ACG Users Guide

v. 1.0

July 2011

Brendan O'Fallon
Dept. of Genome Sciences
University of Washington

Contents

What *is* ACG?

ACG is a tool to infer population parameters and genealogical structure from genetic data. It estimates the shape of a likelihood surface (such as a Bayesian posterior) of a model given some data, and thus is a close relative of "genealogy samplers" such as Mr Bayes, BEAST, and LAMARC. In a typical analysis the model may include population size, recombination rate, the model of nucleotide evolution, as well as the ancestry of the samples described as an Ancestral Recombination Graph (ARG). The probability distribution of these parameters are estimated using the Markov chain Monte Carlo (MCMC) technique first described by Yang & Rannala (1997), a technique for drawing individual samples of the parameters from the desired distribution.

ACG is written quite generally, and the user must pick which parameters, likelihoods, proposal kernels, and output options to use. Because of this freedom, ACG is not explicitly Bayesian, and priors need not be specified for every parameter. At this point it's up to the user to make sure that the analysis is constructed in a way that makes sense.

The primary design goal of ACG is to create a genealogy sampler that works with recombining sequences but had the same efficiency and simplicity as tree-only samplers. For several reasons samplers with recombination are significantly more complex than samplers without recombination, but our hope is to hide this as much as possible and provide an efficient, speedy search of ARG space.

A secondary goal of ACG is to provide a very modular and flexible environment to run MCMC analyses. To that end we've made it easy to install extensions developed by other folks. Just drop a .jar or .class file in the plugins folder and you'll be able to use tools that others have created. Similarly, the graphical interface has been designed to accommodate a wide variety of new models without needing significant changes.
\

Downloading and Installing

To download ACG, visit http://staff.washington.edu/brendano/acg , navigate to the "Downloads" page, and find the version (Mac, Linux, or PC) that matches your platform. After the file has downloaded, extract the contents either by double-clicking on the icon or right clicking and choosing "Extract...". After that, just click on the *acg* (or *acg.jar*) application to begin. You don't need to configure, "make", or install anything. On Unix-like systems (Mac & Linux) you can start ACG from the command line by typing:

```
$ java -jar acg.jar
```

To run an input file called myInputFile.xml from the command line, just type

```
$ java -jar acg.jar myInputFile.xml
```

Quick start in 3 steps

1. Create an input file

To use ACG, you'll first need to create an input file that contains your data and defines your analysis. The easiest way to do this is to take an existing input file and modify it by adding your own data. Several example input files that cover the most common cases can be found in the doc/ directory. The example input files have names like `example1.xml`. They're just text files, and comments within each one describe what the various parts are and what types of analyses are performed.

To add your own data to one of these files you'll first need to remove the existing sequences from the alignment block, which looks something like:

```
<alignment class=”sequence.Alignment”>
      <sequences class=”list”>
            <seqOne class=”sequence.Sequence”>
                  ATATCGACGACTACGACGACTACGAGACT.....
            </seqOne>
            <seqTwo class=”sequence.Sequence”>
                  ATATCGACGACTACGACGACTACGAGACT.....
            </seqTwo>
            ….many more...
      </sequences>
</alignment>
```

Simply erase everything between the first <sequences class=”list”> tag and the closing </sequences> tag and add in your own sequence. The label for each sequence is the name of the element you're creating, for instance the following element:

```
<myCrazySequence class=”sequence.Sequence”>
      AATACGACTACGACGACGCAT
</myCrazySequence>
```

creates a new sequence with the name “myCrazySequence”.
Another (perhaps easier) way to use your own data is to set the alignment block to refer to an external (fasta- or phylip-formatted) sequence file. The format for this is pretty easy, it looks like:

```
<alignment class=”sequence.Alignment” filename=”mysequences.fas” />
```

The above block will work as long as mysequences.fas is in the same directory that you're executing acg from. If it's in a different directory, you'll need to specify the path to the file, like this:

```
<alignment class=”sequence.Alignment” filename=”/path/to/mysequences.fas” />
```

Tip: If you have a fasta or phylip-formatted file of your sequences you can quickly convert the sequences into an acg-style XML block using the argutils.jar tool. The following command:

```
$ java -jar argutils.jar --convert mySequences.fas
```

will produce an alignment block that you can copy and paste right into an input file.

2. Begin the run

There are two ways to run your input file. First, you can run the input file using the graphical interface (GUI). To do this, just double-click on the ACG icon in the main ACG directory. A window will appear asking you to choose your input file. Once you've done this, click on either the "Run with GUI" or "Run in Background" button to begin the MCMC run. If you choose "Run With GUI" you'll be prompted to choose some parameters / likelihoods to watch as the chain runs. This will have no effect on the results, but in some cases it may be useful to observe the progression of the chain in this manner. If you just want the chain to run until completion with no graphical components, choose "Run in background".

A second way to run your simulation is from the command-line. Navigate to the directory where you've downloaded ACG by typing something like

```
$cd ~/ACG
```

at the command line. To begin the run just type

```
$ java -jar acg.jar my_input_file.xml
```

You should immediately see some output in the terminal describing your input sequences and a few other tidbits.

Important note: For runs with more than a few sequencs, MC3 runs with lots of chains, or runs with very long sequences, you'll almost certainly need to increase the amount of memory given to acg. This must be done by supplying two additional options from the command line, like this:

```
$ java -Xms1024m -Xmx1024m -jar acg.jar my_input_file.xml
```

This requests 1.0G of memory for acg, which is sufficient for most analyses (you can always give it more by increasing the numbers in the -Xms and -Xmx arguments).

3. Examine the data

The easiest way to examine output is to look at histograms of the parameter values. If you're interested in population size or growth rate (and you've run a model which estimates those parameters), histograms of the sampled values will approximate the likelihood distribution you're looking for. For more advanced analyses, however, you'll want to examine the output of the various Loggers you've chosen. Different loggers generate different types of data, but all will store them in seperate text files whose names you choose. Those most important of these is the "StateLogger", which records the values of the likelihoods and (some properties of) the parameters over time. The State Logger is optional, but recommended for every analysis since it's the only way to assess convergence of the chain. The State Logger produces a file whose name is specified in the input file, but usually ends with .log. This file can be opened in a program such as Tracer (free, available from http://tree.bio.ed.ac.uk/software/tracer) or in the statistical package R.

Other types of loggers produce their own output files, again with names that are specified in the input file. The format of these output files varies slightly from logger to logger but is pretty consistent. The Breakpoint Density logger produces a file that contains two columns, the first indicates the position along the sequence, the second is the relative density of recombination breakpoints (at any depth) at that position. For thorough description of the loggers, see the Appendix A.

Understanding ACG:

At its core ACG does nothing more than run a Markov chain with Metropolis-Hastings (and reversible-jump) accept / reject steps. There's nothing explicitly Bayesian or even explicitly "genetic" about it, although we happen to provide a few tools that we hope are useful for genetic analysis. There are four main types of objects in ACG: Parameters, Likelihoods, Modifiers and Listeners. A Parameter is any object that can influence a likelihood. In a typical genetic analysis a Parameter may be something like the transition to transversion ratio, the population size, the set of base frequencies, even the entire set of ancestral relationships of the sequences (the ARG). ACG updates the Markov chain by proposing an appropriate change for a single Parameter, calculating the new likelihood, and either accepting or rejecting the proposal.

Likelihood objects are anything that can compute a likelihood that depends on the state of one or more Parameters. For instance, the probability of the ARG conditional of the data is a likelihood component (the "Data Likelihood"), and so is the probability of an ARG given the population size and recombination rate. When a new value for a Parameter is proposed by ACG, the various Likelihoods that depend on that Parameter are informed that they need to compute a new value.

A Modifier is a type of object that can propose a new value for a Parameter (these are often called "proposal kernels"). Different Modifiers are appropriate for different types of Parameters. You can't modify an ARG in the same way that you modify the population size, for

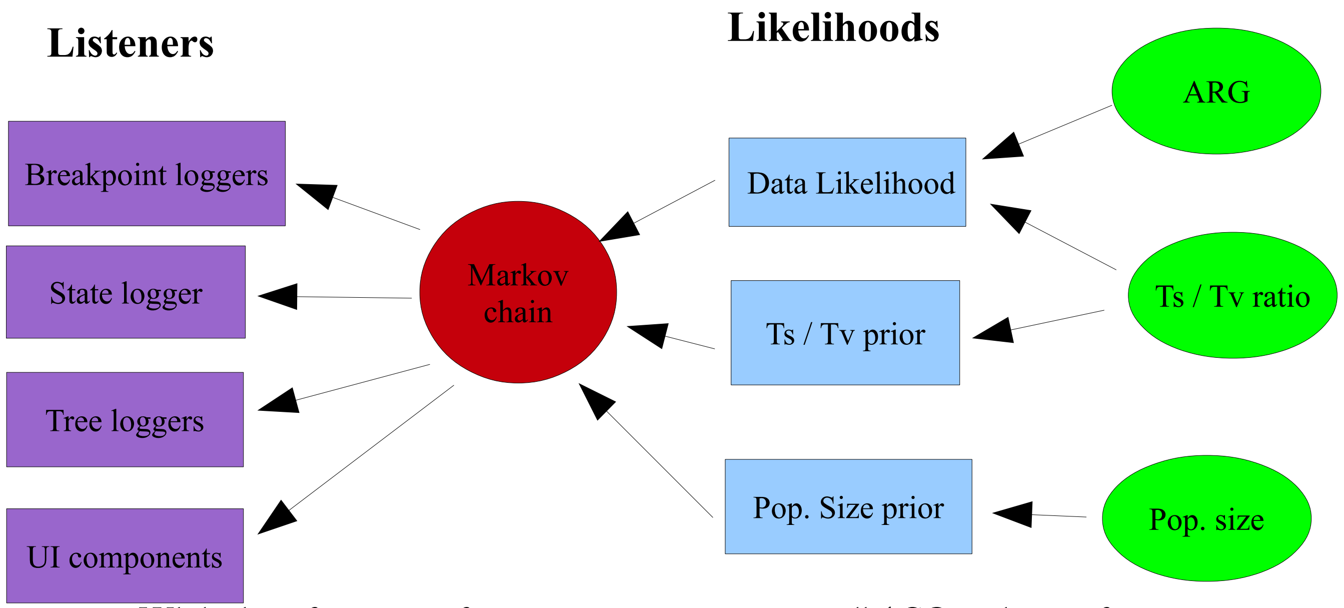
asdasdasd

example.

Finally, Listeners wait around and "listen" for changes in the Markov chain, and periodically do something. Most Listeners write information to files, and these are called "Loggers". Loggers collect information about the state of the chain, for instance the values of

Parameters

the Likelihoods and Parameters, and add it to files that can be examined. See below for a complete list of loggers. Other types of Listeners have to do with debugging or utilities (for instance, writing the current state of the ARG so that MCMC state can be partially recovered after a crash).

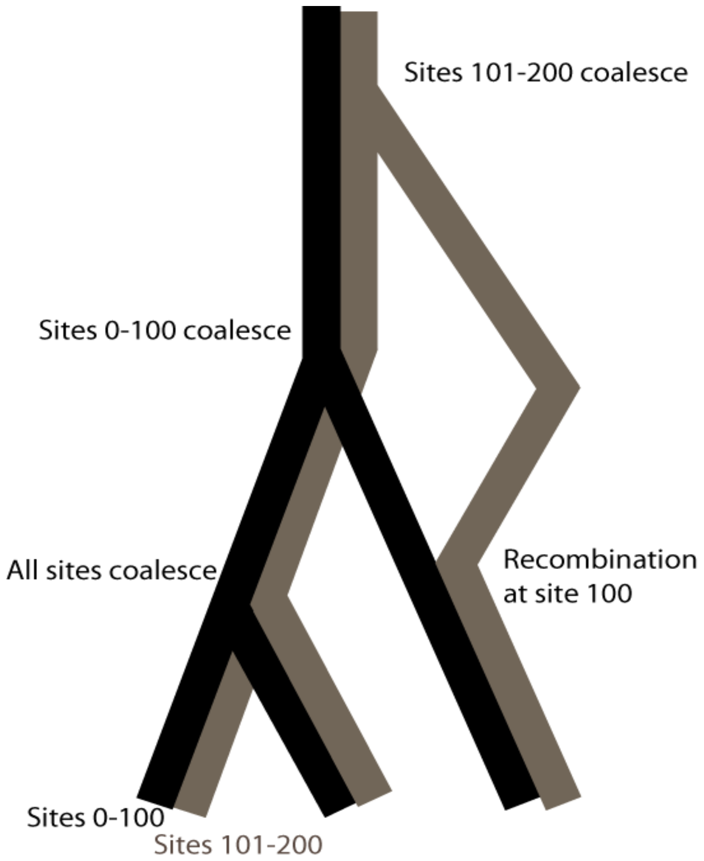


While these four types of components are common to all ACG analyses, a few extra types provide some additional functionality. For instance, a Sequence object stores a sequence, and an Alignment object stores a list of sequences (or a reference to an input file). These types of objects are sometimes required to create a Parameter or Likelihood – for instance, when creating a DataLikelihood object, you'll need a reference to an Alignment as well as an ARG.

An important note about priors: As mentioned above, there's nothing explicitly Bayesian about ACG. It doesn't “know” the difference between the prior and “evidence” portions of the posterior likelihood, and it doesn't require that prior distributions are defined for all parameters. It simply computes the likelihood of whatever components you've defined, and then accepts or rejects the modifications that are proposed. If you don't specify a prior for a parameter, then all values of that parameter are equally likely.   Because of this, the burden is *on the user to ensure a that priors are defined in a way that makes sense.* ACG comes with a few kinds of priors built in, including uniform, exponential and gamma distributions (these are defined in priors.UniformPrior, priors.ExponentialPrior and priors.GammaPrior), and these priors are simply likelihood components that compute the likelihood that an associated parameter has a given value.

Understanding ARGs

An ARG (Ancestral Recombination Graph) is a generalization of a bifurcating tree that includes recombinations, and they are a central feature of most analyses using ACG. In an ARG, lineages can branch as they go both forward (toward the tips) like a tree, and also backward (toward the root) in time. Splits in backwards time represent recombinations, and may significantly alter the topology of the tree. With recombination, site 1 in an alignment



may have a different ancestral tree than site 2. We refer to the tree that is ancestral to a certain site as a "marginal tree". ARGs with more than a few recombinations are difficult to draw and visualize, thus we often resort to dividing an ARG up into its constituent marginal trees and examining them separately (remembering that they're not independent for statistical analysis).

Confusingly, while some recombinations may have a large effect on tree topology, many, if not most, will not. Recombinations may be "trivial" (also called "cryptic"), such that they do not affect the tree shape at any site in any way. Such recombinations are not in principle detectable from genetic data. Other types of recombinations may only affect tree shape slightly, for instance by changing the time of coalescence for some sites by a small amount. These recombinations will be very difficult to detect unless the data has is very informative. ACG hopes to identify as many recombinations as possible, but since so many leave little or no trace in the data we should abandon any hope of pinpointing the exact ARG.

An additional confusing feature of ARGs is that not all features are "visible" from the tips. Some branches and some recombination points are simply not accessible if one traces a particular site from the tips toward the root. For instance, a recombination breakpoint at site 5 is not visible on a branch that is ancestral to sites 10-20. Thus the recombination cannot in principle have any effect on the likelihood of the data given the ARG. Nonetheless, these features are still part of the analysis. Since a breakpoint may *become* ancestral due to rearrangements on other parts of the tree, these "invisible" or "inaccessible" features are maintained at all times in the ARG structure. Consideration of such features is of critical importance for an accurate sampling of ARGs from the distribution defined by the population size and recombination rate.

Many properties of ARGs are viewable from the main application window, under the ARG section of the panel that allows you to choose with properties to log. A brief explanation of each of these follows.

total.recombs : The total number of recombination breakpoints in the ARG

visible.recombs : The number of recombinations that are ancestral to sites observed in the tips. This may be substantially less than the total number of recombinations.

v**isible.height** : The height of the deepest coalescent node in the ARG *at which observed sites coalescence.* For simpler ARGs this will often be the total depth of the ARG, but for more complicated ARGs all sites may end of having a common ancestor at nodes tipward of the root. Another way of thinking about this is that it's the deepest marginal TMRCA for all sites.

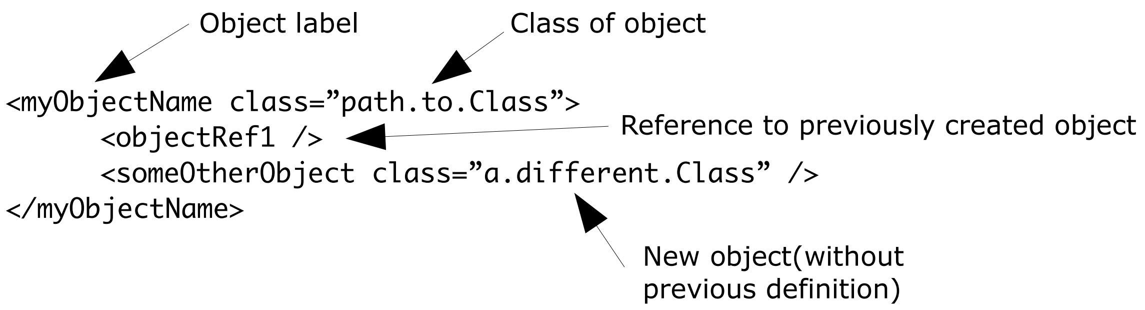
num.nodes : Counts the number of contiguous ranges of coalescing sites, over which the data-likelihood calculating machinery operates. Often the speed of the chain is directly proportional to this value. For simple trees this is always (number of tips) -1 , but the number grows rapidly with recombinations.

num.patterns : A pattern is the fundamental unit of data likelihood computation – it's a single site at which the data likelihood is computed. Since ACG attempts to "compress" or "alias" sites with the same data pattern as much as possible this number is far less than total number of sites * total number of coalescent nodes.

Anatomy of an input file

All ACG analyses are defined by a text-based input file that contains both the data as well as the type of analysis to be performed. They can be a bit confusing at first, one of the best ways to get acquainted with them is to look at the example input files in the doc/ folder. In this section we go through a few of the basics.

An input file is specified in XML and begins with an `<ACG>` tag. Each element of the XML file corresponds to an "object" used in the analysis. The first time a reference to an object appears the object is created, and later XML nodes with the same name are assumed to refer to the same object. Objects are created with a tag that looks something like:



```
<myObjectName class=”path.to.Class”>
      <objectRef1 />
      <someOtherObject class=”a.different.Class” />
</myObjectName>
```

Object label
Class of object
Reference to previously created object
New object(without previous definition)

"myObjectName" is an arbitrary label for the object. It can be anything you like, and is just there so you can refer to the same object again later. The class="some.Class" part defines what *type* of object this is, and is required for the first reference to an object (for a listing of all types see the appendices). The class attribute is case sensitive, be sure to get it right. For later references to the same object you can just use `<myObjectName />`, and skip the class part – the class is only required when you create the object.

Some objects require references to other objects in order to be created. For instance, the DataLikelihood object requires a reference to both an ARG as well as a mutation model. This is accomplished with the following syntax:s

```
<dataLikelihood class=”dlCalculation.DataLikelihood”>
      <myMutationModel />
      <myARG />
</dataLikelihood>
```

Note that order is important here, you'll get an error if you reverse the order of the arg and mutation model references.

Some objects require references to be in "list" form. This typically happens where an object can't anticipate the number of other objects it will get references to. For instance, an alignment doesn't know in advance how many sequences it will be given. In this case, the sequences must be wrapped in a list, like this:

```
<alignment class=”sequence.Alignment”>
      <sequences class=”list”>
            <sequenceOne class=”sequence.Sequence”>
                  ATAGGATACGACGACTAGCGACTACG...
            </sequenceOne>
            <sequenceTwo class=”sequence.Sequence”>
                  ATAGGATACGACGACTAGCGACTACG...
            </sequenceTwo>
            ….
      </sequences>
</alignment>
```

The special class=”list” attribute creates a variable-length list of objects. Other cases where you'll need to make a list include passing modifiers to Parameters and providing lists of Likelihoods, Parameters, and Loggers to MCMC objects.

Much more about what's legal and what's not legal in an ACG input file can be learned by examining the example input files, usually located in the doc/ directory. These are relatively well commented and progress from simple to more complex, hopefully giving you a chance to learn as you go along.

Walkthrough of a simple example

To begin, we'll start with the input file called `example1.xml`. It contains simulated data from 10 sequences of 2000 nucleotides each. Our goal will be to infer a few basic properties such as the (scaled) population size, the shape of the genealogy, and the locations of recombination breakpoints. But first, we'll step through the XML input file to introduce the various objects you'll need.

The first line after the `<ACG>` tag reads:

```
<RandomSource class="math.RandomSource" />
```

This creates an object the generates the random numbers for the analysis. One of these is always required, if you don't create one of these you'll get a "java.lang.NullPointerException" error. If you'd like to specify a seed, you can do so with the seed="XXX" attribute.

The next block specifies an alignment of sequences to use as input for the analysis. The sequences must be specified inside of an object of type "list", as shown below.

```
<alignment class="sequence.Alignment">
      <sequences class="list">
            <tip7 class="sequence.Sequence">
            TCCAAGAATGGAGGCATGCTCTT...
            </tip7>
            … 9 more sequences …
</alignment>
```

Now that we have some sequences we can create an ARG. ARGs can be constructed with a wide variety of different input options, including starting trees, data sets, and modifiers, but for this analysis we'll keep it simple and just provide the <alignment> and a list of modifiers (proposal kernels). The full block looks something like:

```
<arg class="arg.ARG" frequency="30.0">
      <alignment />
      <modifiers class="list">
            <recombAddRemove class="modifier.RecombAddRemove"/>
            <rootHeight class="modifier.RootHeightModifier" />
            <nodeHeight class="modifier.NodeHeightModifier"/>
            <swapper class="modifier.SubtreeSwap" />
            <wideSwap class="modifier.WideSwap" />
            <shifter class="modifier.BreakpointShifter" />
            <bpSwap class="modifier.BreakpointSwapper" />
      </modifiers>
</arg>
```

Note the “list” notation used to wrap the list of modifiers. New here is the use of the frequency=”30.0”, which specifies how frequently the ARG will be sampled for modification, relative to the other parameters. Typically this should be a fairly high number, since ARGs are much more difficult to sample than, say, kappa.

Next up is the mutation model. We're going with the relatively easy F84 (Felsenstein '84) model here, and giving it a single parameter, the transition / transversion (Ts/Tv) ratio. In this model we'd like to estimate this value from the data. To do this, we make a Parameter object like so:

```
<kappa class="parameter.DoubleParameter" name="kappa" value="2.0"
lowerBound="0.5" upperBound="500">
      <kappaMod class="modifier.SimpleModifier" />
</kappa>
```

The class of “DoubleParameter” , which is a type of Parameter that just takes on a single (double-precision) value. The name=”kappa” attribute specifies what name will appear in the log file, the value=”2.0” specifies the starting value, and the upper and lower bounds determine the permissible range of values. The modifier is a simple “sliding-window” type modifier that moves the value up or down by a small amount each modification.

The mutation model itself needs a reference to kappa, and is created with the following:

```
<mutationModel class="dlCalculation.substitutionModels.F84Matrix"
stationaries="0.25 0.25 0.25 0.25">
      <kappa />
</mutationModel>
```

Next up is the “Data Likelihood” calculation object, which calculates the probability of observing the sequences conditional of the (current state of) the ARG. It requires a reference to the ARG and a mutation model:

```
<DLCalculator class="dlCalculation.DataLikelihood">
      <mutationModel />
      <arg />
</DLCalculator>
```

We also specify a simple model of population size and recombination rate. For this example we'd like to estimate both of these parameters from the data, but we'll make the simple assumption that both are constant across sites and time. Each parameter is also modified with a “ScaleModifier” that modifies the value by multiplying it by a value close to 1.0.

```
<constantRecRate class="coalescent.ConstantRecombination">
      <recRateScaler class="modifier.ScaleModifier"/>
```

```
</constantRecRate>

<recPrior class="priors.ExponentialPrior" mean="25">
      <constantRecRate />
</recPrior>

<coalescentLikelihood class="coalescent.CoalescentLikelihood">
      <constantPopSize />
      <constantRecRate />
      <arg />
</coalescentLikelihood>
```

To collect and report data we're going to use a couple of the simplest loggers. The first is the State Logger, which records the likelihoods and values of the parameters in a text file, suitable for viewing with Tracer or R. The StateLogger needs a filename attribute so it knows what file to write to, as well as how frequently to log new data (at least 1000 is good here), and finally whether or not it should echo values to the screen in addition to writing them to the file.

```
<statelogger class="logging.StateLogger" filename="example1.log"
frequency="1000" echoToScreen="true" />
```

We're also interested in finding the locations of all the recombination breakpoints. To accomplish this, we'll use the BreakpointDensity logger, which tracks the locations of recombinations along the sequence as a single histogram. It writes its data to a file with name given by the filename=”example1_bp.txt” attribute. It also requires a reference to the ARG to track the breakpoints for.

```
<bplogger class="logging.BreakpointDensity" filename="example1_bp.txt"
frequency="1000" burnin="10000">
      <arg />
</bplogger>
```

At last, the object is created that actually runs the MCMC analysis. We must supply the object with lists of the Parameters, Likelihoods, and Loggers we're using for the analysis – in that order. It's easy to forget a Parameter or Likelihood here, so be sure to double check and make sure that all Parameters you've defined have been included. The run=”true” option indicates that we'd like to chain to begin running right away.

```
<mc class="mcmc.MCMC" length="5000000" run="true">
      <parameters class="list">
            <constantPopSize />
            <constantRecRate />
            <kappa />
            <arg />
      </parameters>
      <likelihoodComponents class="list">
            <recPrior />
            <coalescentLikelihood />
            <DLCalculator />
        </likelihoodComponents>
        <loggers class="list">
            <statelogger />
            <bplogger />
        </loggers>
</mc>
```

To begin a run, try typing the following at the command prompt:

```
$java -jar acg.jar example1.xml
```

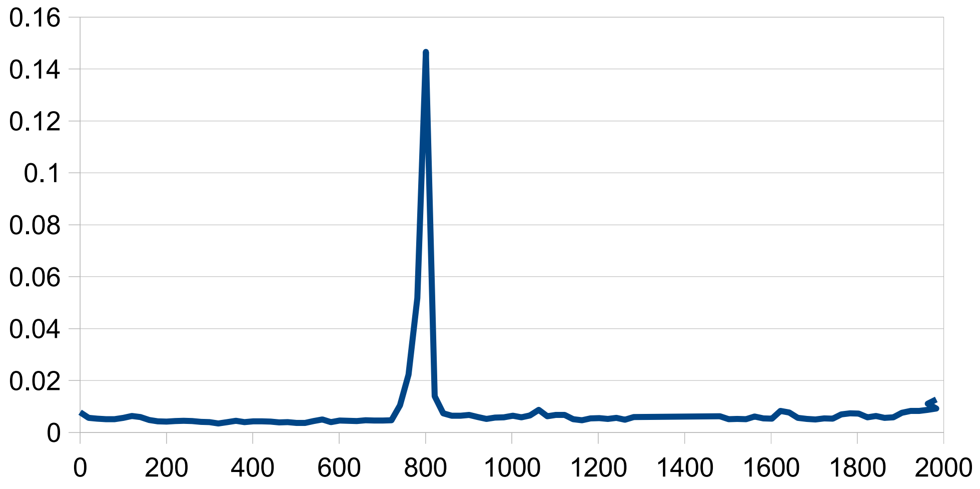
More or less immediately you should see some lines of output like the following :

```
  ACG : Markov-chain Monte Carlo analysis of recombinant genealogies
  version : 0.01   June, 2011

 Brendan O'Fallon
 University of Washington
 email: brendano@u.washington.edu

Initiating random source with seed: 115351319 0
Generating initial tree from UPGMA tree
Initializing timers for Markov chain
State    ExpPrior( Recombination rate )     Coalescent          treeLikelihood       num.patterns       num.nodes
         popSize   rec.rate   kappa    site.height         site.bps  total.bps  mc.speed
1000     -3.2788        6.1143          -4015.8023669368413           324   16
         0.56997        1.498563        2.366604    0.0433452119462273 2             2
         2         727.8
2000     -3.2812        8.8316          -4012.7381486930417           326   16
         0.445941       1.557002        2.338202    0.0426303092869147 2             2
         562.43
3000     -3.2933        11.8637         -4009.1829615898214           327   16
         0.343391       1.860185        2.658961    0.0409187911880259 2             2
         2         575.71
4000     -3.3354        16.9291         -4011.0353869137234           326   16
         0.2219546      2.912105        2.371193    0.0369732151567735 4             2
         2         1383.13
5000     -3.3817        17.4604         -4008.2654314493093           323   16
         0.2144053      4.070623        2.039768    0.0412351241973505 2             2
         2525.25
6000     -3.4082        18.5726         -4010.3251990359336           306   15
         0.2012915      4.734232        2.397306    0.046464184851795355             2
         2         4975.12
7000     -3.3913        21.9116         -4012.705551577121   354   16      0.135288
         4.310426       2.475272        0.041462702419386724         2         2
         8130.08
```

Depending on how fast your system is it will take a few (4-5) minutes for the run to complete. The various columns of output correspond to various properties of the Parameters and Likelihoods. Our two loggers should each produce a text file.

The State Logger should have created a file called example1.log. This file can easily be imported examined with tools such as R, Tracer, or a spreadsheet program such as OpenOffice Calc. One of the first columns to examine is "site.bps", which records how many recombination breakpoints were ancestral to the your data. In this case, you should see a mean number of breakpoints at around 4-5. (The total.bps column records the total number of recombination breakpoints. Not all of these will be ancestral to the data, and hence this statistic is somewhat less interesting than sites.bps.) The fact that the mean number of recombinations is relatively high strongly suggests that there were important recombination breakpoints in our data. Similarly, the distribution of the recombination rate parameter does not include zero, again demonstrating the importance of recombination in the data.

But where are these recombinations? Try opening the example1_bp.txt file, and viewing the contents in R or a spreadsheet program. It should look something like this:



The x-axis denotes position along the sequence, and the y-axis denotes the density of recombination breakpoints. Thus, the analysis feels confident that there's a recombination or two sitting right near site 800, but little evidence for recombinations elsewhere (as it turns out, the recombination is at site 772).

At this point you might be a little confused; didn't the state logger say that there were around 4-5 recombinations? Where are the rest of them? The answer to this question lies in the fact that the MCMC is constantly exploring new ARGs, many of them with additional recombinations. The time the chains spends exploring a certain part of ARG space if proportional to the likelihood of that ARG, so if a recombination makes the ARG a lot more likely, the chain will spend lots of time with that recombination in place. However, many recombinations won't have much of an impact on the likelihood,since they're trivial, or nearly so. These will come and go at all points along the length of the sequence with equal probability., and hence don't show up as a single peak.

What to do now? Well, you've gathered strong statistical support for a recombination breakpoint near site 800. You may be interested in what the trees look like on either side of that breakpoint. To investigate this, you could re-run the analysis, this time with loggers that collect the marginal trees on either side of the breakpoint – for instance, at sites 400 and 1200. Each marginal tree logger will produce its own log file full of all the marginal trees sampled at the given site, and you can examine the trees in a program such as FigTree, or build a consensus tree from them (for instance, using the program CONSENSE from the Phylip package) and compare the trees at the two sites. To create a marginal tree logger, add the following line to the XML input file:

```
<treeLogger400 class=”logging.MarginalTreeLogger”  site=”400”
filename=”marginalTrees400.trees”>
      <arg />
</treeLogger400>
```

And be sure to add a reference to the logger to the listeners for the MCMC! This will log the trees at site 400 to a file called “marginalTrees400.trees”. Adding another one for site 1200 is easy, just add another block like the one above, but with a new name and a different site, and different file name. When you run the analysis now, you should see two new files created with the names you've chosen.

Using the Graphical Interface

Once you've prepared the input file, you can execute it and watch it run within a Graphical User Interface (GUI). The GUI allows you to easily see how various properties of the model (parameter values and likelihoods, in particular) change over time. To start the GUI, either double-click the acg.jar icon, or, from the command line, enter

```
$ java -jar acg.jar
```

The first window will ask you to choose an input file. Choose an input file either by entering the name of a file you've prepared or clicking the "Browse" button to navigate to it. Once you've selected your file, you can choose to run it in the "Background", with no graphical output, or in the GUI. If you choose the first option, the chain will begin running but the window will disappear.

If you choose to "Run with GUI", you'll be met with another screen that allows you to choose what elements you'd like to monitor. This screen can be a bit confusing because different types of parameters give you different choices for things to monitor. For instance, a transition to transversion ratio (or, really, any DoubleParameter) has just a single number to track, so that one's easy. But the Base Frequencies parameter has four values. Worse, an ARG has no obvious numerical value – there are many ways to describe it. To deal with this confusion, each parameter stores a list of values that it can potentially display, and you can choose from these various properties. For DoubleParameters, this is always just a single number. But for ARGs there are a handful. ARGs allow you to watch the total number of recombinations in the tree ("total.recombs"), the number of recombinations that are "visible" or ancestral to observed sequences ("visible.recombs"), the height of the most rootward coalescent node that contains sites ancestral to observed sequences ("visible.height"). Select a few of these (less than 10) and then click "Start chain" to begin the actual MCMC run.

A note about the GUI

In an effort to accommodate expansions and plugins, the input file format is extremely general. In fact, valid input files need not contain any parameters, likelihoods, or MCMC objects. Such a file won't run in the GUI, which expects a few basic things from the input file. The GUI was written to be as accommodating as possible, but it does expect to find Parameters and MCMC objects in the file. If you've done a lot of customization or are working with plugins developed from far afield, the GUI may simply not work. But as long as the input file is valid and refers to objects that can be found in plugin files it can be run from the command line.

When the chain begins another screen will appear that holds a bunch of figures. Each figure corresponds to one item you've chosen on the previous screen. The figures draw lines showing the values that each parameter or likelihood has taken, and new values are added as the chain advances. Many aspects of these figures are configurable, either by double-clicking an item (such as the data series, the axis, the labels, or the key), or by right clicking in the chart area. Basic properties of the individual series can be configured by double-clicking on the series (on the line itself). For instance, you can change the line color, width, and the representation of the series to points, points and lines, or boxes, etc. Series can also be removed by clicking the "Remove" button, either from the Legend configuration pane, or an individual series configuration pane. Often it's convenient to remove the burn-in series once the burn-in period is over.

Initially, the traces are colored gray, indicating the the "burnin" period (set in the previous pane) has not been exceeded. When the burnin is done, traces become more colorful, and several new lines appear. The blue line shows the values over time. The solid red line shows the cumulative mean (the mean of all values sampled since burn-in up to the point in question), and the two dashed red lines indicate the mean +/- one standard deviation (again, a cumulative score since the end of the burn-in). The red lines can help to assess convergence of the chain. If they appear relatively stable over a "long" period, then the chain has probably reached a local optimum.

You can also switch to "Histogram" mode by right-clicking the figure and choosing "Switch to Histogram". Histograms show you how many times that parameter has sampled values in a certain range, and can be very useful if you're interested in the distribution of values your chain is sampling.

It's also possible to easily save images by right-clicking and choosing "Save image" from the pop-up menu. For now, all images are saved in .png format, although we may provide additional formats in the future.

Using argutils.jar

A handful of utilities are provided in the ancillary argutils.jar file. It can be executed from the command line by

```
$ java -jar argutils.jar
```

although doing so without any options or files specified will result in a brief help message. Right now it supports the following options:

--extract-trees [argfile.xml] : Extract all marginal trees as newick strings
--extract-tree [argfile.xml] [site] : Extract the marginal tree at a given site as a newick string
--emit-bps   [argfile.xml] : Emit the locations (site & height) of all recombination breakpoints
--emit-tmrca [argfile.xml] : Emit the time to most recent common ancestor along the sequence
--scale [factor] [argfile.xml] : Emit a copy of the arg with all branch lengths scaled by the given value
--genarg [tips] [theta] [rho] [sites] : Generate a random arg with the given number of

tips, theta, recombination rate (rho), and number of sites, and emit it as an xml file.

--convert [sequencefile.fas | sequencefile.phy] : Generate an ACG-style alignment block from the given fasta or phylip-formatted sequence file

Appendix A: Loggers

State Logger [logging.StateLogger]: The state logger is the most basic type of logger. It asks the Markov chain for all Likelihood and Parameter components, and writes their values periodically to a file that you specify. Each Parameter chooses may choose to define 'value' in a different way. For instance, the transition to transversion ratio in the F84 model has a pretty clearly defined single number as a value. The Base Frequencies parameter, however, has four separate values. ARGs are also a Parameter, but can't easily by summarized by any number. For ARGs, several different characteristics are reported to the state logger. An additional feature of the State Logger is that it tracks the speed of the analysis, and reports it in MCMC states per second. Because the State logger queries the Markov chain for the list of components and parameters, you don't have to supply it with any parameters for it to function.

Marginal Tree logger [logging.MarginalTreeLogger]**:** The marginal tree logger writes out the (non-recombining) single tree at a particular site periodically to a file. The list of trees can then used to build a consensus tree to summarize the ancestral information at a single site, or to examine for other properties.

MPE ARG logger [logging.MPEARG]: Writes the full ARG with the maximum probability given the data to a file.

Last ARG logger [logging.LastARGLogger]: Primarily a utility, this just writes whatever the current ARG is to a file (overwriting the previous ARG), by default every 10000 MCMC states. This is often convenient so that MCMC runs can be continued from a point near where another run left off.

Marginal tree height logger [logging.RootHeightDensity]: Collects posterior distribution of TMRCA at each site across the length of the sequence. By default, seven values are reported, the lower 95%, 90% and 80% HPD boundary, the median, and the upper 80%, 90%, and 95% HPD boundaries. In the same file, the mean tree height across sites is also written.

Breakpoint density logger [logging.BreakpointDensity]: Builds a histogram over sites of the location of recombination breakpoints that are ancestral to the data (recombination breakpoints that are not ancestral to any sites are ignored). By default, the sequence is divided into 500 bins.

Breakpoint location logger [logging.BreakpointLocation]: Builds a 2D histogram of the locations of breakpoints, where one axis extends across sites in the sequence, and the other axis extends across tree heights. This enables one to pinpoint the location of a recombination not only along the sequence, but also in time. The densities of breakpoints (only the ones that are ancestral to the data) are stored as single real numbers in a large matrix. The easiest way to view the matrix is to import it into R (or a similar program), and use the image(x) command, where x is the matrix of values. This will produce a “heat-map” style plot of the areas where recombinations are likely to have occurred.

Common errors

Increasing the amount of memory

Unreferenced params / likelihoods

Weird array index out of bounds error