How to Study Evolution Using Scientific Python

Yoav Ram

PyCon Israel 2016
Who I am

• PhD in **Evolutionary Theory** at TAU
• Using Python since 2002
• Using **Scientific Python** since 2011
• Teaching Python since 2011
• Python training for engineers & scientists
Theoretical Evolution
Formally: Population genetics

• Study **changes in frequency** of gene variants within populations
• The main forces of evolution:
  – Natural selection
  – Random genetic drift
  – Gene flow & recombination
  – Mutation
• Focus on **adaptation**, speciation, population subdivision, and population structure
Evolution

University of California Museum of Paleontology's Understanding Evolution
Natural Selection
Random Genetic Drift
Wright-Fisher Model

Standard model for change in frequency of gene variants.

**R.A. Fisher**
1890-1962
UK & Australia

**Sewall Wright**
1889-1988
USA
Wright-Fisher Model

Standard model for change in frequency of gene variants.

Two gene variants: 0 and 1.
Number of individuals with each variant is $n_0$ and $n_1$.
Total population size is $N = n_0 + n_1$.
Frequency of each variant is $p_0 = n_0/N$ and $p_1 = n_1/N$. 
Wright-Fisher Model

Assume that variant 1 is **favored by selection** due to better survival or reproduction.

The frequency of variant 1 after the effect of selection natural \( p_1 \) is:

\[
p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)}
\]

\( s \) is a selection coefficient, representing **how much variant 1 is favored over variant 0.**
Wright-Fisher Model

Random genetic drift accounts for the effect of random sampling.

Due to genetic drift, the number of individuals with variant 1 in the next generation \((n'_1)\) is:

\[ n'_1 \sim \text{Binomial}(N, p_1) \]

The **Binomial distribution** is the distribution of the number of successes in \(N\) independent trials with probability of success \(p_1\).
Fixation Probability

Assume a single copy variant 1 in a population of size $N$.

What is the probability that variant 1 will take over the population rather than go extinct?
NumPy

The fundamental package for scientific computing with Python:

• N-dimensional arrays
• Random number generators
• Array functions
• Broadcasting
• Tools for integrating C/C++ and Fortran code
• Linear algebra
• Fourier transform

numpy.org

Loosing your loops
Into the code
Natural Selection

\[ p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)} \]

Random drift

\[ n_1' \sim \text{Binomial}(N, p_1) \]

```python
from numpy.random import import binomial

n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```
Random drift

\[ n'_1 \sim \text{Binomial}(N, p_1) \]

Natural Selection

\[ p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)} \]

```
from numpy.random import binomial

n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```
Natural Selection

\[ p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)} \]

Random drift

\[ n'_1 \sim \text{Binomial}(N, p_1) \]

```python
from numpy.random import binomial

n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```
Random drift

\[ n'_1 \sim \text{Binomial}(N, p_1) \]

Natural Selection

\[ p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)} \]

```python
from numpy.random import import binomial

n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```
Due to genetic drift, the number of individuals with variant 1 in the next generation is $n_1$.
Natural Selection

\[ p_1 = \frac{n_1 \cdot (1 + s)}{n_0 + n_1 \cdot (1 + s)} \]

Random drift

\[ n'_1 \sim \text{Binomial}(N, p_1) \]

```python
from numpy.random import import binomial

n1 = 1

while 0 < n1 < N:
    n0 = N - n1
    p1 = n1*(1+s) / (n0 + n1*(1+s))
    n1 = binomial(N, p1)

fixation = n1 == N
```
NumPy vs. Pure Python

NumPy is useful for random number generation:

```python
n1 = binomial(N, p1)
```

Pure Python version would replace this with:

```python
from random import random

rands = (random() for _ in range(N))
n1 = sum(1 for r in rands if r < p1)
```

random is a standard library module
NumPy vs. Pure Python

%timeit simulation(N=1000, s=0.1)
%timeit simulation(N=1000000, s=0.01)

Pure Python version:
100 loops, best of 3: 6.42 ms per loop
1 loop, best of 3: 528 ms per loop

NumPy version:
10000 loops, best of 3: 150 μs per loop x42 faster
1000 loops, best of 3: 313 μs per loop x1680 faster!
Can we do it better faster?

Malene Thyssen
• Optimizing compiler
• Declare the **static type** of variables
• Makes *writing C extensions* for Python as easy as Python itself
• Foreign function interface for invoking C/C++ routines

http://cython.org
def simulation(np.uint64_t N,
               np.float64_t s):
    cdef np.uint64_t n1 = 1
    cdef np.uint64_t n0
    cdef np.float64_t p

    while 0 < n1 < N:
        n0 = N - n1
        p1 = n1 * (1 + s) / (n0 + n1 * (1 + s))
        n1 = np.random.binomial(N, p1)

    return n1 == N
%timeit simulation(N=1000, s=0.1)
%timeit simulation(N=10000000, s=0.01)

Cython vs. NumPy:
10000 loops, best of 3: 87.8 µs per loop x2 faster
10000 loops, best of 3: 177 µs per loop x1.75 faster
To approximate the fixation probability we need to run many simulations. Thousands.

In principle, the standard error of our approximation decreases with the square root of the number of simulations: SEM $\sim 1/\sqrt{m}$
Multiple simulations: for loop

```python
fixations = [
    simulation(N, s)
    for _ in range(1000)
]
```
Multiple simulations: for loop

```python
fixations = [
    simulation(N, s)
    for _ in range(1000)
]

fixations

[False, True, False, ..., False, False]

sum(fixations) / len(fixations)

0.195
Multiple simulations: for loop

```
%%timeit
fixations = [
    simulation(N, s)
    for _ in range(1000)
]
```

1 loop, best of 3: **8.05 s** per loop
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([[True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N

Initialize multiple simulations
Multiple simulations: NumPy

```python
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([[True] * repetitions)

    while update.any):
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N
```

Natural selection:

```
n1 is an array so operations are element-wise
```
Multiple simulations: NumPy

```python
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([[True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N
```

Genetic drift: p1 is an array so binomial(N, p1) draws from multiple distributions
Multiple simulations: NumPy

def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([[True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N

update follows the simulations that didn’t finish yet
Multiple simulations: NumPy

def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([[True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N

update follows the simulations that didn’t finish yet
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    update = np.array([True] * repetitions)

    while update.any():
        p1 = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p1[update])
        update = (n1 > 0) & (n1 < N)

    return n1 == N

result is array of Booleans: for each simulation, did variant 1 fix?
Multiple simulations: NumPy

```python
%timeit simulation(N=1000, s=0.1)
10 loops, best of 3: 25.2 ms per loop
```
x320 faster
Fixation probability as a function of $N$

$$N_{\text{range}} = \text{np.logspace}(1, 6, 20, \text{dtype=\text{np.uint64}})$$

$N$ must be an integer for this to evaluate to True:

$$(n1 > 0) \& (n1 < N)$$
Fixation probability as a function of $N$

```python
fixations = [
    simulation(
        N,
        s,
        repetitions
    ) for N in Nrange
]
```
Fixation probability as a function of $N$

```python
fixations = np.array(fixations)

fixations

array([[False, False, ..., False, False],
       [False, True, ..., False, False],
       ..., 
       [False, False, ..., True, False],
       [False, False, ..., False, False]],
      dtype=bool)
```
Fixation probability as a function of $N$

```python
fixations = np.array(fixations)
mean = fixations.mean(axis=1)
sem = fixations.std(
    axis=1,
    ddof=1
) / np.sqrt(repetitions)
```
Plotting with matplotlib
Approximation

Kimura’s equation:
\[ \frac{e^{-2s} - 1}{e^{-2Ns} - 1} \]

def \textbf{kimura}(N, s):
    return np.expm1(-2 * s) / np.expm1(-2 * N * s)

\textit{expm1}(x) is \( e^x - 1 \) with better precision for small values of \( x \)
kimura works on arrays out-of-the-box

```python
%timeit [kimura(N=N, s=s)
           for N in Nrange]
%timeit kimura(N=Nrange, s=s)
```

1 loop, best of 3: **752 ms** per loop
1000 loops, best of 3: **3.91 ms** per loop

**X200 faster!**
Numexpr

Fast evaluation of element-wise array expressions using a vector-based virtual machine.

def kimura(N, s):
    return numexpr.evaluate("expm1(-2 * s) / expm1(-2 * N * s)"")

%timeit kimura(N=Nrange, s=s)
1000 loops, best of 3: 803 µs per loop x5 faster

https://github.com/pydata/numexpr
Plotting with matplotlib

http://matplotlib.org
Finding the expected **fixation time**:

**SciPy** for numerical integration, **Pandas** and **Seaborn** for statistical analysis and visualization.

![Graph showing fixation time distribution and population size relationship](image)
Dig Deeper

Online Jupyter notebook: [github.com/yoavram/PyConIL2016](https://github.com/yoavram/PyConIL2016)

Multi-type simulation:
Includes $L$ variants, with mutation.
Follow $n_0, n_1, ..., n_L$ until $n_L = N$. 
Dig Deeper

Online Jupyter notebook: github.com/yoavram/PyConIL2016

- **Numba**: JIT compiler, array-oriented and math-heavy Python syntax to machine code
- **IPyParallel**: IPython’s sophisticated and powerful architecture for parallel and distributed computing.
- **IPyWidgets**: Interactive HTML Widgets for Jupyter notebooks and the IPython kernel
Thank You!

Presentation, Jupyter notebook, and more at
[github.com/yoadavram/PyConIL2016](https://github.com/yoadavram/PyConIL2016)

- yoav@yoadavram.com
- @yoadavram
- [github.com/yoadavram](https://github.com/yoadavram)
- [www.yoadavram.com](http://www.yoadavram.com)
- [python.yoadavram.com](http://python.yoadavram.com)