

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

Why Scientific Python

(why not MATLAB?)

Python is...

- **Free as in beer**
- **Free as in speech**
- **General purpose**
- **Large, diverse, active community**
- Beautiful design
- Portable
- Easy to learn, fast to develop
- Fast enough
- Many libraries

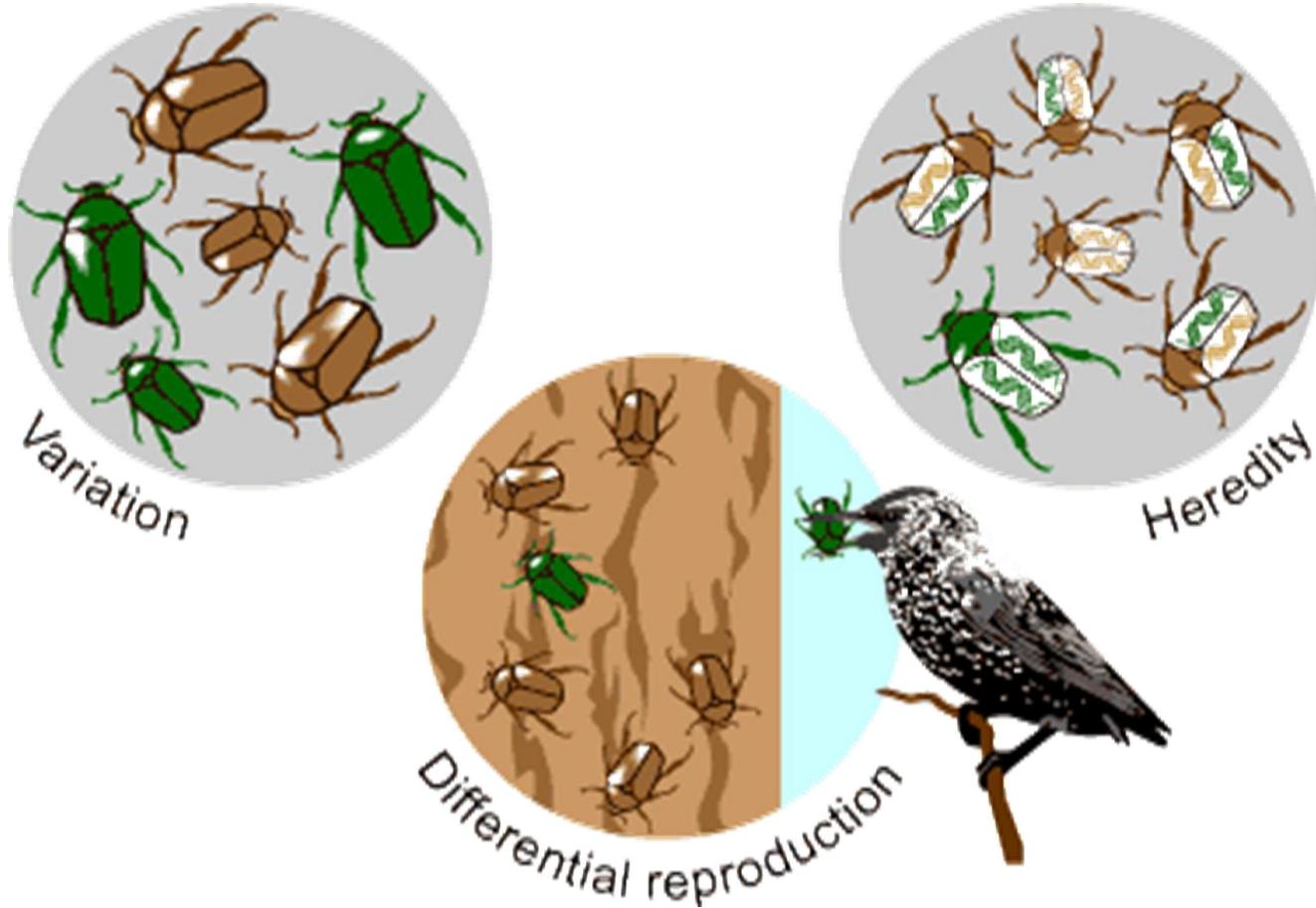
[I used Matlab. Now I use Python. by Steve Tjoa](#)
[Why use Python for scientific computing?](#)
[Introduction to Python for MATLAB users](#)

Theoretical Evolution

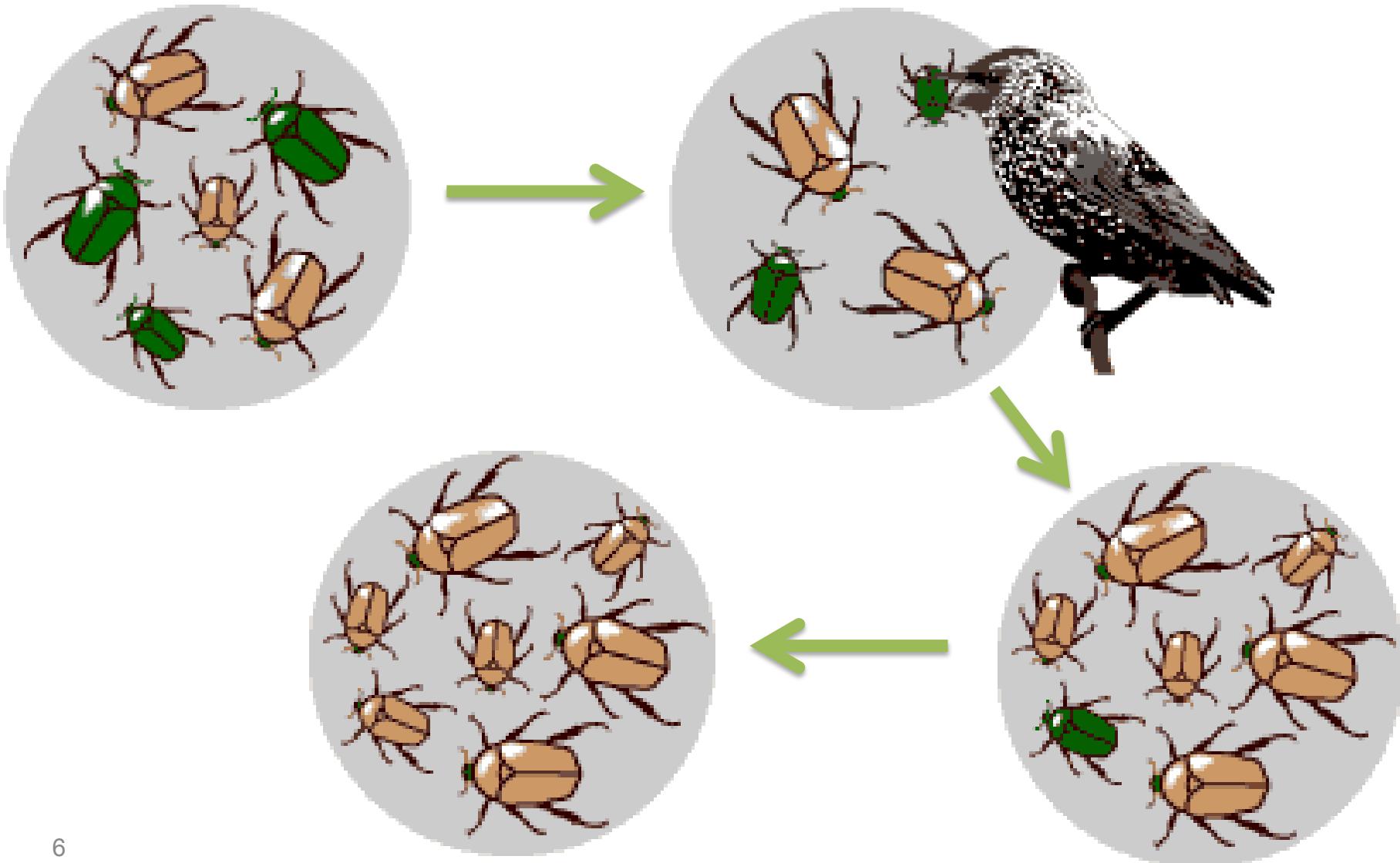
Formally: Population genetics

- 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

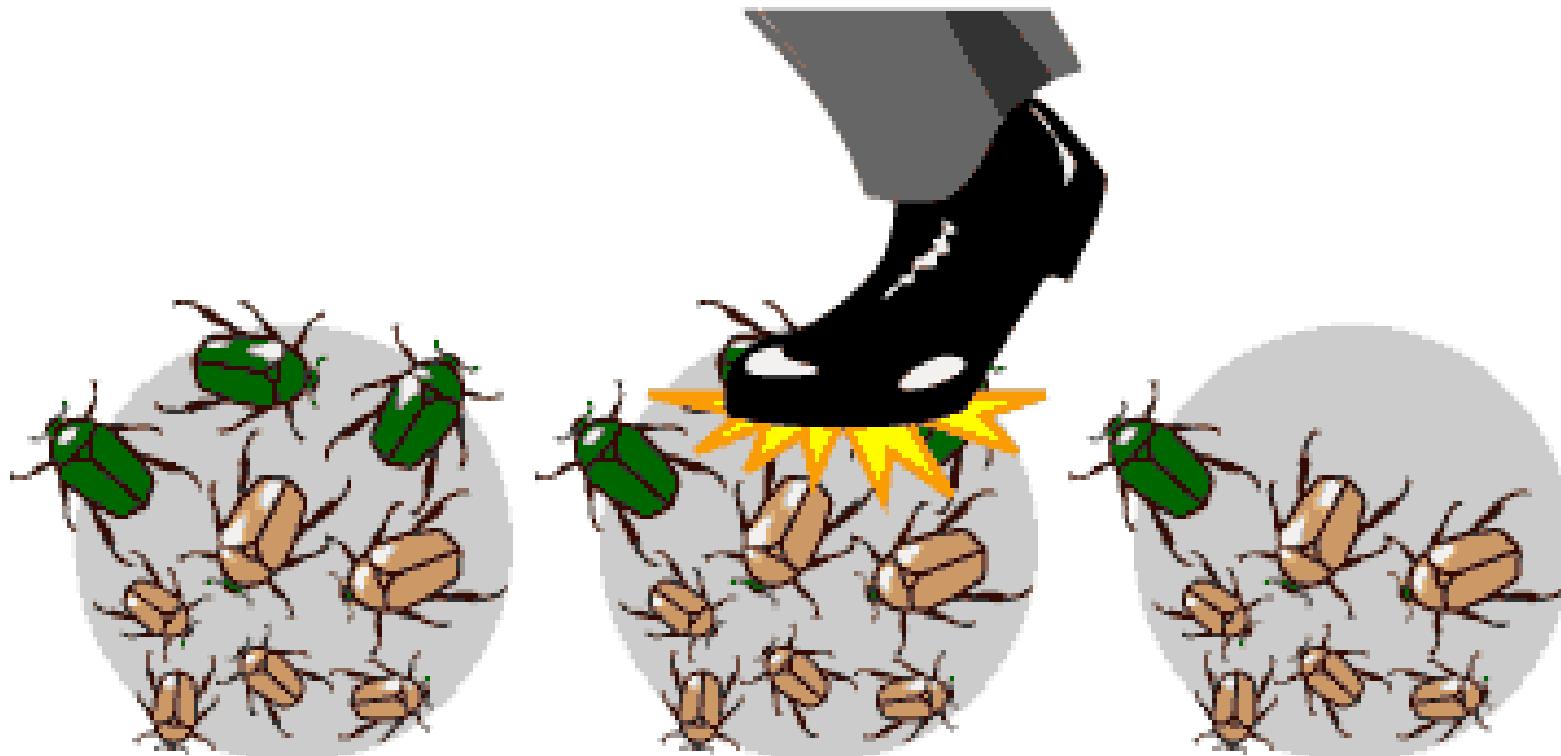
Evolution



Natural Selection

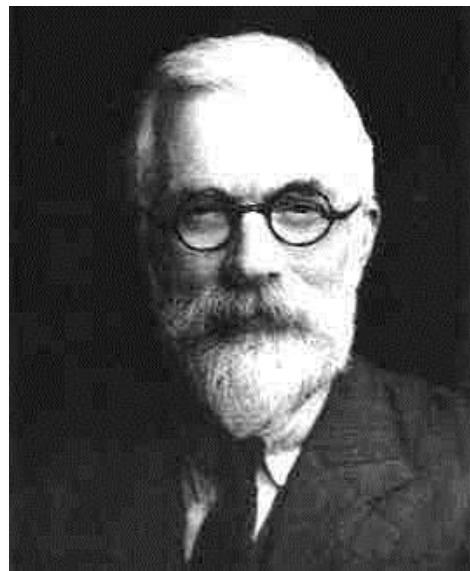


Random Genetic Drift



The Wright-Fisher Model

Standard model for change in frequency of gene variants.



R.A. Fisher
1890-1962
UK & Australia



Sewall Wright
1889-1988
USA

The 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 **$p_1=n_1/N$** .

The 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 (**p₁**) 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**.

The 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 **p₁**.

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**:

- Powerful N-dimensional array object
- **Random number generators**
- Array functions & broadcasting
- Tools for integrating C/C++ and Fortran code
- Linear algebra
- Fourier transform

numpy.org

Into the code

Death to the Stock Photo



Natural Selection

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

Random drift

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

```
from numpy.random import binomial
```

```
n1 = 1
```

Import a binomial random number generator from NumPy

```
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

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

```
from numpy.random import binomial
```

```
n1 = 1
```

Start with a single copy of variant 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

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

```
from numpy.random import binomial
```

```
n1 = 1
```

Until number of individuals
with variant 1 is 0 or N:
extinction or fixation

```
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

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

```
from numpy.random import binomial
```

```
n1 = 1
```

The frequency of variant 1
after selection is p_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

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

```
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)
```

Due to genetic drift, the number of individuals with variant 1 in the next generation is n1

```
fixation = n1 == N
```

Natural Selection

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

Random drift

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

```
from numpy.random import binomial
```

```
n1 = 1
```

Fixation: n_1 equals N
Extinction: n_1 equals 0

```
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

Here, **NumPy** is useful for random number generation:

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

Pure Python version would replace this with:

```
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!

A cheetah is shown in mid-stride, running across a grassy field. The cheetah's body is low to the ground, and its long tail is held high. The background is a blurred green field, suggesting motion. A dark rectangular box covers the lower-left portion of the image, containing white text.

Can we do it
better **faster**?



- **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
```



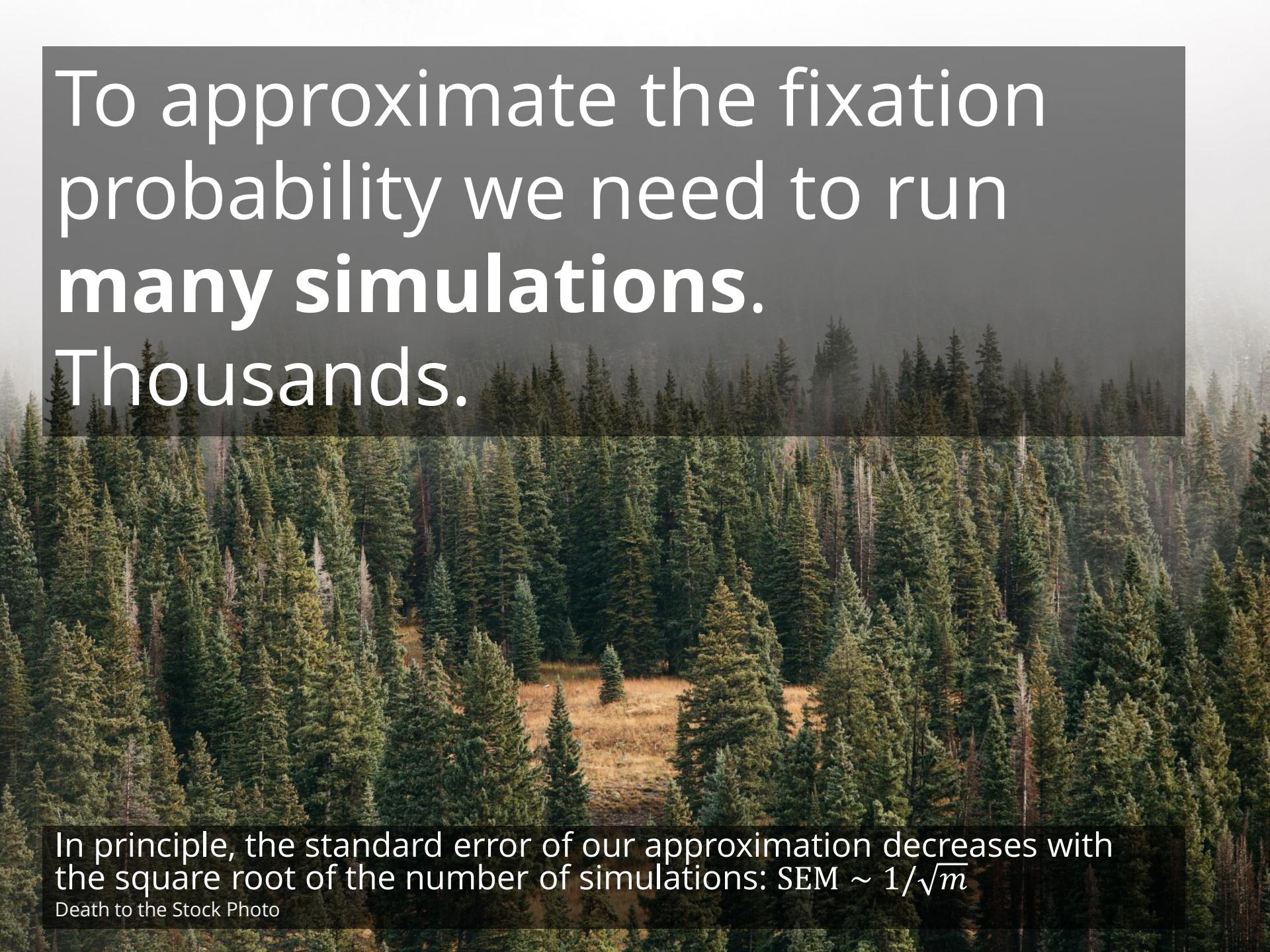
```
%timeit simulation(N=1000, s=0.1)
%timeit simulation(N=1000000, s=0.01)
```

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: $SEM \sim 1/\sqrt{m}$

Death to the Stock Photo

Multiple simulations: for loop

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

Multiple simulations: for loop

```
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

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
```

Initialize multiple simulations

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
```

Natural selection:
n1 is an array so operations are element-wise

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
```

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

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
```

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

Multiple simulations: NumPy

```
%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

```
Nrange = np.logspace(1, 6, 20,  
                      dtype=np.uint64)
```

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

(n1 > 0) & (n1 < N)

Fixation probability as a function of N

```
fixations = [  
    simulation(  
        N,  
        s,  
        repetitions  
    ) for N in Nrange  
]
```

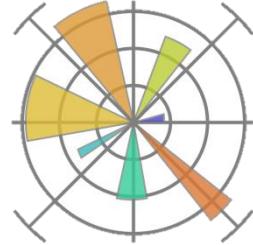
Fixation probability as a function of N

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

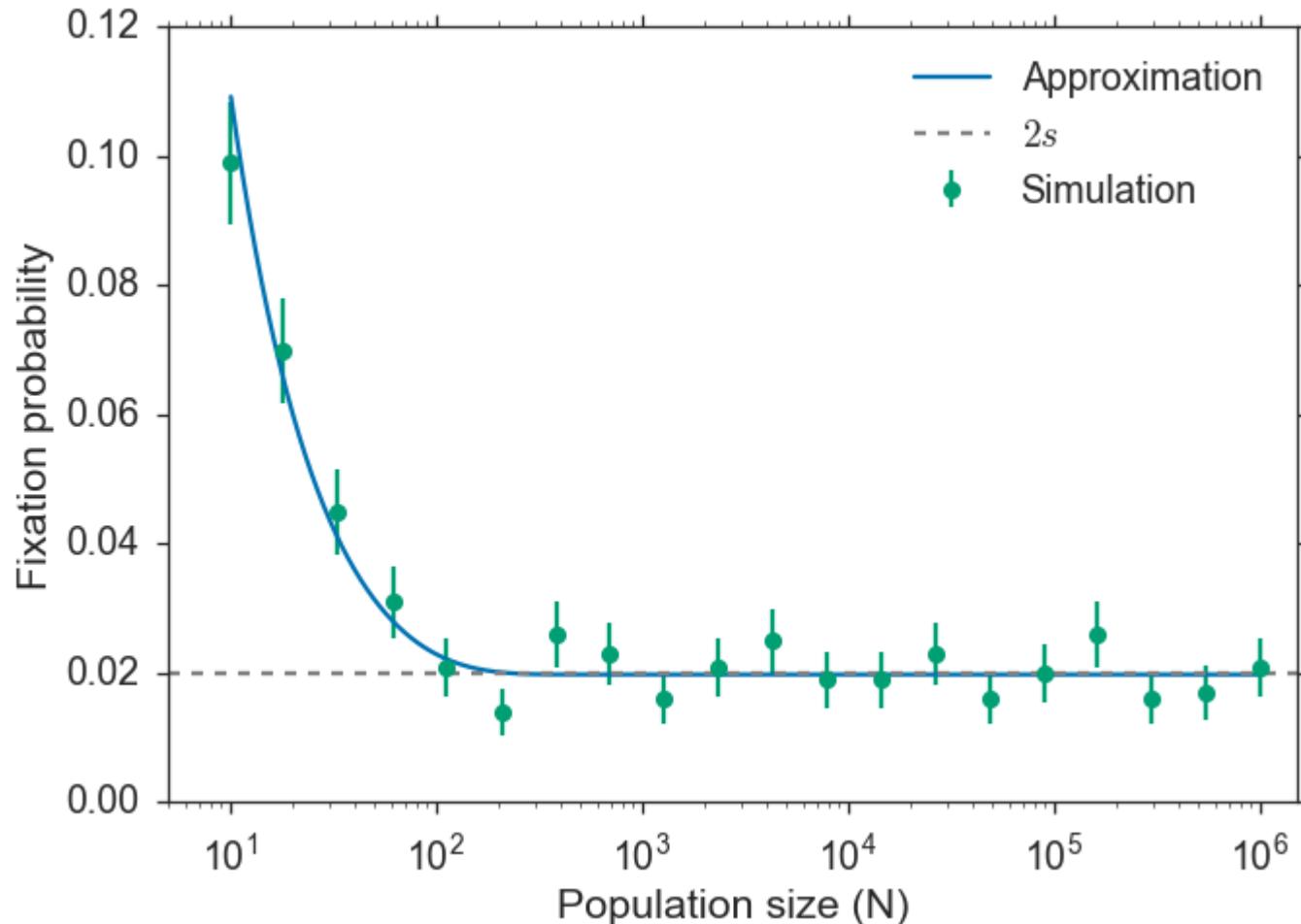
```
array([[False, False, ..., False, False],  
       [False, True, ..., False, False],  
       , ...,  
       [False, False, ..., True, False],  
       [False, False, ..., False, False]],  
      dtype=bool)
```

Fixation probability as a function of N

```
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:

$$\frac{e^{-2s} - 1}{e^{-2Ns} - 1}$$



Motoo Kimura
1924-1994
Japan & USA

```
def kimura(N, s):  
    return np.expm1(-2 * s) /  
          np.expm1(-2 * N * s)
```

expm1(x) is **e^x-1** with better precision for small values of x

kimura works on arrays out-of-the-box

```
%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!

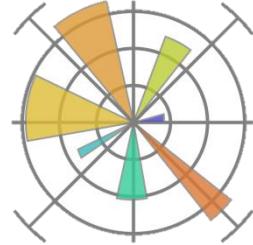
Numexpr

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

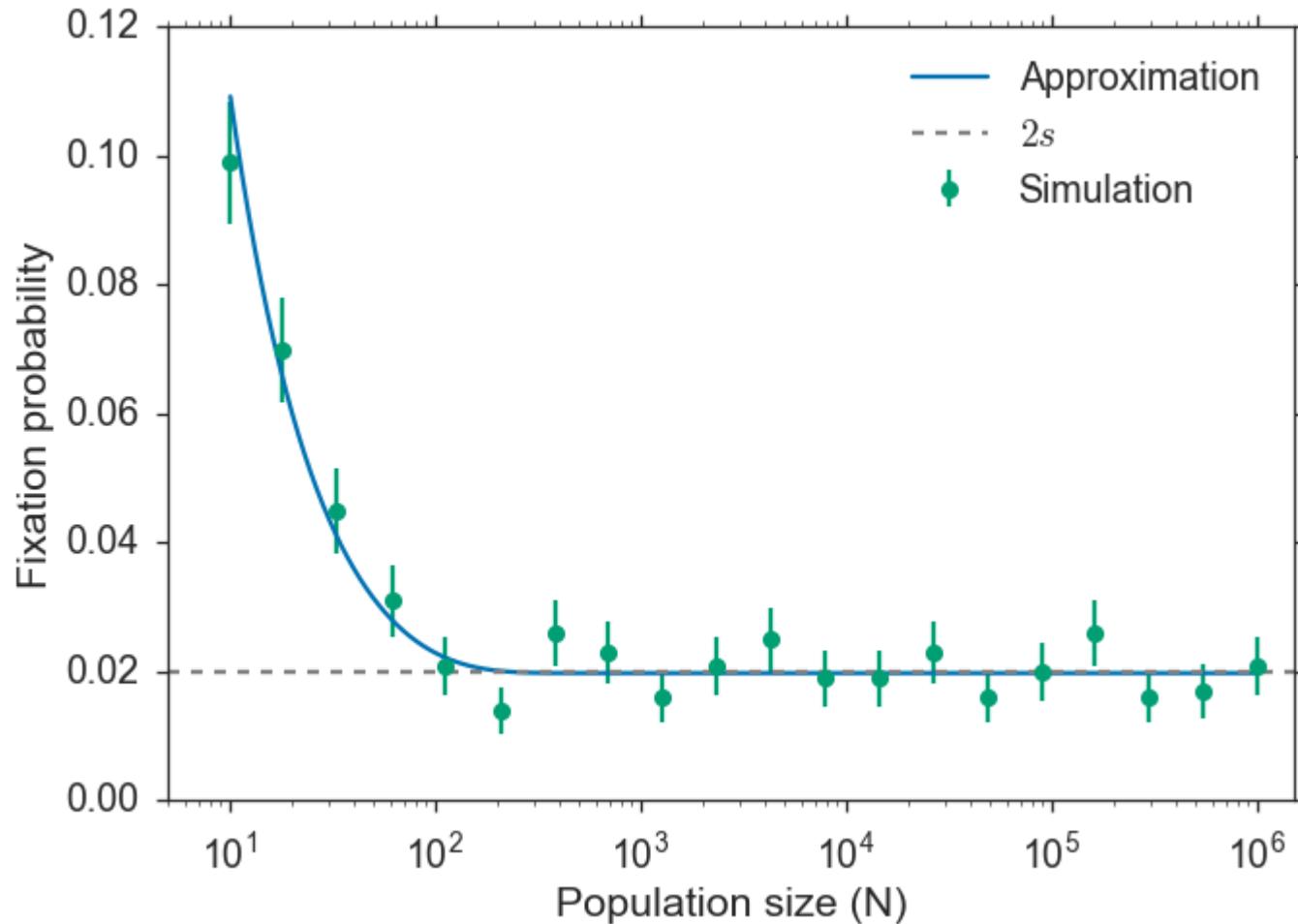
```
def kimura(N, s):  
    return numexpr.evaluate(  
        "expm1(-2 * s) /  
        expm1(-2 * N * s)")
```

```
%timeit kimura(N=Nrange, s=s)
```

1000 loops, best of 3: **803 µs** per loop **x5 faster**



Plotting with matplotlib



Fixation time

How much time does it take variant 1 to go **extinct or to fix?**

We want to keep track of **time***.

time is measured in **number of generations**

https://commons.wikimedia.org/wiki/File:Prim_clockwork.jpg

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    T = np.empty_like(n1)
    update = (n1 > 0) & (n1 < N)
    t = 0
    while update.any():
        t += 1
        p = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p[update])
        T[update] = t
        update = (n1 > 0) & (n1 < N)

    return n1 == N, T
```

t keeps track of time

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    T = np.empty_like(n1)
    update = (n1 > 0) & (n1 < N)
    t = 0
    while update.any():
        t += 1
        p = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p[update])
        T[update] = t
        update = (n1 > 0) & (n1 < N)

    return n1 == N, T
```

T holds time for extinction/fixation

```
def simulation(N, s, repetitions):
    n1 = np.ones(repetitions)
    T = np.empty_like(n1)
    update = (n1 > 0) & (n1 < N)
    t = 0
    while update.any():
        t += 1
        p = n1 * (1 + s) / (N + n1 * s)
        n1[update] = binomial(N, p[update])
        T[update] = t
        update = (n1 > 0) & (n1 < N)

    return n1 == N, T
```

**Return both Booleans
and times (T)**

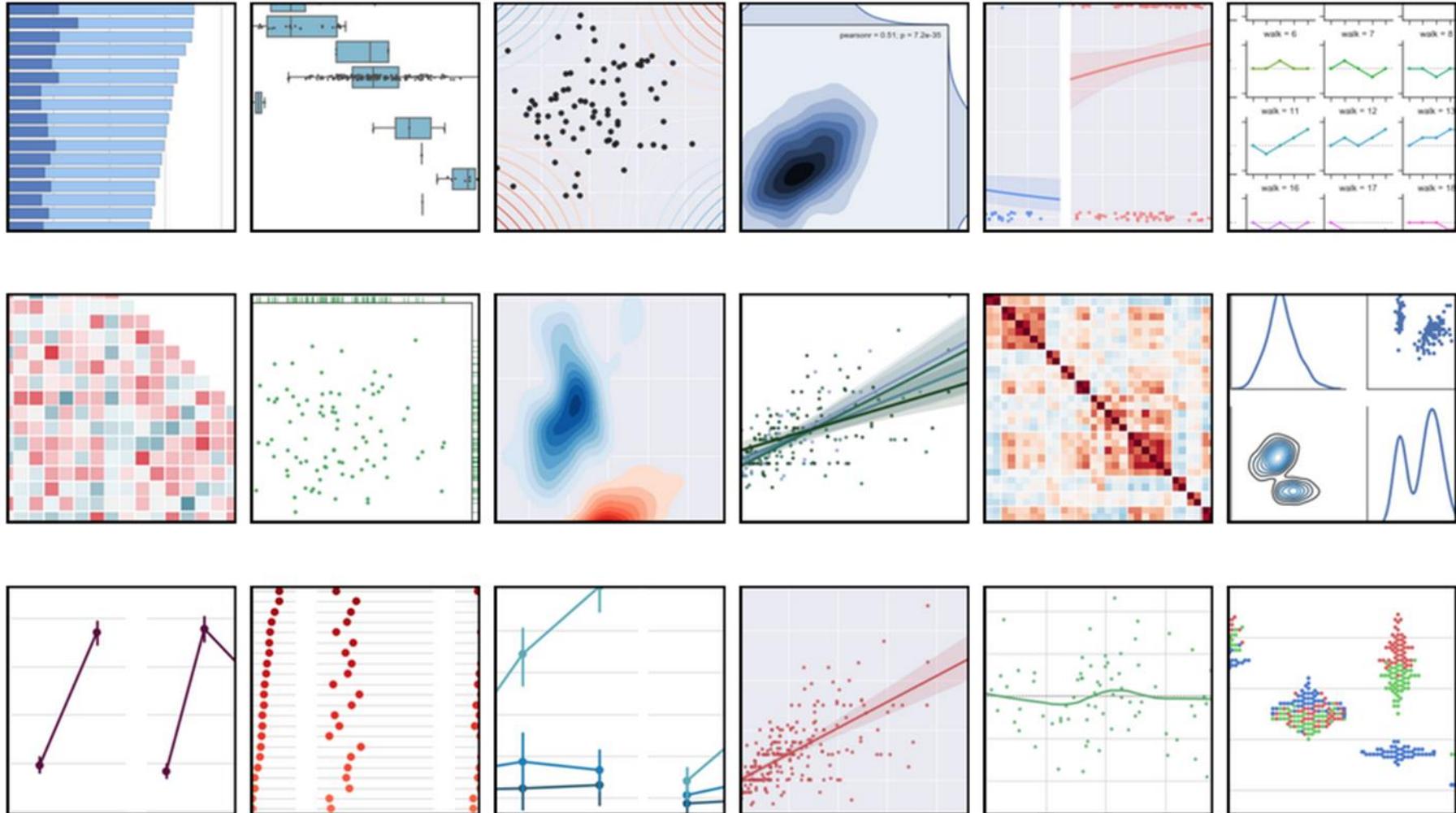
Statistical data visualization with Seaborn

- Visualization library based on **matplotlib** and **Pandas**
- High-level interface for attractive **statistical graphics**

By Michael Waskom, Stanford Ph.D. student

<http://stanford.edu/~mwaskom/software/seaborn>

Statistical data visualization with Seaborn



Plot with Seaborn

```
from seaborn import distplot  
fixations, times = simulation()  
  
distplot(times[fixations])  
  
distplot(times[~fixations])
```

Plot with Seaborn

```
from seaborn import distplot  
  
fixations, times = simulation()  
  
distplot(times[fixations])  
  
distplot(times[~fixations])
```

Plot with Seaborn

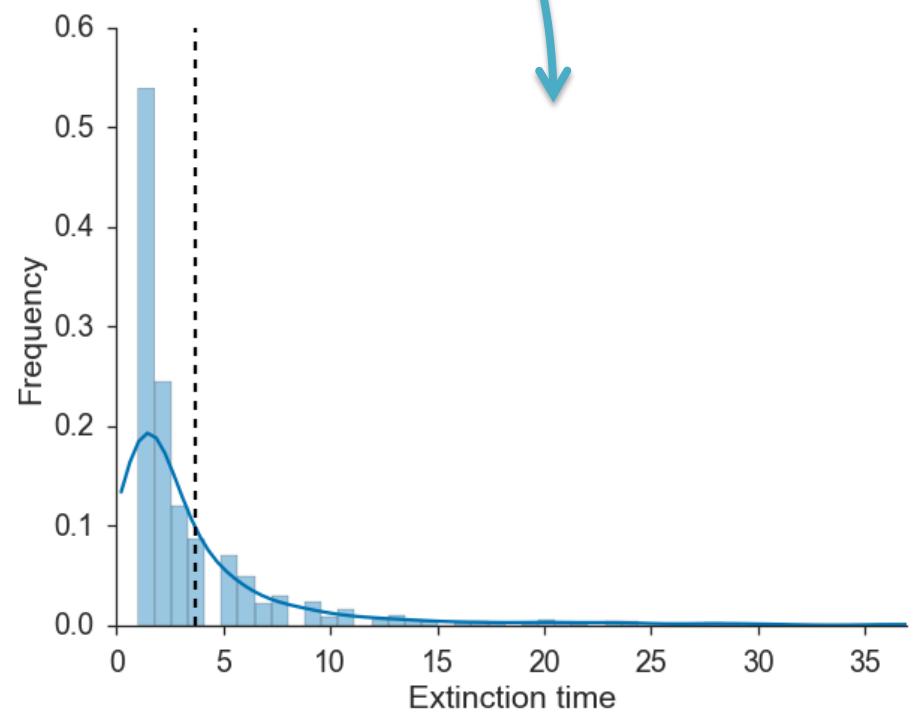
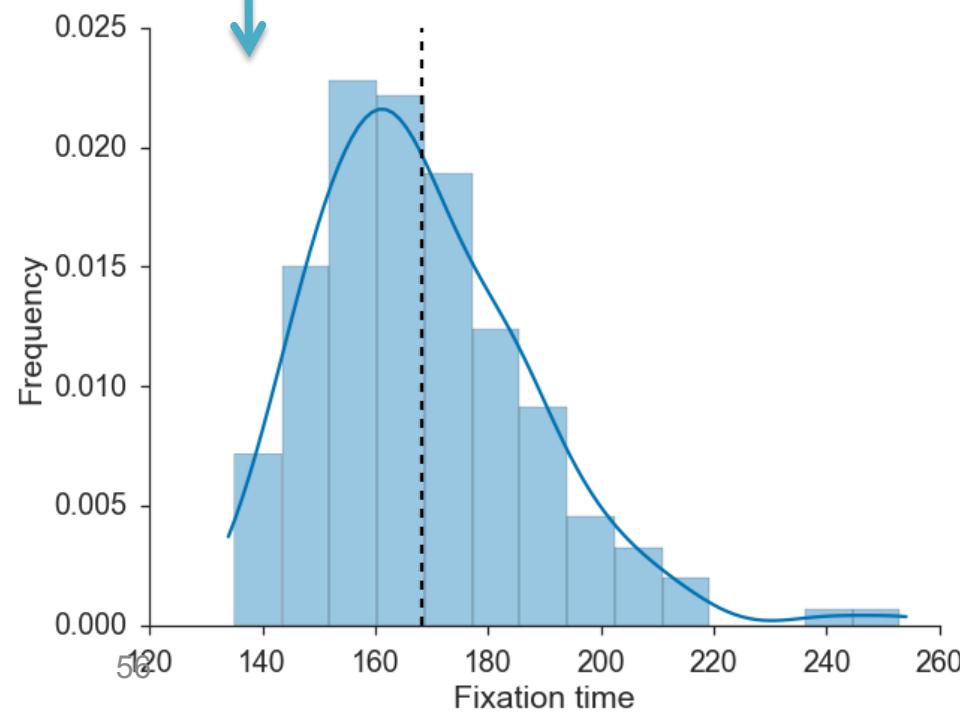
```
from seaborn import distplot  
  
fixations, times = simulation()  
  
distplot(times[fixations])  
  
distplot(times[~fixations])
```

Plot with Seaborn

```
fixations, times = simulation(...)
```

```
distplot(times[fixations])
```

```
distplot(times[~fixations])
```



Diffusion equation approximation

$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1 - e^{-2Ns})} \int_x^1 I_1(y) dy$$

$$J_2 = \frac{1}{s(1 - e^{-2Ns})} \int_0^x I_2(y) dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$T_{fix} = J_1 + \frac{1 - u}{u} J_2$$



Motoo Kimura
1924-1994
Japan & USA

Diffusion equation approximation

$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1 - e^{-2Ns})} \int_x^1 I_1(y) dy$$

$$J_2 = \frac{1}{s(1 - e^{-2Ns})} \int_0^x I_2(y) dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$T_{fix} = J_1 + \frac{1 - u}{u} J_2$$

Requires
integration...



Motoo Kimura
1924-1994
Japan & USA

```
from functools import partial
from scipy.integrate import quad

def integral(f, N, s, a, b):
    f = partial(f, N, s)
    return quad(f, a, b)[0]
```

integral will calculate $\int_a^b f(N, s, x) dx$

```
from functools import partial
from scipy.integrate import quad

def integral(f, N, s, a, b):
    f = partial(f, N, s)
    return quad(f, a, b)[0]
```

partial freezes N and s
in $f(N, s, x)$ to create $f(x)$

```
from functools import partial
from scipy.integrate import quad

def integral(f, N, s, a, b):
    f = partial(f, N, s)
    return quad(f, a, b)[0]
```

SciPy's quad computes a definite integral $\int_a^b f(x)dx$
(using a technique from the Fortran library QUADPACK)

```

def I1(N, s, x):
    ...
def I2(N, s, x):
    ...

```

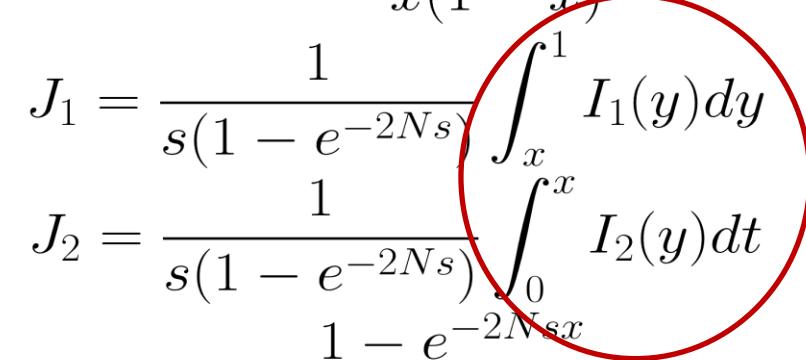
$$I_1(x) = \frac{1 - e^{-2Nsx} - e^{-2Ns(1-x)} + e^{-2Ns}}{x(1-x)}$$

$$I_2(x) = \frac{(e^{2Nsx} - 1)(1 - e^{-2Ns(1-x)})}{x(1-x)}$$

$$J_1 = \frac{1}{s(1 - e^{-2Ns})} \int_x^1 I_1(y) dy$$

$$J_2 = \frac{1}{s(1 - e^{-2Ns})} \int_0^x I_2(y) dt$$

$$u = \frac{1 - e^{-2Nsx}}{1 - e^{-2Ns}}$$

$$T_{fix} = J_1 + \frac{1 - u}{u} J_2$$


I1 and I2 are defined according to the equations

```

@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2

```

T_kimura is the fixation time given a single copy of variant 1: frequency $x=1/N$

```

@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2

```

J1 and J2 are calculated using integrals of I1 and I2

```

@np.vectorize
def T_kimura(N, s):
    x = 1.0 / N
    J1 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I1, N, s, x, 1)
    J2 = -1.0 / (s * expm1(-2 * N * s)) *
          integral(I2, N, s, 0, x)
    u = expm1(-2 * N * s * x) /
        expm1(-2 * N * s)

    return J1 + ((1 - u) / u) * J2

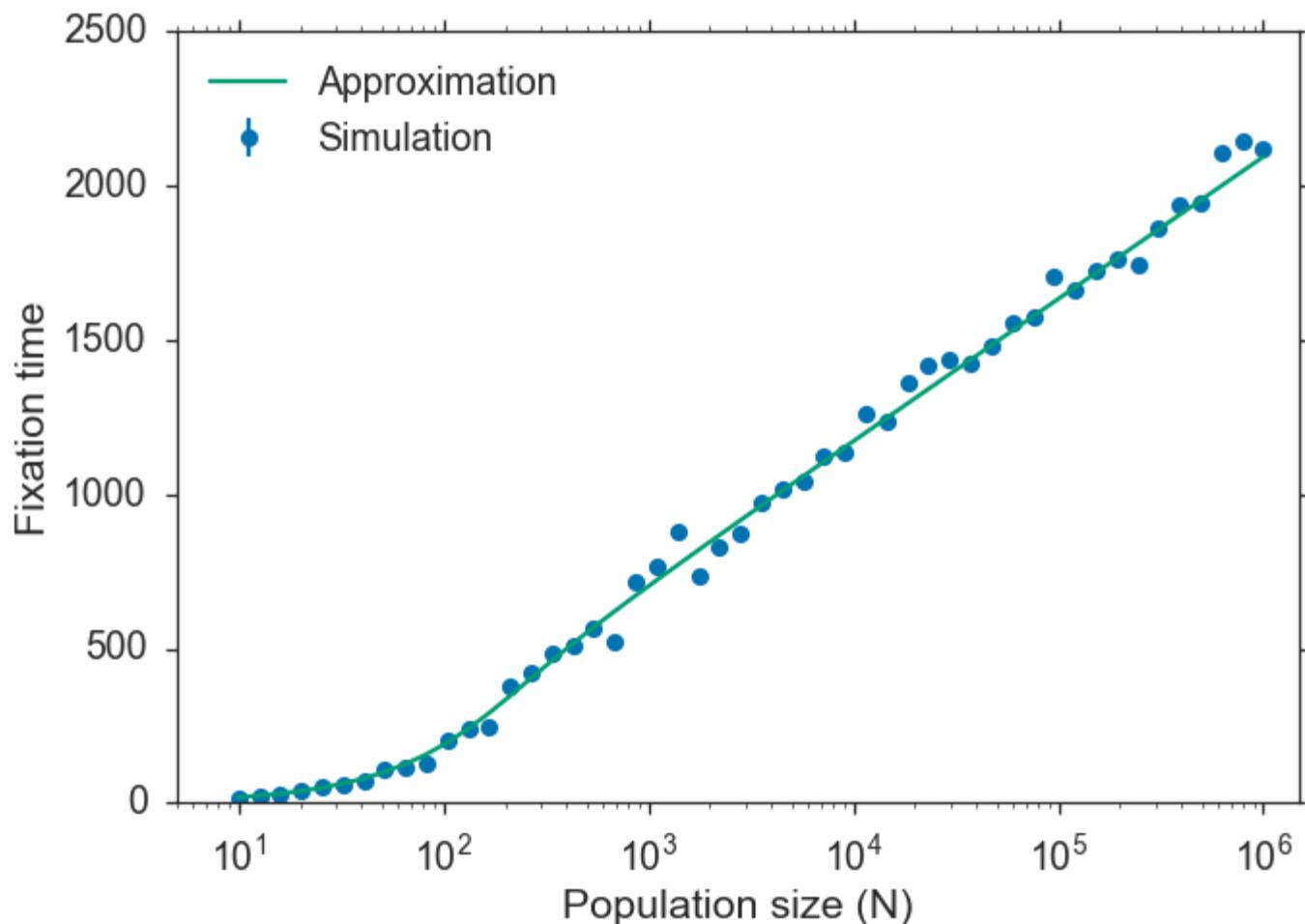
```

T_{fix} is the return value

```
@np.vectorize
```

```
def T_kimura(N, s):  
    x = 1.0 / N  
    J1 = -1.0 / (s * expm1(-2 * N * s)) *  
         integral(I1, N, s, x, 1)  
    J2 = -1.0 / (s * expm1(-2 * N * s)) *  
         integral(I2, N, s, 0, x)  
    u = expm1(-2 * N * s * x) /  
        expm1(-2 * N * s)  
  
    return J1 + ((1 - u) / u) * J2
```

np.vectorize **creates a function that takes a sequence and returns an array - x2 faster**

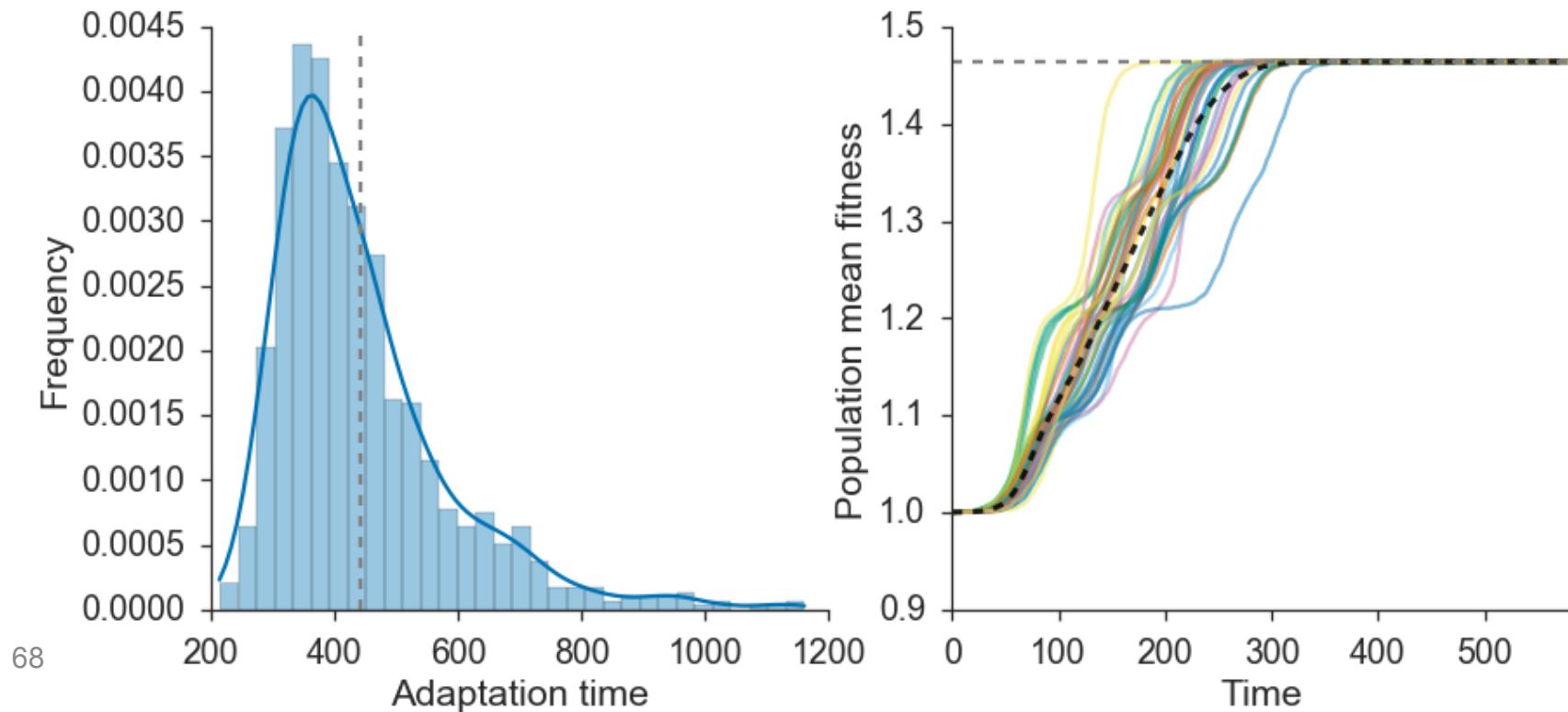


Dig Deeper

Online **Jupyter** notebook: github.com/yoavram/PyConIL2016

More **NumPy** - multi-type simulation:
Includes **L** variants, with mutation.

Follow n_0, n_1, \dots, n_L until $n_L=N$.



Dig Deeper

Online Jupyter notebook: github.com/yoavram/PyConIL2016

- [**Pandas**](#): high-performance, easy-to-use data structures and **data analysis tools**
- [**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/yoavram/PyConIL2016



- ✉ yoav@yoavram.com
- 🐦 [@yoavram](https://twitter.com/yoavram)
- 🐱 github.com/yoavram
- 🏡 www.yoavram.com
- 🐍 python.yoavram.com