Coalescent Simulation in Python or Julia

Alan R. Rogers

March 14, 2022

1 Introduction

In this project, you will look under the hood to see how coalescent simulations are written in either Python or Julia. I'll provide a working program, which uses the coalescent algorithm to build a gene genealogy, assuming selective neutrality, random mating, and constant population size. There is no mutation.

Your first step is to get it running on your own machine. The next step is to modify the code so that it simulates the site frequency spectrum. For this purpose, you will write a function that traverses the gene genealogy, simulates mutations on each branch, and allocates these mutations to the relevant entry of the site frequency spectrum.

The initial code is available in two versions. This one is in Python, and this one is in Julia. Both versions implement the same algorithm. I include the Python version, because Python is widely used and easy to learn. The Julia version is much much faster and would be a better choice in real research.

Download the version of your choice, and make sure you have a recent version of Python and/or Julia. Here's an interaction that illustrates how to use the Python version:

```
$ python3
Python 3.10.2 (v3.10.2:a58ebcc701, Jan 13 2022, 14:50:16) \
    [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from coalesce import *
>>> main()
Repetition 0
  tree depth         : 109193.49879333175
  total branth length: 357621.02604959125
  number of tips     : 5
Repetition 1
  tree depth         : 168516.8538641843
  total branth length: 566500.3139273041
  number of tips     : 5
```

I've written this code so that all it does is define functions. Consequently, when you import it, nothing runs. There is a function called `main`, which will run the coalescent simulation several times and print some output. After importing the code into the Python shell, type `main()` to run this program.

Here is an illustration of the Julia version:

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.7.0 (2021-11-30)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("coalesce.jl")
main (generic function with 3 methods)

julia> main()
Repetition 1
  tree depth         : 45006.27393393751
  total branth length: 130365.20506580963
  number of tips     : 5
Repetition 2
  tree depth         : 52792.55282534899
  total branth length: 131608.06412097567
  number of tips     : 5
```

2 Nodes

The coalescent algorithm builds a *gene genealogy*—a tree of connected nodes, each of which represents a piece of DNA (perhaps a single nucleotide site) at some point in time. There is one node for each nucleotide in our modern sample. These "tip nodes" have age 0, and they have no children, but each of them has a parent. The oldest node in the genealogy is called the "root;" it has two children but no parent. All others are "internal nodes" and have two children and one parent.

To implement this in software, we define something called "Node," which is a "class" in the Python version. It has four data fields:

age a floating-point number

parent either another Node or None

left a child, which is either another Node or None

right a child, which is either another Node or None

Here is the Python code that defines class Node:

```
class Node:

    def __init__(self, left=None, right=None, age=0.0):
        self.parent = None
        self.left = left
        self.right = right
        self.age = age
        if self.left != None:
            self.left.parent = self
```

```
        if self.right != None:
            self.right.parent = self
```

The `__init__` method is a constructor, which builds an instance of class Node. This method is called in response to statements such as

```
a = Node()
b = Node()
```

This says that `a` and `b` will refer to newly created objects of type Node. Since I called Node with no arguments, the defaults of the `__init__` method will be used: the new nodes will have age 0, and their `parent`, `left`, and `right` fields will all equal None. The `Node` constructor is also used to link nodes together. For example,

```
c = Node(a, b, 1.234)
```

creates node `c`, whose children are `a` and `b`, and whose age is 1.234. This statement also modifies nodes `a` and `b` so that their `parent` fields are no longer None, but instead refer to `c`. The `parent` field of `c` equals None.

3 The coalescent algorithm

For background on coalescent theory, see chapters 4–5 of Lecture Notes on Gene Genealogies. That document focuses on the expectations of statistics related to gene genealogies. Here, our emphasis is on the variation around these expectations. Computer simulations are useful in studying this variation, and the coalescent algorithm is central to this endeavor. It is an approximation to the Wright-Fisher model of genetic drift. The Wright-Fisher model measures time in discrete generations. The coalescent algorithm treats time as continuous and is therefore only an approximation. It is an excellent approximation, however, if the population is at all large.

We start with a sample of k modern gene copies and trace their ancestry backwards into the past. Now and then, two of the lineages will coalesce into a single ancestral lineage. This is called a *coalescent event*. As we travel backwards into the past, how long is the interval until the next such event? If there are k lines of descent, this interval is an exponential random variable with mean $4N/k(k-1)$. When a coalescent event occurs, we choose a random pair of lineages, join them to form a single parental node, reduce the number of lineages by 1, and then proceed with the next coalescent interval.

Let us work our way through a Python function that implements this algorithm. The first line is

```
def coalesce(rng, n = 10000, k=5):
```

This defines a function called "coalesce," which takes 3 arguments: `rng` is a random number generator, `n` is the diploid population size, and `k` is the haploid sample size (number of gene copies). Next, we define several variables:

```
    age = 0.0
    fourN = 4*n
    nodes = [Node() for i in range(k)]
```

Here, `age` represents time in generations measured backwards from the present. Each time we define a new Node, its age will equal the current value of this variable. The `fourN` variable is just 4 times `n`. It is defined outside the "while" loop below to avoid doing this multiplication over and over again within the loop. The variable `nodes` is a list of `k` objects, each created by a call to `Node()`. These represent the gene copies of our modern sample.

The heart of the algorithm is a "while" loop. Each pass through the loop deals with one coalescent interval. The first pass deals with the most recent interval, in which the number of lineages equals the size of the modern sample. At the end of each pass through the loop, the number of lineages is reduced by 1. We fall out of the loop when k=1. The loop begins like this:

```
    while k > 1:
        tmean = fourN/(k*(k-1))
        age += rng.exponential(tmean)  # time until next coalescent event
```

Here, we have set `tmean`, the expected time until the next calescent event. Using this value, we draw a random value from the exponential distribution and add this to the current value of `age`. This establishes the time of the next coalescent event. Having chosen the time, the next step is to choose two random nodes, `i` and `j`, ensuring that `i < j`.

```
        i = rng.integers(k)
        j = rng.integers(k-1)
        if j == i:
            j = k-1
        if j < i:
            i, j = j, i
```

Next, join nodes `i` and `j`, and put their parent into position `i` within the array of nodes.

```
        nodes[i] = Node(nodes[i], nodes[j], age)
```

Finally, shorten the array, and forget about node j.

```
        if j < k-1:
            nodes[j] = nodes[k-1]

        k -= 1
```

We are now out of the "while" loop, and the number, `k`, of nodes is equal to 1. This last node is the root of the gene genealogy. This root node is the value returned by the function.

```
    return nodes[0]
```

4 Using recursive functions to work with gene genealogies

Your project will involve writing a function that calculates the site frequency spectrum. Before discussing that problem, let's consider a simpler one: calculating the number of tip nodes that are descendants of a given node. Here's a function that does this job:

```
def ndescendants(node):
    if node.left == None:
        return 1
    else:
        return ndescendants(node.left) + ndescendants(node.right)
```

This is an example of a *recursive* function—one that calls itself. The function begins by asking whether `node.left` equals `None`. This is true only for tip nodes, and a tip node has 1 "descendant"—itself. Therefore, if this condition is true, the function returns 1.

Otherwise, we're at either the root or an internal node, and the number of descendants is the sum of the descendants of the left and right children. The function therefore calls itself twice (once for the left child and once for the right) and returns the sum of the two results.

Once you get the hang of recursive functions, you will find that they often make things far far simpler. Without recursion, the `ndescendants` function would be much more complex.

5 Calculating the site frequency spectrum

A function that calculates the site frequency spectrum will need several arguments:

node a reference to the current node in the gene genealogy. On the initial call, `node` will refer to the root of the gene genealogy. But then the function will call itself recursively to traverse the entire genealogy.

rng a random number generator. This isn't needed in Julia, because the random number generator is global. The `rng` is used to generate mutations by sampling from the Poisson distribution.

mu the mutation rate, a floating point number.

spec an array of integers, whose length should equal `k-1`, where `k` is the number of gene copies in the sample. On the initial call, all entries of `spec` should equal 0. On return from the Python version of the code, `spec[i]` will equal the number of mutations that have `i+1` descendants within the sample. In Julia, `spec[i]` will equal the number of mutations that have `i` mutations within the sample. This difference arises because the first entry in the array is `spec[0]` in Python but `spec[1]` in Julia.

The first line of the function looks like this in Python:

```
def spectrum(node, rng, mu, spec):
```

and like this in Julia

```
function spectrum(node::Node, mu::Float64, spec::Array{Int64})
```

The function will use a random number generator to determine the number of mutations that arose along the branch between the current node and its parent. Then it will add those mutations to the appropriate entry of the `spec` vector. The appropriate entry depends on how many descendants the current node has within the modern sample. For example, suppose that there are `nmut` mutations and `nkids` descendants. Then we would want to execute this statement in Python:

```
spec[nkids-1] += nmut
```

or this one in Julia

```
spec[nkids] += nmut
```

To determine `nkids`, the function should visit descendant nodes before modifying `spec`, and it (like `ndescendants` above) should return the number of descendants the current node has within the modern sample.

The behavior of the `spectrum` function should depend on whether the current node is the root, a tip, or an internal node. The root node has no parent. Furthermore, mutations upstream from the root node do not contribute to the site frequency spectrum. For this reason, if the current node is the root, the function should not modify `spec`. It should call itself recursively on the `left` and `right` children. These calls will return the numbers of descendants of the two children. At the root node, the function should do nothing but return the sum of these two values.

If the current node is a tip, then there are no children: `left` and `right` will equal `None` (in Python) or `nothing` (in Julia). Nonetheless, a tip node does have one descendant: itself. Thus, `nkids=1` and there is no need for recursive calls to `spectrum`.

If the current node is neither the root nor a tip, then it is an internal node. Before modifying `spec`, it will be necessary to call `spectrum` on the `left` and `right` children in order to determine `nkids`, the number of descendants. Then, the length of the branch connecting the current node to its parent is

```
brlen = node.parent.age - node.age
```

and the expected number of mutations is `mu * brlen`. This quantity can be handed to a Poisson random number generator in order to determine the number of mutations. This looks like

```
nmut = rng.poisson(mu * brlen)
```

in Python, or

```
nmut = rand(Poisson(mu * brlen))
```

in Julia. After adding `nmut` to the appropriate entry of `spec`, the function returns `nkids`.

This leads us to the following algorithm:

1. If `node.parent` equals `None` (in Python) or `nothing` (in Julia), then the current node is the root. Call `spectrum` on the `left` child and on the `right` child, and return the sum of the two resulting values.
2. If we reach this step, then we're at either a tip or an internal node, and we need to set `nkids`. If `node.left` equals `None` (Python) or `nothing` (Julia), then set `nkids = 1`. Otherwise, set `nkids` equal to the sum of the values obtained by calling `spectrum` on the `left` and `right` children.
3. Calculate `brlen` and `nmut` as explained above.
4. Add `nmut` to `spec[nkids-1]` (Python) or `spec[nkids]` (Julia).
5. Return `nkids`.

6 Additional functions

The files `coalesce.py` and `coalesce.jl` contain two additional functions, which will be useful in the assignment below:

`espectrum` calculates the expected value of the site frequency spectrum [1, Ch. 6], a function of k (the haploid sample size), and $\theta = 4N\mu$, where N is the diploid population size and μ the mutation rate per site per generation.

`msqdiff` returns the mean squared difference between two vectors.

In the assignment below, you will use `espectrum` to calculate the expected spectrum, and `msqdiff` to compare the observed and expected spectra.

7 Assignment

Coalescent theory tells us that the expected value of the site frequency spectrum depends on only two parameters: the haploid sample size, and $\theta = 4N\mu$, where N is the diploid population size and μ is the mutation rate. If you double N and also halve μ, the expected spectrum is unchanged. But what about the variation about this expected value? Does it vary in response to N, even when θ is constant? This exercise will use computer simulations to answer this question.

1. Write the `spectrum` function described above, either in Python or in Julia.
2. Modify the `main` function so that it calls `spectrum` each time it runs a coalescent simulation. This just involves removing comment characters from several lines in function `main`.
3. Calculate the mean squared difference between the simulated spectrum and its expected value [1, Ch. 6]. This also involves removing comments from lines in `main`.
4. Average this mean squared difference across at least 10,000 coalescent simulations.
5. Repeat this experiment for diploid population sizes 100, 1000, 10,000, 100,000, and 1,000,000, using a haploid sample size of 30 in each case. For each of these experiments, adjust the mutation rate so that θ doesn't change.
6. How does the variation of the site frequency spectrum about its expectation depend on population size, when θ is held constant?

Your report should include your code, sample output, and a few sentences summarizing your conclusions.

References

[1] Alan R. Rogers. Lecture notes on gene genealogies. `http://content.csbs.utah.edu/~rogers/tch/ant5221/ggeneal.pdf`, 2013.