is [3, theorem 1.5, p. 11]

$$A = k! \binom{n-1}{k-1}^{-1} \binom{n}{x_1, \ldots, x_k}^{-1} \tag{7}$$

The left side of table 1 shows all ways of partitioning a set of 4 descendants among 2 ancestors along with the probability of each partition. The descendants of each ancestor define a nucleotide site pattern. For example, the first partition is "1112," which says that the first three descendants share a single ancestor. A mutation in this ancestor would be shared by these descendants, and so the descendants correspond to a site pattern.

This result is used in an algorithm that calculates (a) all possible partitions of descendants at the ancient end of the segment along with their probabilities, and (b) the contribution of the current segment to the expected branch length of each site pattern. The algorithm loops first across values of k, where $1 \leq k \leq n$. For each k, it loops across set partitititions using Ruskey's algorithm [11, pp. 764–765]. The probability that a given partition occurs at the ancient end of a segment, given the set of descendants at its recent end, is the product of $x_k(t/2N)$ (Eqn. 2) and A (Eqn. 7). Each partition also makes a contribution to the expected branch length associated with k site patterns—one for each ancestor. That contribution is the product of $L_k(t, 2N)$ (Eqn. 6) and A (Eqn. 7). These contributions are summed across partitions and segments to obtain the expected branch length of each site pattern.

2.4.2 A faster algorithm for the root segment

Consider the event that a particular set of d descendants (and no others) descend from a single ancestor in some previous generation, given that there were k ancestors in that generation. This event is of interest, because a mutation in this ancestor would be shared uniquely by the d descendants.

The probability of this event is

$$Q_{dk} = \begin{cases} 1 & \text{if } k = 1 \\ k\binom{n-d-1}{k-2}\binom{n-1}{k-1}^{-1}\binom{n}{d}^{-1} & \text{if } k > 1 \end{cases} \tag{8}$$

To justify this result, consider first the case in which $k = 1$. This requires that all n descendants descend from a single ancestor, so d must equal n. There is only one way this can happen, and because the probability distribution must sum to 1, it follows that $Q_{dk} = 1$. The result for $k > 1$ is derived in appendix A.

Example 1 Suppose $k = n$. In this case, each ancestor has 1 descendant, so $d = 1$, and $Q_{1,n}$ must equal 1. Equation 8 agrees:

$$Q_{1,n} = n\binom{n-2}{n-2}\binom{n-1}{n-1}^{-1}\binom{n}{1}^{-1} = n \times 1 \times 1 \times \frac{1}{n} = 1$$

Example 2 Suppose that $k = n - 1$. In this case, we are reckoning descent from the previous coalescent interval, in which there were $n - 1$ ancestors. Consider first the case in which $d = 1$. Among the n descendants, 2 derive from an ancestor that split, and $n - 2$ derive from one that did not split. This implies that $Q_{1,n-1}$ equals $(n - 2)/n$, the probability a random descendant derives from an ancestor that did not split.

The case of $d = 2$ is also easy. There are $\binom{n}{2}$ ways to choose 2 descendants from n, and only one of these pairs derives from a single ancestor in the previous coalescent interval. Thus, $Q_{2,n-1} = \binom{n}{2}^{-1}$. Equation 8 confirms both of these results:

$$\begin{aligned} Q_{1,n-1} &= (n-1)\binom{n-2}{n-3}\binom{n-1}{n-2}^{-1}\binom{n}{1}^{-1} \\ &= (n-1) \times (n-2) \times \frac{1}{n-1} \times \frac{1}{n} = (n-2)/n \\ Q_{2,n-1} &= (n-1)\binom{n-3}{n-3}\binom{n-1}{n-2}^{-1}\binom{n}{2}^{-1} \\ &= (n-1) \times 1 \times \frac{1}{n-1} \times \binom{n}{2}^{-1} = \binom{n}{2}^{-1} \end{aligned}$$

Example 3 We can also evaluate Eqn. 8 by comparing its results to Eqn. 7. Table 1 shows all partitions and their probabilities for the case in which $k = 2$ and $n = 4$. Notice that subsets of sizes 1, 2, and 3 have probabilities 1/6, 1/9, and 1/6. Eqn. 8 yields identical values:

$$\begin{aligned} Q_{1,2} &= 2\binom{2}{0}\binom{3}{1}^{-1}\binom{4}{1}^{-1} = 2 \times 1 \times \frac{1}{3} \times \frac{1}{4} = 1/6 \\ Q_{2,2} &= 2\binom{1}{0}\binom{3}{1}^{-1}\binom{4}{2}^{-1} = 2 \times 1 \times \frac{1}{3} \times \frac{1}{6} = 1/9 \\ Q_{3,2} &= 2\binom{0}{0}\binom{3}{1}^{-1}\binom{4}{3}^{-1} = 2 \times 1 \times \frac{1}{3} \times \frac{1}{4} = 1/6 \end{aligned}$$

In the root segment, the program uses the following algorithm: Loop first across values of k, where $1 \le k \le n$. For each k, loop across values of d. If $k = 1$, then $d = n$. Otherwise, d can take

any integer value such that $1 \leq d \leq n - k + 1$. For each d, calculate Q_{dk} using Eqn. 8, and loop across ways of choosing d of n descendants, using algorithm T of Knuth [11, p. 359]. Each such choice corresponds to a nucleotide site pattern. Add $Q_{dk}L_k(t, 2N)$ to the expected branch length associated with this site pattern.

2.5 Simulated data sets

To evaluate the new algorithm, I used 50 data sets simulated with msprime [9], using the model in Fig. 1, which is identical to that used in a previous publication [20]. ~~Parameters are defined in the caption of Fig. 3. There are 11 free parameters. Code and numerical values of simulation parameters are~~ Each simulated genome consisted of 1000 chromosomes, each with 2×10^6 nucleotide sites. Each simulated data set consisted of 4 genomes, one each from populations X, Y, N, and D, which represent the African, European, Neanderthal, and Denisovan populations. XY is the population ancestral to X and Y, ND is that ancestral to N and D, and $XYND$ is that ancestral to X, Y, N, and D. The mutation rate was 1.4×10^{-8} per base pair per generation, and the recombination rate was 10^{-8} per base pair per generation.

The time parameters in the simulation model, expressed in generations, are as follows:

T_{XYND}	=	25920	separation of XY and ND
T_{ND}	=	15000	separation of N and D
T_{XY}	=	3788	separation of X and Y
T_D	=	1734	age of Denisova fossil
T_A	=	1760	age of Altai Neanderthal fossil
T_α	=	1897	time of Neanderthal admixture
T_ϵ	=	1896	time of Denisovan admixture

Admixture proportions are:

m_α	=	0.05	fraction of segment y2 derived from n2
m_ϵ	=	0.025	fraction of segment y derived from d0

Population sizes are expressed as "haploid" counts, which represent twice the number of diploid individuals. These parameters are:

$2N_{XYND}$	=	64964.1	ancestral population $XYND$
$2N_{XY}$	=	44869.2	population ancestral to X and Y
$2N_{ND}$	=	5000	population ancestral to N and D
$2N_N$	=	9756.8	Neanderthal population, N
$2N_D$	=	5000	Denisovan population, D
$2N_X$	=	20000	modern African population, X
$2N_Y$	=	20000	modern European population, Y

Simulation code is in section S1 of Supplementary Materials. ~~All analyses are available~~ Simulation results are in the archive (`doi:10.17605/OSF.IO/74BJF`).

2.6 ~~Data analysis~~Analysis of simulated data

The data analysis pipelines for ~~both~~ the deterministic and stochastic algorithms are detailed in supplementary section S2. In both cases, the analysis was based on a model of history specified by

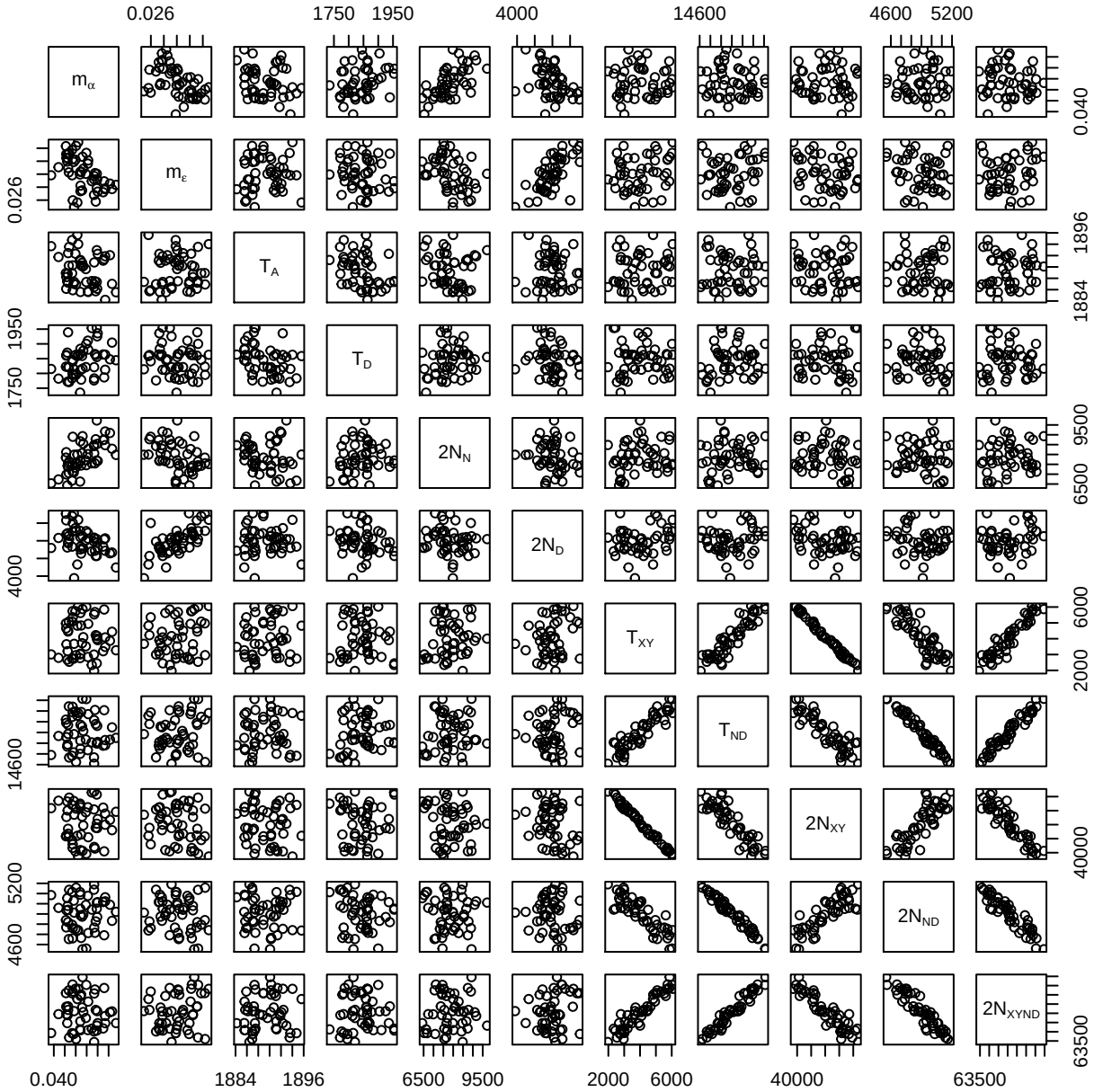

Figure 3: Scatter plot of each parameter against each other, based on 50 simulated data setssimulated under the model in Fig. 1. Key: m_α, fraction of admixture from N into Y; m_ϵ, fraction of admixture from D into Y; T_{XY}, separation time of X and Y; T_{ND} separation time of N and D, T_A, age of fossil genome from population N; T_D, age of fossil from D; N_{XYND}, size of ancestral population; N_{XY}, size of population ancestral to X and Y; N_{ND}, size of population ancestral to N and D; N_N, size of population N; N_D, size of population N. The separation time, T_{XYND}, of XY and ND was fixed exogeneously to calibrate the molecular clock.

the input file `a.lgo` (supplementary section SB.1). This file defines the network of segments shown in Fig. 2.

Several of the parameters of the simulation model were treated as fixed constants, because their values have no effect on expected site pattern frequencies: $2N_X$, $2N_Y$, T_a, and T_c. Another parameter, T_{XYND}, was fixed at its true value to calibrate the molecular clock. The remaining 11 parameters were estimated.

For both algorithms, data analysis ~~involves~~ involved 5 stages. In stage 1, `legofit` ~~is~~ was run on each of 50 simulated data sets. Each run ~~produces~~ produced two output files: a `.legofit` file, which contains parameter estimates, and a `.state` file, which records the state of the optimizer at the end of the run. The optimizer uses the *differential evolution* algorithm [17]~~, which~~. This algorithm maintains a swarm of points, each of which represents a guess about the values of the free parameters. There are ten times as many points as free parameters~~. Each point represents a guess as to the values of the 11 free parameters~~, as recommended by Price et al. [17].

Although differential evolution is good at finding global optima, it is possible that some of the stage 1 runs will get stuck on different local optima. Stage 2 is designed to avoid this problem. Each job in stage 2 begins by reading all 50 of the `.state` files produced in stage 1, and sampling among these to construct a swarm of points. This allows legofit to choose among local optima.

Figure 3 plots pairs of free parameters after stage 2 of the analysis. Each sub-plot has 50 points, ~~each of which represents one of the simulated data sets. As you can see, several of the~~ one for each simulated data set. Several pairs of parameters are tightly correlated~~with each other. These~~, and these correlations reflect "identifiability" problems: different sets of parameter values imply almost identical site pattern frequencies. To ameliorate this problem, stage 3 of the analysis ~~performs~~ uses the *pclgo* program to perform a principal components analysis, which re-expresses the free variables in terms of uncorrelated principal components (PCs). In previous publications [20–22], we ~~have~~ used this step to reduce the dimension of the analysis, by excluding components that explain little of the variance. However, excluding dimensions can introduce bias, especially in the presence of identifiability problems, so I chose here to retain the full dimension. Even without any reduction in dimension, re-expression in terms of PCs improves the fit of model to data, because it allows legofit to operate on uncorrelated dimensions.

Stages 4 and 5 are like stages 1 and 2, except that the free variables are re-expressed in terms of ~~principal components~~PCs.

The program uses KL divergence [13] to measure the discrepancy between observed and predicted site pattern frequencies. Minimizing KL divergence is equivalent to maximizing multinomial composite likelihood. The optimizer stops after a fixed number of iterations or when the difference between the best and worst KL divergences falls to a pre-determined threshold. This threshold was 3×10^{-6} for the deterministic algorithm and 2×10^{-5} for the stochastic algorithm. This difference reflects the fact that the deterministic algorithm is capable of much greater precision.

2.7 Analysis of speed as a function of model complexity

As model complexity increases, the number of states increases. This reduces the speed of the deterministic algorithm and increases memory usage. To study this effect, I used the *legosim* program, which calculates the site pattern frequencies implied by a given model. I studied a series of models without migration or changes in population size. The models differed in the number of populations, which ranged from four to nine. Timings were done on a 2018 MacBook Air.

2.8 Analysis of real data

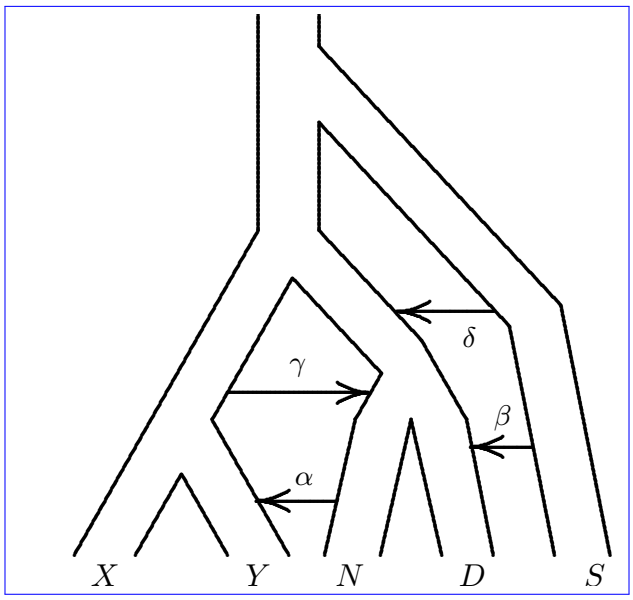

Figure 4: A population network including four episodes of gene flow. Upper case letters (X, Y, N, D, and S) represent populations (Africa, Europe, Neanderthal, Denisovan, and superarchaic). Greek letters label episodes of admixture.

I used the deterministic algorithm to replicate the analysis of Rogers et al. [22]. (Data sets and analysis files are in directory `xyvad` of the archive (`doi:10.17605/OSF.IO/74BJF`).) That paper studied modern human sequence data from Europe and Africa [15], along with three high-coverage archaic genomes: two Neanderthals (Altai [19] and Vindija [18]), and one Denisovan [16]. It analyzed these data under eight different models, all of which are based on the history in Fig. 4.

In that figure, capital roman letters refer to populations: X is Africa, Y is Europe, N is Neanderthal, D is Denisovan, and S (for "superarchaic" [19]) is a population that separated from all other humans early in the Pleistocene. Greek letters label episodes of admixture. Episode α refers to admixture from Neanderthals into Europeans, β to admixture from superarchaics into Denisovans [12, 18, 19, 24, 25], γ to admixture from early moderns into Neanderthals [12], and δ to admixture from superarchaics into the "neandersovan" ancestors of Neanderthals and Denisovans [22].

Following Rogers et al. [22], I considered eight models, all of which include α, and including all combinations of β, γ, and/or δ. I label models by concatenating Greek letters. For example, $\alpha\beta$ is the model that includes α and β but not γ and δ. This analysis is described in section S3 of the supplement.

3 Results and Discussion

I used both algorithms—one deterministic and the other stochastic—to fit 50 ~~data sets simulated using the model in Fig. 1.~~ simulated data sets. In each case, this involved 200 runs of the legofit program—4 for each of 50 data sets—and 1 run of pclgo. Altogether, the deterministic version of this analysis took 18.7 CPU minutes. Because these calculations were parallelized, the elapsed time was only 1.7 minutes. Using the stochastic algorithm, the same analysis took 514.8 CPU hours, or 11.4 hours of elapsed time. For this model, the deterministic algorithm is 1654 times as fast as the

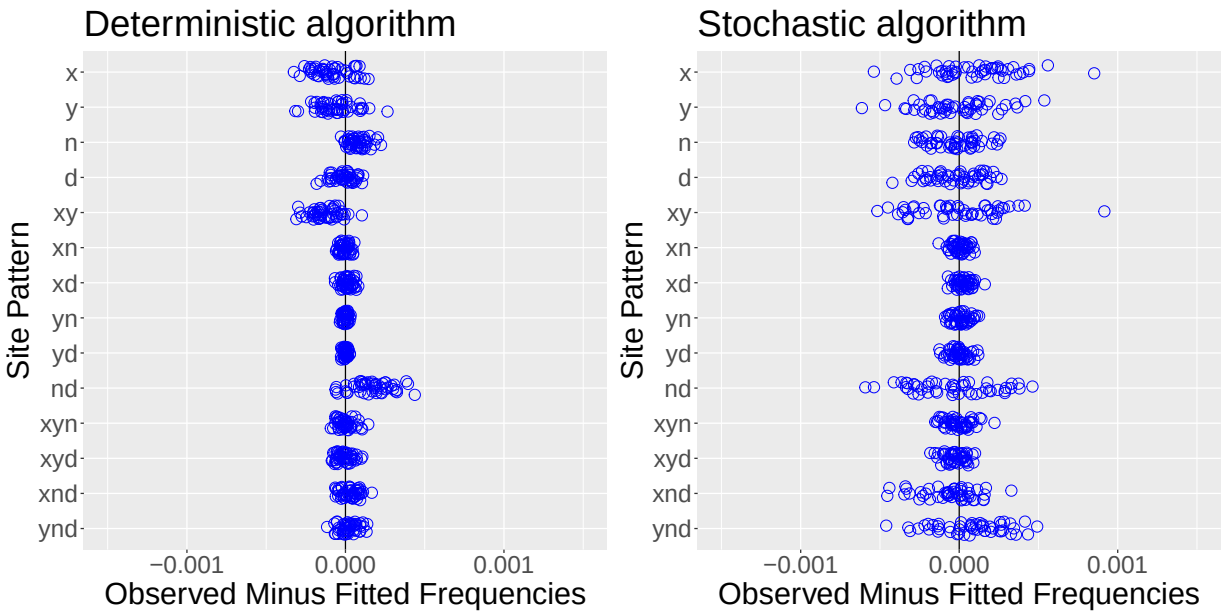

Figure 5: Residual error of deterministic and stochastic algorithms, based on 50 simulated data sets. Each circle refers to a different simulated data set.

stochastic one.

These timings were done on a node at the Center for High Performance Computing (CHPC) at the University of Utah, using 96 parallel threads of execution. To get a sense of how long these calculations would take on a less powerful computer, I did one run of legofit on a 2018 MacBook Air, using the deterministic algorithm with 2 threads. That took 26.2 seconds of CPU time or 13.7 seconds of elapsed time. By comparison, the CHPC node did this job in 12.4 seconds of CPU time, or 1 second of elapsed time. The high-performance node is nearly 14 times as fast as the MacBook Air, implying that the full analysis would take 24 minutes on the MacBook Air. Thus, the deterministic algorithm makes Legofit feasible on small computers.

Figure 5 shows the residual error in site pattern frequencies under the two algorithms. Residuals are substantially smaller under the deterministic algorithm because of its greater accuracy. When parameters are estimated by computer simulation, each additional decimal digit of precision requires a 100-fold increase in the number of iterations. This imposes a limit on the accuracy of the stochastic algorithm, even with the fastest computers.

To estimate site pattern frequencies, both algorithms integrate over the states of the stochastic process. The number of states increases with model complexity, so both algorithms are slower when the model is complex. Figure 6 illustrates the effect on speed. In complex models, the stochastic algorithm is faster than the deterministic one.

Figure 7 shows the parameter estimates from the 50 data sets (blue dots) along with the true parameter values (red crosses). The two algorithms behave similarly. It does not appear that the smaller residual error of the deterministic algorithm (Fig. 5) translates into more accurate parameter estimates. ~~Presumably, this is~~ This is probably because most of the spread in the parameter estimates reflects the identifiability problems seen in Fig. 3. To understand this effect, note the tight correlation between T_{XY} and $2N_{XY}$ in Fig. 3. This correlation exists because it is hard to distinguish the case in which $2N_{XY}$ is large and T_{XY} small from that in which the opposite is true. Because of this ambiguity, both parameters exhibit large uncertainties in Fig. 7.

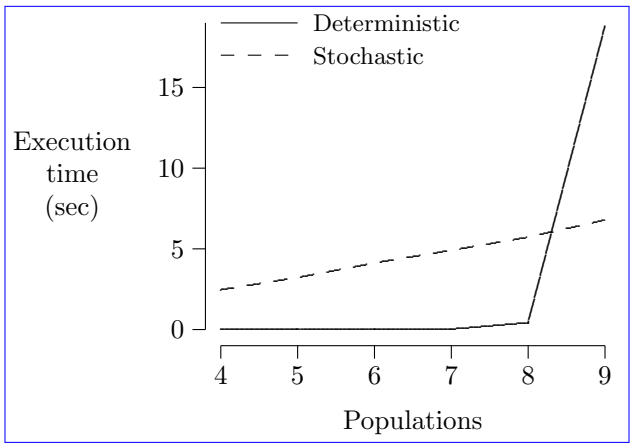

Figure 6: Execution time of legosim, excluding system calls, in models without migration. For the stochastic algorithm, each run used two million iterations.

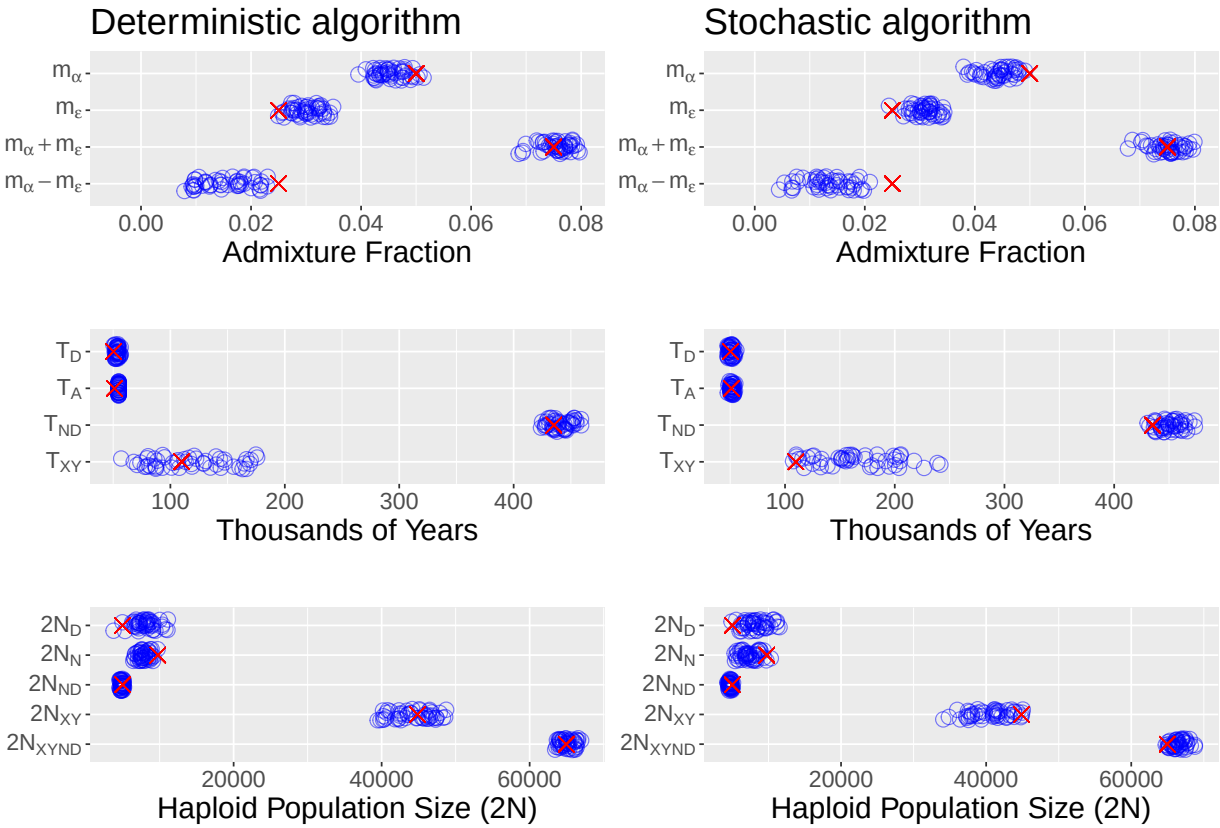

Figure 7: Parameter estimates from 50 simulated data sets, using the deterministic and stochastic algorithms. Blue circles are estimates and red crosses are the true parameter values.

Table 2: CPU time expended in analysis of each model from Rogers et al. [22]. Each analysis involves 204 runs of legofit and 1 run of pclgo. Elapsed times were much shorter, because calculations were done in parallel. "Acceleration" is the ratio of execution speed in the deterministic model to that in the stochastic model. Models are arranged in order of increasing execution time with the deterministic algorithm.

	$\log_{10}$ seconds		
Model	Deterministic	Stochastic	Acceleration
α	1.60246	5.94980	22250.5
$\alpha\beta$	2.60384	5.81015	1608.1
$\alpha\gamma$	2.83806	5.94417	1276.8
$\alpha\beta\gamma$	3.67736	6.03901	230.0
$\alpha\delta$	4.42860	6.16730	54.8
$\alpha\beta\delta$	4.86285	6.04239	15.1
$\alpha\gamma\delta$	5.47544	6.14022	4.6
$\alpha\beta\gamma\delta$	6.04505	6.20171	1.4

Some bias is evident in these estimates. For example, the estimates of m_α tend to be a little low and those of m_ϵ a little high [20]. This reflects the negative correlation between these parameters that can be seen in Fig. 3. Because the two source populations (N and D) are so similar, they are hard to distinguish. We get a better estimate of the sum $(m_\alpha + m_\epsilon)$ than of the difference $(m_\alpha - m_\epsilon)$. ~~Nonetheless, the swarm of estimates tends~~ There is also some bias in $2N_D$ and $2N_N$. In spite of these biases, the swarms of estimates tend to enclose the true parameter ~~value~~values, so the ~~bias~~ biases in these estimates ~~is~~ are modest compared with their uncertainties. It should not, however, be assumed that this will always be the case. ~~One must check for bias by doing simulations of the sort illustrated here.~~

To illustrate the new algorithm in a full-scale analsis of real data, I replicated the analysis of Rogers et al. [22]. Table 2 shows the CPU time used by each algorithm in analysis of the eight models in that publication. For this set of models, the deterministic algorithm is always faster, but its execution time ranges across several orders of magnitude. These execution times are not strictly comparable, because they involve several compute clusters, which vary in processor speed. These differences are minor, however, compared with the enormous differences in run time seen in table 2.

To choose among models, I used the bootstrap estimate of predictive error, "bepe" [4, 5, 20]. This method uses variation among data sets (the real data plus 50 replicates generated by a moving-blocks bootstrap [14]) to approximate variation in repeated sampling. It fits the model to one data set and then tests this fit against all the others. Table 3 uses all models to compare the bepe values calculated by the deterministic and stochastic algorithm. In all cases, the deterministic algorithm yields a smaller bepe value than the stochastic algorithm, indicating a better fit of model to data. The order of the eight models, however, is the same. Because the deterministic algorithm yields smaller bepe values, one should use the same algorithm (stochastic or deterministic) for all models in any analysis. Otherwise, model selection will be biased in favor of deterministic results because of their smaller bepe values.

When several models provide reasonable descriptions of the data, it is better to average across models than to choose just one. This allows uncertainty about the model itself to be incorporated into confidence intervals. For this purpose, Legofit uses bootstrap model averaging, "booma" [2, 20]. The booma weight of the ith model is the fraction of data sets (including the real data and

Table 3: Bootstrap estimate of predictive error (bepe) values and bootstrap model average (booma) weights, based on the data of Rogers et al. [22]. Values for the stochastic algorithm are also from that publication. Models are arranged in order of decreasing bepe values.

	Deterministic		Stochastic	
Model	bepe	weight	bepe	weight
α	1.13×10^{-6}	0	1.16×10^{-6}	0
$\alpha\delta$	0.82×10^{-6}	0	0.87×10^{-6}	0
$\alpha\gamma$	0.61×10^{-6}	0	0.62×10^{-6}	0
$\alpha\gamma\delta$	0.40×10^{-6}	0	0.44×10^{-6}	0
$\alpha\beta$	0.14×10^{-6}	0	0.18×10^{-6}	0
$\alpha\beta\gamma$	0.14×10^{-6}	0	0.17×10^{-6}	0
$\alpha\beta\delta$	0.11×10^{-6}	0.02	0.15×10^{-6}	0.16
$\alpha\beta\gamma\delta$	0.10×10^{-6}	0.98	0.13×10^{-6}	0.84

50 bootstrap replicates) in which that model "wins," i.e. has the lowest value of bepe. The weights of all models are shown in table 3.

The new analysis, using the deterministic algorithm, replicates the main result of Rogers et al. [22]: that the most complex model ($\alpha\beta\gamma\delta$) is preferred over all others. The strength of this preference, however, is stronger under the deterministic algorithm. The 2nd-place model ($\alpha\beta\delta$) gets 16% of the weight with the stochastic algorithm but only 2% with the deterministic one. The greater precision of the deterministic algorithm apparently improves Legofit's ability to discriminate among models. The difference between these models is that $\alpha\beta\gamma\delta$ includes gene flow from early modern humans into Neanderthals, as proposed by Kuhlwilm et al. [12]. The current results strengthen the case for this hypothesis.

The model-averaged estimates of all parameters are shown in supplementary table S2. The two algorithms provide similar estimates, but there are two differences. First, the deterministic algorithm provides an unrealistic estimate of T_{XY}, the separation time of Europeans and Africans. This estimate—323 generations, or about 9000 y—is clearly too small. This may indicate that something is missing from the model or that identifiability problems have introduced bias. Further work would be needed to evaluate these alternatives. Second, the estimate of N_S is even larger—over 700,000—with the deterministic algorithm than with the stochastic one. This supports our previous suggestion that the superarchaic population was large or deeply subdivided [22].

4 Conclusions

Legofit's new deterministic algorithm ~~provides an enormous increase in~~ increases both speed and accuracy~~and makes the package practicable on desktop computers.~~. The increase in accuracy results in smaller residual errors and better discrimination between alternative hypotheses. It has no large effect on confidence intervals, however, because these are primarily measuring uncertainty arising from statistical identifiability problems. The increase in speed is dramatic with models of small to moderate complexity and makes Legofit practicable on laptop computers. The deterministic algorithm slows dramatically, however, as models increase in complexity. For very complex models, the stochastic algorithm is still needed.

The deterministic algorithm replicated all the findings of Rogers et al. [22]. Because of its greater accuracy, it provided stronger support for the hypothesis that early modern humans contributed genes to Neanderthals [12]. It also strengthened the evidence that the superarchaic population was

large or deeply subdivided [22].

Legofit is open source and freely available at `https://github.com/alanrogers/legofit`.

Acknowledgements

I thank Greg Martin for comments on appendix B. The Legofit package is freely available at . , Elizabeth Cashdan for editorial suggestions, and those who reviewed the manuscript for *PCI Mathematical and Computational Biology*. Analysis files are archived at `doi:10.17605/OSF.IO/74BJF`. This work was supported by NSF BCS 1638840, NSF BCS 1945782, and the Center for High Performance Computing at the University of Utah.

A The probability that d of n descendants derive from 1 of k ancestors

Eqn. 8 presents a formula for Q_{dk}, the probability that a particular set of d descendants, chosen from a total of n, derives from a single unspecified ancestor, given that there were k ancestors in that ancestral generation. If $k = 1$, $Q_{dk} = 1$ as explained above. The result for $k > 1$ can be derived in two different ways.

A.1 Short argument

Condition on the event that r of the k ancestors have Suppose that some ancestor has d descendantseach. The probability that a particular group of d descendents derives from one of these is $r/\binom{n}{d}$this ancestor is $1/\binom{n}{d}$, where $\binom{n}{d}$ is the number of ways of choosing d descendants from a total of n. The corresponding unconditional probability is $Q_{dk} = E[r]/\binom{n}{d}$. If r ancestors have d descendants each, the probability of descent from one of these is $r/\binom{n}{d}$. In reality, r is a random variable, and the probability becomes $Q_{dk} = E[r]/\binom{n}{d}$, where $E[r]$ is the expected value of r.

To derive $E[r]$, number the ancestors from 1 to k, and let y_i represent the number of descendants of the ith ancestor, where $y_i > 0$ and $\sum y_i = n$. I will refer to a particular set of values, $y_1, \ldots, y_k$, as an allocation of descendants among ancestors. The number of such allocations is $\binom{n-1}{k-1}$ [6, pp. 38–39]. Furthermore, each allocation has the same probability, $\binom{n-1}{k-1}^{-1}$, under the coalescent process [3, p. 13].

The k ancestors are statistically equivalent, which implies that $E[r] = \sum_{i=1}^{k} \Pr\{y_i = d\} = k \Pr\{y_i = d\}$ for an arbitrary ancestor i. If this ancestor has d descendants, there are $\binom{n-d-1}{k-2}$ ways, each with probability $\binom{n-1}{k-1}^{-1}$, to allocate the $n - d$ remaining descendants among the $k - 1$ remaining ancestors. Thus $\Pr\{y_i = d\} = \binom{n-d-1}{k-2}\binom{n-1}{k-1}^{-1}$, and Q_{dk} equals the expression in Eqn. 8.

A.2 Longer argument

The k ancestors define a partition of the set of descendants into k subsets, each corresponding to a different ancestor. Let $x_1, x_2, \ldots, x_k$ denote the sizes of the k subsets, i.e., the numbers of descendants of the k ancestors. The probability of such a partition is given above in Eqn. 7. Suppose that a set of d descendants (and no others) derive from a single ancestor in interval k. This can happen only if $x_i = d$ for some i. The ancestors are numbered in an arbitrary order, so

let us set $x_k = d$ and rewrite Eqn. 7 as

$$A = k!\binom{n-1}{k-1}^{-1}\binom{n}{d}^{-1}\binom{n-d}{x_1,\ldots,x_{k-1}}^{-1}$$

To calculate Q_{dk}, we need to sum this quantity across all ways to partition the set of $n-d$ remaining descendants into $k-1$ subsets.

This is not the same as summing across values of x_i, because each array of x_i values may correspond to numerous partitions of the set of descendants. This is illustrated in table 1, where the left side lists the 7 ways of partitioning a set of 4 descendants among 2 ancestors, along with the probability of each partition as given by Eqn. 7. The first four set partitions have equal probability, because each one divides the descendants into subsets of sizes 3 and 1, and the x_j values of these partitions therefore make equal contributions to Eqn. 7. Similarly, the last three set partitions have equal probability, because each divides the ancestors into two sets of size 2. These two cases: $3+1=4$ and $2+2=4$ are the two ways of expressing 4 as a sum of two positive integers. Eqn. 7 implies that all set partitions corresponding to a given integer partition have equal probability.

There are $\binom{n-d}{x_1,\ldots,x_{k-1}} \Big/ \prod_m c_m!$ set partitions for a given partition of the integer $n-d$ into $k-1$ summands [1, theorem 13.2, p. 215]. In this expression, c_m is the number of times m appears among $x_1,\ldots,x_{k-1}$. Multiplying this into A and summing gives

$$Q_{dk} = k!\binom{n-1}{k-1}^{-1}\binom{n}{d}^{-1}\sum\left(\prod_m c_m!\right)^{-1} \tag{9}$$

where the sum is over ways of partitioning $n-d$ into $k-1$ summands. Appendix B shows that this sum equals $\binom{n-d-1}{k-2}/(k-1)!$. Substituting into Eqn. 9 reproduces Eqn. 8.

B An identity involving integer partitions

The partition of a positive integer n into k parts can be written as $n = \sum_{i=1}^{k} x_i$, where the x_i are positive integers. ~~On the other hand, this~~ This same partition is also $n = \sum_i i c_i$, where c_i is the number of times i appears among the x_i values. In other words, c_i is the multiplicity of i in the partition. In terms of these multiplicities, $k = \sum c_i$. This appendix will show that

$$\sum\left(\prod_i c_i!\right)^{-1} = \frac{1}{k!}\binom{n-1}{k-1} \tag{10}$$

where the sum is across all partitions of an integer n into k parts.

This identity follows from the fact that there are $\binom{n-1}{k-1}$ ways to put n balls into k boxes so that no box is empty [6, pp. 38–39]. Let us call each of these an "allocation" of balls to boxes. For each allocation, there is a corresponding partition of the integer n into k parts. The number of allocations often larger than the number of partitions. For example, there are $\binom{2}{1} = 2$ ways to put 3 balls into 2 boxes, `**|*` and `*|**`, where the stars represent balls and the bar separates boxes. Both allocations, however, correspond to a single partition, $3 = 2+1$, of the integer 3. For a given integer partition, $c_1, c_2, \ldots$, there are $k!/\prod c_i!$ distinct ways to allocate balls to boxes. (This is the number of ways to reorder the boxes while ignoring the order of boxes with equal numbers of balls.) The sum of this quantity across partitions must therefore equal $\binom{n-1}{k-1}$. Dividing both sides by $k!$ produces identity 10. Greg Martin posted a different proof of this identity on StackExchange.[2]

[2] `https://math.stackexchange.com/questions/938280/on-multiplicity-representations-of-integer-partitions-of-fixed-length`

References

[1] George E. Andrews. *The Theory of Partitions*. Addison Wesley, Reading, MA, 1976.

[2] Steven T Buckland, Kenneth P Burnham, and Nicole H Augustin. Model selection: an integral part of inference. *Biometrics*, 53(2):603–618, 1997.

[3] Richard Durrett. *Probability Models for DNA Sequence Evolution*. Springer, New York, 2nd edition, 2008.

[4] Bradley Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[5] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall, New York, 1993.

[6] William Feller. *An Introduction to Probability Theory and Its Applications*, volume II. Wiley, New York, 2nd edition, 1971.

[7] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13–es, 2007. ISSN 0098-3500.

[8] RC Griffiths and Simon Tavaré. The age of a mutation in a general coalescent tree. *Stochastic Models*, 14(1-2):273–295, 1998.

[9] Jerome Kelleher, Alison M Etheridge, and Gilean McVean. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Computational Biology*, 12(5):1–22, 5 2016.

[10] Motoo Kimura. The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutation. *Genetics*, 61:893–903, 1969.

[11] Donald E. Knuth. *The Art of Computer Programming: Volume 4A, Combinatorial Algorithms. Part 1*. Addison-Wesley, New York, 2011. ISBN 0-201-03804-8.

[12] Martin Kuhlwilm, Ilan Gronau, Melissa J. Hubisz, Cesare de Filippo, Javier Prado-Martinez, Martin Kircher, Qiaomei Fu, Hernán A. Burbano, Carles Lalueza-Fox, Marco de la Rasilla, Antonio Rosas, Pavao Rudan, Dejana Brajkovic, Željko Kucan, Ivan Gušic, Tomas Marques-Bonet, Aida M. Andrés, Bence Viola, Svante Pääbo, Matthias Meyer, Adam Siepel, and Sergi Castellano. Ancient gene flow from early modern humans into Eastern Neanderthals. *Nature*, 530(7591):429–433, Feb 2016. ISSN 1476-4687.

[13] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, Mar 1951.

[14] Regina Y. Liu and Kesar Singh. Moving blocks jacknife and bootstrap capture weak dependence. In Raoul LePage and Lynne Billard, editors, *Exploring the "Limits" of the Bootstrap*, pages 225–248. Wiley, New York, 1992.

[15] Swapan Mallick, Heng Li, Mark Lipson, Iain Mathieson, Melissa Gymrek, Fernando Racimo, Mengyao Zhao, Niru Chennagiri, Susanne Nordenfelt, Arti Tandon, Pontus Skoglund, Iosif

Lazaridis, Sriram Sankararaman, Qiaomei Fu, Nadin Rohland, Gabriel Renaud, Yaniv Erlich, Thomas Willems, Carla Gallo, Jeffrey P. Spence, Yun S. Song, Giovanni Poletti, Francois Balloux, George van Driem, Peter de Knijff, Irene Gallego Romero, Aashish R. Jha, Doron M. Behar, Claudio M. Bravi, Cristian Capelli, Tor Hervig, Andres Moreno-Estrada, Olga L. Posukh, Elena Balanovska, Oleg Balanovsky, Sena Karachanak-Yankova, Hovhannes Sahakyan, Draga Toncheva, Levon Yepiskoposyan, Chris Tyler-Smith, Yali Xue, M. Syafiq Abdullah, Andres Ruiz-Linares, Cynthia M. Beall, Anna Di Rienzo, Choongwon Jeong, Elena B. Starikovskaya, Ene Metspalu, Jüri Parik, Richard Villems, Brenna M. Henn, Ugur Hodoglugil, Robert Mahley, Antti Sajantila, George Stamatoyannopoulos, Joseph T. S. Wee, Rita Khusainova, Elza Khusnutdinova, Sergey Litvinov, George Ayodo, David Comas, Michael F. Hammer, Toomas Kivisild, William Klitz, Cheryl A. Winkler, Damian Labuda, Michael Bamshad, Lynn B. Jorde, Sarah A. Tishkoff, W. Scott Watkins, Mait Metspalu, Stanislav Dryomov, Rem Sukernik, Lalji Singh, Kumarasamy Thangaraj, Svante Pääbo, Janet Kelso, Nick Patterson, and David Reich. The Simons Genome Diversity Project: 300 genomes from 142 diverse populations. *Nature*, 538:201–206, 2016. ISSN 1476-4687.

[16] Matthias Meyer, Martin Kircher, Marie-Theres Gansauge, Heng Li, Fernando Racimo, Swapan Mallick, Joshua G Schraiber, Flora Jay, Kay Prüfer, Cesare de Filippo, Peter H. Sudmant, Can Alkan, Qiaomei Fu, Ron Do, Nadin Rohland, Arti Tandon, Michael Siebauer, Richard E. Green, Katarzyna Bryc, Adrian W. Briggs, Udo Stenzel, Jesse Dabney, Jay Shendure, Jacob Kitzman, Michael F. Hammer, Michael V. Shunkov, Anatoli P. Derevianko, Nick Patterson, Aida M. Andrés, Evan E. Eichler, Montgomery Slatkin, David Reich, Janet Kelso, and Svante Pääbo. A high-coverage genome sequence from an archaic Denisovan individual. *Science*, 338 (6104):222–226, 2012.

[17] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer Science and Business Media, Berlin, 2006.

[18] Kay Prüfer, Cesare de Filippo, Steffi Grote, Fabrizio Mafessoni, Petra Korlević, Mateja Hajdinjak, Benjamin Vernot, Laurits Skov, Pinghsun Hsieh, Stéphane Peyrégne, David Reher, Charlotte Hopfe, Sarah Nagel, Tomislav Maricic, Qiaomei Fu, Christoph Theunert, Rebekah Rogers, Pontus Skoglund, Manjusha Chintalapati, Michael Dannemann, Bradley J. Nelson, Felix M. Key, Pavao Rudan, Željko Kućan, Ivan Gušić, Liubov V. Golovanova, Vladimir B. Doronichev, Nick Patterson, David Reich, Evan E. Eichler, Montgomery Slatkin, Mikkel H. Schierup, Aida Andrés, Janet Kelso, Matthias Meyer, and Svante Pääbo. A high-coverage Neandertal genome from Vindija Cave in Croatia. *Science*, 358(6363):655–658, 2017.

[19] Kay Prüfer, Fernando Racimo, Nick Patterson, Flora Jay, Sriram Sankararaman, Susanna Sawyer, Anja Heinze, Gabriel Renaud, Peter H Sudmant, Cesare de Filippo, Heng Li, Swapan Mallick, Michael Dannemann, Qiaomei Fu, Martin Kircher, Martin Kuhlwilm, Michael Lachmann, Matthias Meyer, Matthias Ongyerth, Michael Siebauer, Christoph Theunert, Arti Tandon, Priya Moorjani, Joseph Pickrell, James C. Mullikin, Samuel H. Vohr, Richard E. Green, Ines Hellmann, Philip L. F. Johnson, Hélène Blanche, Howard Cann, Jacob O. Kitzman, Jay Shendure, Evan E. Eichler, Ed S. Lein, Trygve E. Bakken, Liubov V. Golovanova, Vladimir B. Doronichev, Michael V. Shunkov, Anatoli P. Derevianko, Bence Viola, Montgomery Slatkin, David Reich, Janet Kelso, and Svante Pääbo. The complete genome sequence of a Neanderthal from the Altai Mountains. *Nature*, 505(7481):43–49, 2014.

[20] Alan R. Rogers. Legofit: Estimating population history from genetic data. *BMC Bioinformatics*, 20:526, 2019.

[21] Alan R. Rogers, Ryan J. Bohlender, and Chad D. Huff. Early history of Neanderthals and Denisovans. *Proceedings of the National Academy of Sciences, USA*, 114(37):9859–9863, 2017.

[22] Alan R. Rogers, Nathan S. Harris, and Alan A. Achenbach. Neanderthal-Denisovan ancestors interbred with a distantly-related hominin. *Science Advances*, 6(8):eaay5483, 2020.

[23] Simon Tavaré. Line-of-descent and genealogical processes, and their applications in population genetics models. *Theoretical Population Biology*, 26:119–164, 1984.

[24] P. J. Waddell. Happy New Year *Homo erectus*? More Evidence for Interbreeding with Archaics Predating the Modern Human/Neanderthal Split. *ArXiv*, 1312.7749, December 2013.

[25] Peter J Waddell, Jorge Ramos, and Xi Tan. *Homo denisova*, correspondence spectral analysis, finite sites reticulate hierarchical coalescent models and the Ron Jeremy hypothesis. *ArXiv*, 1112.6424, 2011.

[26] Stephen Wooding and Alan R. Rogers. The matrix coalescent and an application to human SNPs. *Genetics*, 161:1641–1650, 2002.

Supplementary Materials*

Alan R. Rogers

May 3, 2021

S1 Simulations

In the archive, simulation code and simulated data sets are in directory `ae/sim`. To simulate 50 data sets using msprime [1], I executed the following bash command:

```
seq 0 49 | xargs -n 1 bash sim.sh
```

This invokes the shell script `sim.sh` 50 times. This shell script looks like this:

```
ofile=sim${1}.opf
efile=sim${1}.err
python3 msp.py -r | simpat 1>${ofile} 2>${efile}
```

Here, `simpat` is part of the Legofit package, and `msp.py` is a Python script that runs msprime. It is listed in appendix SA.1 on p. 3 and defines the model and all parameter values. This command generates 50 output files with names like `sim0.opf`, `sim1.opf`, and so on.

S2 ~~Data analysis~~Analysis of simulated data

S2.1 Deterministic algorithm

Analysis files using the deterministic algorithm can found in directory `ae/exact` within the archive. The pipeline in that directory is designed for use on a compute cluster running `slurm`, and the details of this pipeline can be found within that directory in the files `README.md`, `pipeline.sh`, `a1.slr` (appendix SB.2, p. 9), `a2.slr` (appendix SB.3, p. 10), `pclgo.slr` (appendix SB.4, p. 11), `b1.slr` (appendix SB.5, p. 11), and `b2.slr` (appendix SB.6, p. 12). However, this analysis is fast enough to run on an ordinary desktop computer, so I will describe the underlying Legofit commands here without reference to commands involving the cluster.

The analysis proceeds in several stages. Stages 1 and 2 use file `a.lgo`, which describes the simulation model accurately, except that the parameter values are perturbed away from their true values. Stage 1 executes the following command for data set 0:

```
legofit -1 -d 0 --stateOut a1-0.state --tol 3e-6 -S 5000 a.lgo \
  ../sim/sim0.opf > a1-0.legofit
```

*This supplement accompanies "An Efficient Algorithm for Estimating Population History from Genetic Data"

Similar commands are executed for each of the other 49 data sets. Here `a.lgo` is the input file describing the model of population history, `../sim/sim0.opf` is the simulated data set to be analyzed, and `a1-0.state` is the name of an output file that will record the optimizer's final state. The argument "`-1`" says not to ignore singleton site patterns, "`-d 0`" says to use the deterministic algorithm without ignoring states with low probability, "`--tol 3e-6`" tells the optimizer to stop when the spread of objective function values falls to 3×10^{-6}, and "`-S 5000`" tells the optimizer to do at most 5000 iterations. Legofit writes to standard output, which is redirected into `a1-0.legofit`.

Legofit uses the *differential evolution* optimization algorithm [2], which is quite good at finding global optima. Nonetheless, it is still possible that some of the 50 jobs in stage 1 will have gotten stuck on different local optima. To deal with this problem, each job in stage 2 of the analysis begins by reading the 50 `.state` files produced in stage 1, and generating an initial swarm of points that includes points from all 50 of the stage 1 jobs. Each stage 2 job is then able to choose among the optima found in stage 1. The legofit command for data set 0 in stage 2 looks like this:

```
legofit -1 -d 0 --tol 3e-6 -S 5000 a.lgo ../sim/sim0.opf \
  --stateIn a1-0.state \
  --stateIn a1-1.state \
  ...
  --stateIn a1-49.state > a2-0.legofit
```

This is just like the stage 1 command, except that there is no `--stateOut` command, and there are 50 `--stateIn` commands. Stage 2 generates 50 output files with names like `a2-0.legofit`.

Stages 1 and 2 present the optimizer with a challenge, because several of the free variables are tightly correlated, as shown in Fig. 3 of the main text. To alleviate this problem, stage 3 re-expresses the free variables in terms of principal components. To do this, `pclgo` reads `a.lgo` along with all 50 of the `.legofit` files produced in stage 3, and writes a single output file called `b.lgo`. The command that does this is

```
# (grep ^# a.lgo; pclgo a.lgo a2-*.legofit; grep -v ^# a.lgo |
 egrep -v "\<free\>") > b.lgo
```

This prints into `b.lgo` (1) the comments from `a.lgo`, (2) the free variables redefined as principal components, and (3) the rest of `a.lgo`. In previous publications, we have used the `--tol` argument of `pclgo` in order to reduce the number of dimensions. In the current analysis, we omit that argument, so there is no reduction in dimension. It is possible that reducing dimension can introduce bias, especially when there are identifiability problems in the data. Even without any reduction in dimension, `pclgo` makes the optimizer's job easier by re-expressing in orthogonal dimensions.

Stages 4 and 5 are just like stages 1 and 2, except that they use `b.lgo` instead of `a.lgo`.

S2.2 Stochastic algorithm

The stochastic algorithm is much slower and is best run on a compute cluster. The full pipeline, including slurm scripts, is available in the archive, but I will ignore cluster-related details here. Directory `ae/stoch` contains the code for and results from analysis under the stochastic algorithm, which follows the same steps as the deterministic one. The command for stage 1 looks like

```
legofit -1 --stateOut a1-0.state --tol 3e-5 \
  -S 5000@10000 -S 100@100000 -S 1000@2000000 a.lgo ../sim/sim0.opf \
  > a1-0.legofit
```

Note that the `-d 0` argument has been omitted. This tells `legofit` to use the default stochastic algorithm. I'm using a looser tolerance (`--tol 3e-5` rather than `--tol 3e-6`), because the stochastic algorithm can't achieve the same accuracy. Finally, the `-S` arguments are different. The first `-S` argument tells `legofit` to begin with 5000 generations of differential evolution, in each of which the objective function is evaluated using 10000 iterations of coalescent simulation. The next `-S` argument says to do 100 differential evolution generations with 100,000 coalescent iterations. The final one specifies 1000 differential evolution generations with 2,000,000 coalescent iterations. The optimizer will stop before these iterations are complete if it reaches the goal specified with `--tol`.

Stage 2 of the stochastic algorithm is similar to stage 2 of the deterministic one, except that it does not use the `-d 0` argument, and it specifies: `--tol 2e-5 -S 1000@2000000`. The tolerance is somewhat tighter than in stage 1, and there is only one `-S` argument.

Stage 3 of the stochastic algorithm is just like stage 3 of the deterministic algorithm. Stages 4 and 5 of the stochastic algorithm are just like stages 1 and 2, except that `b.lgo` is used instead of `a.lgo`.

S3 Replication of results from Rogers et al. [4] using the deterministic algorithm

To illustrate the new deterministic algorithm in a realistic context, I reanalyzed the eight models considered by Rogers et al. [4], using the same data set, and the same bootstrap replicates. This analysis is described in section 2.8 of the main text. This supplement provides additional detail.

Each model is defined using an input file in .lgo format. The one for model $\alpha\beta\gamma\delta$ is in appendix SC.1, page 13. The 22 parameters defined there are listed in table S1.

The legofit program was run using the `-d 0` option, which invokes the deterministic algorithm, with a tolerance (`--tol 3e-6`) of 3×10^{-6}. This tolerance value is much smaller than that used by Rogers et al. [4]. It tells the estimator to attempt to fit the model to very high precision. One of the slurm scripts used in this analysis is shown in appendix SC.2 on page 15.

I used the bootstrap estimate of predictive error (bepe) for model selection and bootstrap model averaging (booma) for model averaging [3]. The model-averaged estimates of all parameters are shown in table S2.

SA Simulations

SA.1 msp.py

```
#!/usr/bin/python3
import msprime
import os, sys, time

def usage():
    print("Usage: ./msp.py [options]")
    print("  where options may include:")
    print("  -r or --run: run simulation. Default: run")
    print("               DemographyDebugger")
    sys.exit(1)
```

Table S1: Parameters used in reanalysis of data from Rogers et al. [4]. The "text" column gives the symbol used in the text of this document; the "code" column gives the symbol used in input file `a.lgo` (appendix SC.1). Population sizes are in "haploid" units, which refer to twice the number of diploid individuals.

Symbol		
text	code	Description
T_{XYND}	`Txynd`	separation time of XY and ND
T_{XYNDS}	`Txynds`	separation time of S
T_{ND}	`Tnd`	separation time of N and D
T_{N_0}	`Tav`	time of change in Neanderthal population size
T_{XY}	`Txy`	separation time of X and Y
T_D	`Td`	age of Denisovan fossil
T_A	`Ta`	age of Altai Neanderthal fossil
T_V	`Tv`	age of Vindija Neanderthal fossil
T_α	`TmN`	α admixture time
T_β	`TmS`	β admixture time
T_γ	`TmXY`	γ admixture time
T_δ	`TmSND`	δ admixture time
$2N_{N_0}$	`twoNav`	size of early Neanderthal population
$2N_{N_1}$	`twoNn`	size of late Neanderthal population
$2N_{ND}$	`twoNnd`	size of population ND
$2N_{XY}$	`twoNxy`	size of population XY
$2N_{XYND}$	`twoNxynd`	size of population $XYND$
$2N_S$	`twoNs`	size of population S
m_α	`mN`	α admixture fraction
m_β	`mS`	β admixture fraction
m_γ	`mXY`	γ admixture fraction
m_δ	`mSND`	δ admixture fraction

Table S2: Parameter estimates and confidence intervals using real data and both algorithms. Results for stochastic algorithm are from Rogers et al. [4].

Parameter	Deterministic		Stochastic	
	Estimate	95% C.I.	Estimate	95% C.I.
m_α	0.019	(0.017, 0.021)	0.019	(0.018, 0.021)
m_β	0.023	(0.021, 0.030)	0.021	(0.017, 0.027)
m_γ	0.009	(0.008, 0.013)	0.016	(0.013, 0.025)
m_δ	0.035	(0.034, 0.052)	0.034	(0.022, 0.043)
T_D	3485	(3394, 3729)	3480	(3242, 3665)
T_V	2471	(2357, 2628)	2506	(2317, 2656)
T_A	5295	(5208, 5452)	5310	(5078, 5415)
T_{N_0}	10644	(10656, 12535)	15706	(14594, 16956)
T_{ND}	25135	(25055, 25619)	25400	(25135, 25526)
T_{XY}	323	(62, 506)	1262	(548, 2611)
T_{XYNDS}	78036	(69103, 77977)	79334	(71901, 89765)
$2N_{N_1}$	5509	(5532, 6278)	6745	(6444, 6994)
$2N_{N_0}$	15915	(15192, 18389)	31411	(24767, 41586)
$2N_{ND}$	1618	(597, 1756)	1081	(797, 1680)
$2N_{XY}$	58955	(58716, 60979)	55818	(51085, 58416)
$2N_{XYND}$	40500	(40214, 41453)	40831	(39917, 41264)
$2N_S$	1465235	(557565, 2336867)	53995	(40181, 104336)

```
do_simulation = False
for arg in sys.argv[1:]:
    if arg == "-r" or arg == "--run":
        do_simulation = True
    else:
        usage()

# time parameters in generations
Txynd = 25920
Tnd = 15000
Txy = 3788
Td = 1734      # age of Denisova fossil
Ta = 1760      # age of Altai fossil
Talpha = 1897     # time of Neanderthal admixture
Tepsilon = Td-1     # time of Denisovan admixture

# population sizes
Nxynd = 64964.1/2.0  # ancestral population
Nxy = 44869.2/2.0
Nnd = 5000/2.0
Nn = 9756.8/2.0
Nd = 5000/2.0
Nx = 20000/2.0 # modern Africa
```

```
Ny = 20000/2.0 # modern Europe

# admixture
mAlpha = 0.05
mEpsilon = 0.025

nchromosomes = 1000      # number of chromosomes
basepairs = 2e6   # number of nucleotides per chromosome
u_per_site = 1.4e-8 # mutation

# Recombination rate.  recomb is the probability of recombination
# between sites at opposite ends of the simulated sequence.
c = 1e-8 # rate per base pair per generation

# One haploid sample from each of 4 populations: two modern (X,Y),
# and two archaic (N,D).
samples = [
    msprime.Sample(population=0, time=0),  # population X
    msprime.Sample(population=1, time=0),  # population Y
    msprime.Sample(population=2, time=Ta), # population N
    msprime.Sample(population=3, time=Td)  # population D
]

lbl = ("x", "y", "n", "d")
npops = len(lbl)

# associate sample sizes with population labels
sampsize = {}
for s in lbl:
    sampsize[s] = 0

for samp in samples:
    ndx = samp.population
    assert ndx < len(lbl)
    assert lbl[ndx] in sampsize
    sampsize[lbl[ndx]] = 1

for s in lbl:
    if not (sampsize[s] > 0):
        print("Population %s has no samples" % s, file=sys.stderr)
        sys.exit(1)

# Population configurations. No sample sizes are listed here, because
# those are specified above in "samples".
popconf = [
    msprime.PopulationConfiguration(initial_size=Nx),
    msprime.PopulationConfiguration(initial_size=Ny),
    msprime.PopulationConfiguration(initial_size=Nn),
```

```
    msprime.PopulationConfiguration(initial_size=Nd)
]

events = [
    msprime.MassMigration(
        time=Tepsilon,
        source=1,
        dest=3,
        proportion=mEpsilon), # D->Y gene flow
    msprime.MassMigration(
        time=Talpha,
        source=1,
        dest=2,
        proportion=mAlpha), # N->Y gene flow
    msprime.MassMigration(
        time=Txy,
        source=1,
        dest=0,
        proportion=1.0), # X-Y split
    msprime.PopulationParametersChange(
        time=Txy,
        initial_size=Nxy,
        population_id=0),
    msprime.MassMigration(
        time=Tnd,
        source=3,
        dest=2,
        proportion=1.0), # N-D split
    msprime.PopulationParametersChange(
        time=Tnd,
        initial_size=Nnd,
        population_id=2),
    msprime.MassMigration(
        time=Txynd,
        source=2,
        dest=0,
        proportion=1.0), # XY-ND split
    msprime.PopulationParametersChange(
        time=Txynd,
        initial_size=Nxynd,
        population_id=0)
]

if do_simulation:
    # run simulation
    seed = int(time.time()) ^ os.getpid()
    sim = msprime.simulate(samples = samples,
                           population_configurations = popconf,
```

```
                                demographic_events = events,
                                length = basepairs,
                                recombination_rate = c,
                                mutation_rate = u_per_site,
                                num_replicates = nchromosomes,
                                random_seed = seed)

    # header
    print("npops = %d" % len(lbl))
    print("%s %s" % ("pop", "sampsize"))
    for s in lbl:
        print("%s %d" % (s, sampsize[s]))

    for i, chromosome in enumerate(sim):
        for variant in chromosome.variants():
            print(i, end=" ")
            for g in variant.genotypes:
                print(g, end=" ")
            print()

else:
    # Run demography debugger and quit
    dd = msprime.DemographyDebugger(
        population_configurations=popconf,
        demographic_events=events)
    dd.print_history()
    print("Use \"./sim -r\" to run simulation")
```

SB ~~Data analysis~~ Analysis of simulated data using the deterministic algorithm

SB.1 Model of history: a.lgo

```
# Model: ((X,Y), (N,D)), migration: N -> Y, D -> Y
# This file is like true.lgo but the free parameter values have been
# arbitrarily moved to incorrect values so that legofit will have some
# work to do.
time fixed   zero  = 0
twoN fixed   one   = 1
time fixed   Txynd = 25920   # \citet[table~S12.2, p.~90]{Li:N-505-43-S88}
time free      Tnd = 20000
time free      Txy = 2500
time fixed     Talpha = 1897    # \citep[table~2]{Sankararaman:PLO-8-e1002947}
time fixed     Tepsilon = 1       # arbitrary
time free      Td  = 1200
time free      Ta  = 1100
```

```
twoN free   twoNnd = 800
twoN free    twoNn = 1000
twoN free    twoNd = 1000
twoN free   twoNxy = 20000
twoN free twoNxynd = 20000
mixFrac free    mAlpha = 0.01
mixFrac free    mEpsilon = 0.01
segment x    t=zero   twoN=one     samples=1
segment y    t=zero   twoN=one     samples=1
segment n    t=Ta     twoN=twoNn   samples=1
segment n2   t=Talpha    twoN=twoNn
segment d0   t=Tepsilon    twoN=one
segment d    t=Td     twoN=twoNd   samples=1
segment y1   t=Tepsilon    twoN=one
segment y2   t=Talpha    twoN=one
segment nd   t=Tnd    twoN=twoNnd
segment xy   t=Txy    twoN=twoNxy
segment xynd t=Txynd  twoN=twoNxynd
mix    y  from y1 + mEpsilon * d0
mix    y1 from y2 + mAlpha * n2
derive x  from xy
derive y2 from xy
derive n  from n2
derive n2 from nd
derive d0 from d
derive d  from nd
derive xy from xynd
derive nd from xynd
```

SB.2 Deterministic a1.slr

```
#!/bin/bash
#SBATCH -J a1
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
#SBATCH --time=0:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o a1-%a.legofit # stdout
#SBATCH -e a1-%a.err # stderr

i=${SLURM_ARRAY_TASK_ID}
ifile=‘printf "../sim/sim%d.opf" $i‘    # input file
stateout=‘printf "a1-%d.state" $i‘

time legofit -1 -d 0 --stateOut ${stateout} --tol 3e-6 -S 5000 a.lgo ${ifile}
```

SB.3 Deterministic a2.slr

```
#!/bin/bash
#SBATCH -J a2
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
#SBATCH --time=0:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o a2-%a.legofit # stdout
#SBATCH -e a2-%a.err # stderr

i=${SLURM_ARRAY_TASK_ID}
ifile=‘printf "../sim/sim%d.opf" $i‘    # input file

time legofit -1 -d 0 --tol 3e-6 -S 5000 a.lgo ${ifile} \
  --stateIn a1-0.state \
  --stateIn a1-1.state \
  --stateIn a1-10.state \
  --stateIn a1-11.state \
  --stateIn a1-12.state \
  --stateIn a1-13.state \
  --stateIn a1-14.state \
  --stateIn a1-15.state \
  --stateIn a1-16.state \
  --stateIn a1-17.state \
  --stateIn a1-18.state \
  --stateIn a1-19.state \
  --stateIn a1-2.state \
  --stateIn a1-20.state \
  --stateIn a1-21.state \
  --stateIn a1-22.state \
  --stateIn a1-23.state \
  --stateIn a1-24.state \
  --stateIn a1-25.state \
  --stateIn a1-26.state \
  --stateIn a1-27.state \
  --stateIn a1-28.state \
  --stateIn a1-29.state \
  --stateIn a1-3.state \
  --stateIn a1-30.state \
  --stateIn a1-31.state \
  --stateIn a1-32.state \
  --stateIn a1-33.state \
  --stateIn a1-34.state \
  --stateIn a1-35.state \
  --stateIn a1-36.state \
  --stateIn a1-37.state \
```

```
  --stateIn a1-38.state \
  --stateIn a1-39.state \
  --stateIn a1-4.state \
  --stateIn a1-40.state \
  --stateIn a1-41.state \
  --stateIn a1-42.state \
  --stateIn a1-43.state \
  --stateIn a1-44.state \
  --stateIn a1-45.state \
  --stateIn a1-46.state \
  --stateIn a1-47.state \
  --stateIn a1-48.state \
  --stateIn a1-49.state \
  --stateIn a1-5.state \
  --stateIn a1-6.state \
  --stateIn a1-7.state \
  --stateIn a1-8.state \
  --stateIn a1-9.state
```

SB.4 pclgo.slr

```
#!/bin/bash
#SBATCH -J pclgo
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
#SBATCH --time=00:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -e pclgo.err # stderr

# Make b.lgo.
cmd="(grep ^# a.lgo;"
cmd+=" pclgo a.lgo a2-*.legofit;"
cmd+=" grep -v ^# a.lgo | egrep -v \"\<free\>\")"
echo "# "$cmd > b.lgo
eval $cmd >> b.lgo
```

SB.5 Deterministic b1.slr

```
#!/bin/bash
#SBATCH -J b1
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
#SBATCH --time=0:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o b1-%a.legofit # stdout
#SBATCH -e b1-%a.err # stderr
```

```
i=${SLURM_ARRAY_TASK_ID}
ifile=‘printf "../sim/sim%d.opf" $i‘    # input file
stateout=‘printf "b1-%d.state" $i‘

time legofit -1 -d 0 --stateOut ${stateout} --tol 3e-6 -S 5000 b.lgo ${ifile}
```

SB.6 Deterministic b2.slr

```
#!/bin/bash
#SBATCH -J b2
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
#SBATCH --time=0:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o b2-%a.legofit # stdout
#SBATCH -e b2-%a.err # stderr

i=${SLURM_ARRAY_TASK_ID}
ifile=‘printf "../sim/sim%d.opf" $i‘    # input file

time legofit -1 -d 0 --tol 3e-6 -S 5000 b.lgo ${ifile} \
  --stateIn b1-0.state \
  --stateIn b1-1.state \
  --stateIn b1-10.state \
  --stateIn b1-11.state \
  --stateIn b1-12.state \
  --stateIn b1-13.state \
  --stateIn b1-14.state \
  --stateIn b1-15.state \
  --stateIn b1-16.state \
  --stateIn b1-17.state \
  --stateIn b1-18.state \
  --stateIn b1-19.state \
  --stateIn b1-2.state \
  --stateIn b1-20.state \
  --stateIn b1-21.state \
  --stateIn b1-22.state \
  --stateIn b1-23.state \
  --stateIn b1-24.state \
  --stateIn b1-25.state \
  --stateIn b1-26.state \
  --stateIn b1-27.state \
  --stateIn b1-28.state \
  --stateIn b1-29.state \
  --stateIn b1-3.state \
  --stateIn b1-30.state \
```

```
  --stateIn b1-31.state \
  --stateIn b1-32.state \
  --stateIn b1-33.state \
  --stateIn b1-34.state \
  --stateIn b1-35.state \
  --stateIn b1-36.state \
  --stateIn b1-37.state \
  --stateIn b1-38.state \
  --stateIn b1-39.state \
  --stateIn b1-4.state \
  --stateIn b1-40.state \
  --stateIn b1-41.state \
  --stateIn b1-42.state \
  --stateIn b1-43.state \
  --stateIn b1-44.state \
  --stateIn b1-45.state \
  --stateIn b1-46.state \
  --stateIn b1-47.state \
  --stateIn b1-48.state \
  --stateIn b1-49.state \
  --stateIn b1-5.state \
  --stateIn b1-6.state \
  --stateIn b1-7.state \
  --stateIn b1-8.state \
  --stateIn b1-9.state
```

SC Analysis of real data using the deterministic algorithm

SC.1 Model $\alpha\beta\gamma\delta$: a.lgo input file

```
# Gene flow: N->Y, S->D, XY->N, S->ND
# S = superarchaic; XY = early modern; ND = Neandersovan; Y = Europe;
# A, Altai; V=Vindija; D=Denisovan. Altai and Vindija on same branch
# TmN < Tv < Ta
time fixed  zero = 0
twoN fixed  one = 1
time fixed  TmN = 1        # no coalesc. events can happen b/t 0 and Tv
time fixed  Txynd = 25920  # \citet[table~S12.2, p.~90]{Li:N-505-43-S88}
time free   Txynds = 60000 # arbitrary: 1.74 myr
time free   Tnd = 24000
time free   Tav = 14000
time free   Txy = 10000
time free   Td  = 2500
time free   Ta  = 4000
time free   Tv  = 1000
twoN free   twoNav = 2000
twoN free   twoNn = 2000
twoN free   twoNnd = 2000
```

```
twoN free   twoNxy = 20000
twoN free   twoNxynd = 20000
twoN free   twoNs = 2000
mixFrac free   mN = 0.01
mixFrac free   mS = 0.01
mixFrac free   mSND = 0.01
mixFrac free   mXY = 0.01
time constrained TmXY = 0.5*(Txy + Tav)
time constrained TmS = 0.5*(Td + Tnd)
time constrained TmSND = 0.5*(Tnd + Txynd)
segment x    t=zero   twoN=one     samples=1
segment y    t=zero   twoN=one     samples=1
segment v    t=Tv     twoN=twoNn   samples=1
segment a    t=Ta     twoN=twoNn   samples=1
segment a2   t=TmXY   twoN=twoNn
segment d    t=Td     twoN=one     samples=1
segment d2   t=TmS    twoN=one
segment y2   t=TmN    twoN=one
segment n    t=TmN    twoN=twoNn
segment s    t=TmS    twoN=twoNs
segment s2   t=TmSND  twoN=twoNs
segment av   t=Tav    twoN=twoNav
segment nd   t=Tnd    twoN=twoNnd
segment nd2  t=TmSND  twoN=twoNnd
segment xy   t=Txy    twoN=twoNxy
segment xy2  t=TmXY   twoN=twoNxy
segment xynd t=Txynd  twoN=twoNxynd
segment xynds t=Txynds twoN=twoNxynd
mix    d    from d2 + mS * s
mix    y    from y2 + mN * n
mix    a    from a2 + mXY * xy2
mix    nd   from nd2 + mSND * s2
derive n    from v
derive v    from a
derive a2   from av
derive av   from nd
derive d2   from nd
derive x    from xy
derive y2   from xy
derive xy   from xy2
derive xy2  from xynd
derive nd2  from xynd
derive s    from s2
derive s2   from xynds
derive xynd from xynds
```

SC.2 a1boot.slr

```
#!/bin/bash
#SBATCH -J ABCDa1boot
#SBATCH --account=rogersa-kp
#SBATCH --partition=rogersa-kp
#SBATCH --time=36:00:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o a1boot%a.legofit # stdout
#SBATCH -e a1boot%a.err # stderr

i=${SLURM_ARRAY_TASK_ID}
ifile=$(printf "../boot/boot%d.opf" $i)    # input file
stateout=$(printf "a1boot%d.state" $i)
lgofile=a.lgo

time legofit -1 -d 0 --stateOut $stateout --tol 3e-6 -S 5000 $lgofile $ifile
```

References

[1] Jerome Kelleher, Alison M Etheridge, and Gilean McVean. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Computational Biology*, 12(5):1–22, 5 2016.

[2] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer Science and Business Media, Berlin, 2006.

[3] Alan R. Rogers. Legofit: Estimating population history from genetic data. *BMC Bioinformatics*, 20:526, 2019.

[4] Alan R. Rogers, Nathan S. Harris, and Alan A. Achenbach. Neanderthal-Denisovan ancestors interbred with a distantly-related hominin. *Science Advances*, 6(8):eaay5483, 2020.