Fixing with a fat tail

Bjarki Eldon[1 2]

Abstract

This code generates excursions of the evolution of a haploid population partitioned into two genetic types, with viability weight determined by $W = e^{-s(g-g_0)^2}$, where g is the genetic type of a given individual, and g_0 is the optimal type, and $s > 0$ is the strength of selection. The population evolves according to a model of random sweepstakes and viability selection and randomly occurring bottlenecks. We estimate the probability of fixation of the type conferring advantage, and the expected time to fixation conditional on fixation of the advantageous type.

Contents

[1]MfN Berlin, Germany

[2]Supported by Deutsche Forschungsgemeinschaft (DFG) - Projektnummer 273887127 through DFG SPP 1819: Rapid Evolutionary Adaptation grant STE 325/17-2 to Wolfgang Stephan; acknowledge funding by the Icelandic Centre of Research through an Icelandic Research Fund Grant of Excellence no. 185151-051 to Einar Árnason, Katrín Halldórsdóttir, Alison M. Etheridge, WS, and BE. BE also acknowledges Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.
January 23, 2022

1 Copyright

2 Compilation, output and execution

This CWEB [4] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c` [3] file.

One can use `cweave` to generate a `.tex` file, and `ctangle` to generate a `.c` file. To compile the C++ code (the `.c` file), one needs the GNU Scientific Library, and the C++ boost library. Using a Makefile can be helpful, calling this file `iguana.w`

```
iguana.pdf :  iguana.tex
      cweave iguana.w
      pdflatex iguana
      bibtex iguana
      pdflatex iguana
      pdflatex iguana
      ctangle iguana
      c++ -Wall -Wextra -pedantic -O3 -march=native -m64 iguana.c -lm -lgsl
-lgslcblas

clean :
      rm -vf iguana.c iguana.tex
```

Use `valgrind` to check for memory leaks:

```
valgrind -v -leak-check=full -show-leak-kinds=all <program call>
```

3 introduction

We consider a haploid population of fixed size N. Let $X^N, X_1^N, \ldots, X_N^N$ be i.i.d. discrete random variables taking values in $\{1, \ldots, \Psi_N\}$; the $X_1^N, \ldots, X_N^N$ denote the random number of juveniles independently produced in a given generation according to

$$\mathbb{P}\left(X^N = k\right) = \frac{(\Psi_N + 1)^\alpha}{(\Psi_N + 1)^\alpha - 1}\left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha}\right), \quad 1 \le k \le \Psi_N. \tag{1}$$

The mass in Eq (1) is normalised so that $\mathbb{P}\left(1 \le X^N \le \Psi_N\right) = 1$, and $\mathbb{P}\left(X^N = k\right) \ge \mathbb{P}\left(X^N = k+1\right)$. Given a pool of at least N juveniles, we sample N juveniles for the next generation. Leaving out an atom at zero gives $X_1^N + \cdots + X_N^N \ge N$ almost surely, guaranteeing that we always have at least N juveniles to choose from in each generation.

Write $X_1 \sim L(\alpha, \Psi_N)$ if X_1 is distributed according to Eq (1) for given values of α and Ψ_N. Let $1 < \alpha_1 < 2$ and $\alpha_2 > 2$ be fixed and consider the mixture distribution [2]

$$X_1, \ldots, X_N \sim \begin{cases} L(\alpha_1, \Psi_N) & \text{with probability } \varepsilon_N, \\ L(\alpha_2, \Psi_N) & \text{with probability } 1 - \varepsilon_N. \end{cases} \tag{2}$$

Similarly, by identifying the appropriate scaling of ε_N one can keep α fixed and varied Ψ_N [1].

Each juvenile inherits the type of its' parent, and is assigned a viability weight z accoring to the type; the wild type is assigned the weight $z = e^{-s}$ for some fixed $s > 0$, and advantageous type the weight one. For each juvenile we sample an exponential with rate the given viability weight, and the N juveniles with the smallest exponential replace the parents. In any given generation a bottleneck of a fixed size N_b occurs with a fixed probability. If a bottleneck occurs we sample N_b individuals independently and uniformly at random without replacement. The surviving individuals then produce juveniles, and if the total number of juveniles is less than the capacity N all the juveniles survive, otherwise we assign weights and sample N juveniles according to the weights.

Let $Y_t \equiv \{Y_t : t \ge 0\}$ denote the frequency of the type conferring selective advantage, and write $T_k(y) := \min\{t \geq 0 : Y_t = k, Y_0 = y\}$. We are interested in the quantitites

$$\begin{aligned} p_N &:= \mathbb{P}\left(T_N(1) < T_0(1)\right) \\ \tau_N &:= \mathbb{E}\left[T_N(1) : T_N(1) < T_0(1)\right] \end{aligned} \tag{3}$$

4 Code

We collect the key containers and constants into a struct § 4.2, we use the GSL random number generator § 4.3, in § 4.4 we compute the cumulative density function for sampling random numbers of juveniles according to the inverse CDF method, in § 4.5 we sample a random number of juveniles, in § 4.6 we define a comparison function for sorting the exponentials in § 4.7, in § 4.8 we sample a pool of juveniles and assign weight to them in § 4.9, in § 4.11 we sample the number of individuals of the advantageous type surviving a bottleneck, in § 4.12 we count the number of advantageous type surviving selection according to their weight, in § 4.13 we step through one generation by checking if a bottleneck occurs and then produce juveniles if neither fixation nor loss of the advantageous type occurs, in § 4.14 we generate one excursion until fixation or loss of the advantageous type starting with one copy of the advantageous type, the main module § 4.15 generates a given number of trajectories, § 5 holds examples of trajectories to fixation of the advantageous type.

4.1 Includes

The included libraries.

5 ⟨includes 5⟩ ≡

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <list>
#include <string>
#include <fstream>
#include <chrono>
#include <forward_list>
#include <assert.h>
#include <math.h>
#include <unistd.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

This code is used in chunk 19.

4.2 the data struct

The data structure collecting the main containers and the constants; we start with the type conferring advantage in one copy, unless otherwise stated.

6 ⟨ structM 6 ⟩ ≡

```
struct M { const size_t N = 1000000;
    /* keep alpha_one less than or equal to alpha_two
     */
const double alpha_one = 0.75;
const double alpha_two = 3.0;     /* psi is the cutoff Ψ_N
     */
const size_t cutoff = N;
const size_t bottlenecksize = 100;
const double pbottle = 0.1;
    /* psi is psione and psione / N to 0; need pN = (psione/N)^{2-alpha} */
const double psi = static_cast<double>(cutoff);
const double psitwo = static_cast<double>(cutoff);
    /* c_strength_selection the strength of selection s > 0
     */
const double c_strength_selection = 0.5;
    /* for cutoffs need pN as epsilonN and pN = (psione/N)^{2-alpha}
     */    /* epsilon_N is the probability of producing juveniles with alpha_one or
     psitwo; consider as ε_N = c/N where c_c is c > 0 a constant
     */    /* for cutoffs need pN as epsilonN and pN = (psione/N)^{2-alpha} */
const double epsilon_N = 0.1;     /* 1./static_cast<double>(N) ;
     */    /* probability of psi is one minus epsilon; probability of psi2 is epsilon */
const int number_experiments = 1000;
const std::string c_configuration = "_tablen";
const std::string c_excursion_skra = "excursion_skra_" + c_configuration + "_.txt";
```

```
size_t SN = 0;
size_t SNW = 0;    /* initial number of copies of advantageous type
    */
size_t y = 1;
size_t Nprime
{ }
;
double Nth
{ }
;    /* CDF using alpha_one
    */
std::vector < double > cdf_one
{ }
;    /* CDF using alpha_two
    */
std::vector < size_t > index
{ }
; std::vector < double > cdf_two
{ }
; std::vector < double > vEall
{ }
; std::vector < double > vE
{ }
; void freemem( ) { index.clear( ); std::vector < size_t > ( ).swap(index);
    vE.clear( ); std::vector < double > ( ).swap(vE);
    vEall.clear( ); std::vector < double > ( ).swap(vEall);
    cdf_two.clear( ); std::vector < double > ( ).swap(cdf_two);
```

```
cdf_one.clear(); std::vector<double>().swap(cdf_one); }
  /* compute the mass function in Eq (1)
   */
double masspx(const double c_k, const double c_alpha, const double c_cutoff)
{
  return ((pow(1.0/c_k, c_alpha) - pow(1.0/(c_k + 1.0),
      c_alpha))/(1.0 - pow(1.0/(c_cutoff + 1.0), c_alpha)));
}
} ;
```

This code is used in chunk 19.

4.3 the random number generator

7 ⟨gslrng 7⟩ ≡

```
gsl_rng * rngtype;
static void setup_rng(unsigned long int s)
{
  const gsl_rng_type*T;
  gsl_rng_env_setup();
  T = gsl_rng_default;
  rngtype = gsl_rng_alloc(T);
  gsl_rng_set(rngtype, s);
}
```

This code is used in chunk 19.

4.4 compute the CDF

Compute the cumulative density function for the distribution of juveniles Eq (1) ; used for sampling number of juveniles using the inverse cdf method in §˜??SEC:random$_{n}umber_{j}uveniles$.

8 ⟨ cdf 8 ⟩ ≡

```
static void invcdf(M &self)
{
  self.cdf_one.clear();
  self.cdf_two.clear();
  self.cdf_one.push_back(0.);
  self.cdf_two.push_back(0.);
  double k = 1.;
  while (k ≤ self.psi) {
    self.cdf_two.push_back(self.cdf_two.back() + self.masspx(k, self.alpha_two, self.psi));
    self.cdf_one.push_back(self.cdf_one.back() + self.masspx(k, self.alpha_one, self.psi));
    ++k;
  }
}
```

This code is used in chunk 19.

4.5 sample a random number of juveniles

Sample a random number of juveniles using the inverse-cdf method, i.e. drawing a random uniform and check where it lands, i.e. if F denotes the cumulative density function we compute

$$j = \min\{k \in \mathbb{N} : F(k) > u\} \tag{4}$$

The CDF is computed in § 4.4.

9 ⟨ samplej 9 ⟩ ≡

```
static size_t samplexi(M &self, const size_t one_two)
{    /* we sample at least one juvenile
     */
  size_t j = 1;    /* sample a random uniform
     */
  const double u = gsl_rng_uniform(rngtype);
  ;
  if (one_two < 2) {    /* use CDF for alpha_one
       */
    while (u > self.cdf_one[j]) {
      ++j;
    }
  }
  else {    /* use CDF for alpha_two
       */
    while (u > self.cdf_two[j]) {
      ++j;
    }
  }
  assert(j ≤ static_cast⟨size_t⟩(self.psi));
  return (j);
```

```
}
```

This code is used in chunk 19.

4.6 comparison module for partial sorting

Comparison module for partial sorting of juveniles given their viability weight; used in § 4.7.

10 ⟨comp 10⟩ ≡

```
static bool comp(const double a, const double b)
{
  return (a < b);
}
```

This code is used in chunk 19.

4.7 compute the Nth element

Compute the Nth element using partial sorting into ascending order using the comparison function § 4.6. If $S_N > N$ juveniles, then returns the value t so that

$$\sum_{1 \le i \le S_N} \mathbb{1}_{\{t_i \le t\}} = N \tag{5}$$

11 ⟨nth 11⟩ ≡

```
static void nthelm(M &self)
{      /* call from R as c(0, Rvec) */
  std::nth_element(self.vEall.begin(), self.vEall.begin() + (self.N - 1), self.vEall.end(),
       comp);
  self.Nth = self.vEall[self.N - 1];
}
```

This code is used in chunk 19.

4.8 sample pool of juveniles

Sample a random number of juveniles for a given subset of the current individuals using § 4.5; returns the total number of juveniles.

12 ⟨ samplepooljuveniles 12 ⟩ ≡

```
static size_t samplepool(const size_t c_ninds, const size_t c_one_two, M &self)
{     /* c_ninds is number of individuals of a given type; c_one_two is the indicator for
         using α1 or α2
         */
  assert(c_ninds > 0);
  assert(c_ninds ≤ self.N);
  int j = static_cast⟨int⟩(c_ninds + 1);
  size_t telja = 0;
  size_t SN = 0;
  while (−−j > 0) {      /* counter for internal control
       */
    ++telja;      /* samplexi in § 4.5
       */
    SN += samplexi(self, c_one_two);
  }
  assert(SN > 0);
  assert(telja ≡ c_ninds);
  return (SN);
}
```

This code is used in chunk 19.

4.9 assign weight to juveniles

Assign weight to all juveniles; the weight of the advantageous type is one, of the wild type is $\exp(-s)$ where $s > 0$ is the strength of selection. We record a random exponential with rate the corresponding weight.

13 ⟨ assignweight 13 ⟩ ≡

```
static void assignweight(M &self, gsl_rng * r) {   /* assign weights to all juveniles;
        SNW is the number of juveniles of the advantageous type, SN the number of
        juveniles of the wild type
        */
    assert(self.SNW > 0);
    assert(self.SN > 0);
    int i = static_cast<int>(self.SNW + 1);   /* clear the containers with the weights
        */
    self.vE.clear();   /* vEall contains all the weights for computing the Nth element
        */
    self.vEall.clear();
    self.vE.shrink_to_fit(); std::vector < double > ().swap(self.vE);
    self.vEall.shrink_to_fit(); std::vector < double > ().swap(self.vEall);
    while (--i > 0) {   /* the optimal genotype correspondingly trait value is
          g_0 = 0, so weight is W = 1
          */
      self.vE.push_back(gsl_ran_exponential(r, 1.));
      self.vEall.push_back(self.vE.back());
    }
    i = static_cast<int>(self.SN + 1);
    while (--i > 0) {   /* the wild type is denoted 1, so weight is W = e^{-s}
          */
      self.vEall.push_back(gsl_ran_exponential(r, 1./exp(-self.c_strength_selection)));
```

```
    }
    }
```

This code is used in chunk 19.

4.10 count survivors of a bottleneck

Count number of individuals of the advantageous type surviving a bottleneck N_b. We suppose the y individuals of the advantageous type are enumerated from 0 to $y-1$, we shuffle the N indexes and count, with $p := \min(y, N_b)$,

$$y' = \sum_{i=1}^{N} \mathbb{1}_{\{\sigma(i)<p\}} \tag{6}$$

where $\sigma(i)$ is the shuffled index.

14 ⟨surviving 14⟩ ≡

```
static void count_number_surviving_bottleneck(M &self) { size_t newy = 0;
    const size_t reference_point = self.y < self.bottlenecksize ? self.y : self.bottlenecksize;
    std::random_shuffle(std::begin(self.index), std::end(self.index)); for (const auto
        &i:self.index)
    {     /* check if an advantageous individual survives the bottleneck Eq (6)
           */
      newy += (i < reference_point ? 1 : 0);
    }     /* set number of good type to count of surviving bottleneck
           */
    self.y = (newy < self.bottlenecksize ? newy : self.N);
    self.Nprime = (newy < self.bottlenecksize ? self.bottlenecksize - newy : 0); }
```

This code is used in chunk 19.

4.11 surviving bottleneck by hypergeometric

The number surviving a bottleneck is a hypergeometric.

15 ⟨ Bottleneckypergeometric 15 ⟩ ≡

```
static void surviving_bottleneckHypergeometric(M &self, gsl_rng *r)
{
  const unsigned newy = gsl_ran_hypergeometric(r, self.y, self.Nprime, self.bottlenecksize);
     /* set number of good type to count of surviving bottleneck
      */
  self.y = (newy < self.bottlenecksize ? newy : self.N);
     /* Nprime is number of wild type
      */
  self.Nprime = (newy < self.bottlenecksize ? self.bottlenecksize - newy : 0);
}
```

This code is used in chunk 19.

4.12 count surviving by weight

Count the number of juveniles of the advantageous type surviving a selection by weight.

16 ⟨byweight 16⟩ ≡

```
static void count_number_surviving_assigning_weight(M &self) { self.y = 0;
      /* self.vE is container with juveniles of the advantageous type
       */
    for (const auto &t:self.vE)
    {
      self.y += (t ≤ self.Nth ? 1 : 0);
    }
    }
```

This code is used in chunk 19.

4.13 one step

take one step through an excursion

17 ⟨ onestep 17 ⟩ ≡

```
static void onestep_after_bottleneck(M &self, gsl_rng * r)
{   /* check if bottleneck occurs, and if it occurs sample surviving § 4.11
    */
  if (gsl_rng_uniform(r) < self.pbottle) {
    surviving_bottleneckHypergeometric(self, r);
  }
  if (self.y > 0) {
    if (self.y < self.N) {   /* need to sample juveniles
        */
      const size_t c_one_two = (gsl_rng_uniform(r) < self.epsilon_N ? 1 : 2);
      assert(self.y > 0);
      assert(self.Nprime > 0);   /* sample juveniles with advantageous type § 4.8
        */
      self.SNW = samplepool(self.y, c_one_two, self);
        /* sample juveniles with wild type
        */
      self.SN = samplepool(self.Nprime, c_one_two, self);
      if (self.SNW + self.SN ≤ self.N) {
          /* total number of juveniles does not exceed N so all survive
          */
        self.y = self.SNW;
        self.Nprime = self.SN;
      }
      else {   /* total number of juveniles exceeds N so need to assign weights § 4.9
          */
```

```
            assignweight(self, r);      /* compute the Nth smallest exponential § 4.7
                 */
            nthelm(self);      /* § 4.12
                 */
            count_number_surviving_assigning_weight(self);
            assert(self.y ≤ self.N);
            self.Nprime = self.N − self.y;
          }
        }
        else {
          assert(self.y ≡ self.N);
        }
      }
      else {
        assert(self.y < 1);
      }
    }
```

This code is used in chunk 19.

4.14 trajectory

18 ⟨traject 18⟩ ≡

```
static void trajectory(M &self, gsl_rng * r) { std::vector < double > excursion_to_fixation
    {}
    ;
    double timi = 0.0;
    int j = 0;
    ; std::vector < double > current_number_individuals
    {}
    ;
    self.y = 1;
    self.Nprime = self.N - 1;
    while ((self.y < self.N) ∧ (self.y > 0)) {
      current_number_individuals.push_back(self.Nprime + self.y);
      excursion_to_fixation.push_back(self.y);
      timi += 1.0;
      onestep_after_bottleneck(self, r);
    }
    printf("%d␣%g\n", self.y < self.N ? 0 : 1, timi); if (self.y > 0) {
        std::cout ≪ self.c_excursion_skra ≪ '\n';
    std::ofstream f(self.c_excursion_skra, std::ofstream::app);
    assert(f.is_open()); for (auto const &y:excursion_to_fixation)
    {
      f ≪ y/current_number_individuals[j] ≪ '␣';
      ++j;
    }
    f ≪ "1\n";
    f.close(); } excursion_to_fixation.clear();
```

$$excursion\_to\_fixation.shrink\_to\_fit(\,);\;\; std::vector\;\;<\;\;\mathbf{double}>(\,).swap(excursion\_to\_fixation);\;\}$$

This code is used in chunk 19.

4.15 the main module

The *main* function

```
19 ⟨ includes 5 ⟩
   ⟨ structM 6 ⟩
   ⟨ gslrng 7 ⟩
   ⟨ cdf 8 ⟩
   ⟨ samplej 9 ⟩
   ⟨ comp 10 ⟩
   ⟨ nth 11 ⟩
   ⟨ samplepooljuveniles 12 ⟩
   ⟨ assignweight 13 ⟩
   ⟨ surviving 14 ⟩
   ⟨ Bottleneckypergeometric 15 ⟩
   ⟨ byweight 16 ⟩
   ⟨ onestep 17 ⟩
   ⟨ traject 18 ⟩
   int main(int argc, char *argv[ ])
   {
     M d
     { }
     ;
     setup_rng(static_cast⟨unsigned long int⟩(atoi(argv[1])));
     invcdf(d);
     for (int i = 0; i < d.number_experiments; ++i) {
       printf("%d␣/␣%d␣:␣", i, d.number_experiments);
       trajectory(d, rngtype);
```

```
  }      /* clear the memory occupied by the containers 4.2
          */
  d.freemem( );      /* free the random number generator in § 4.3
        */
  gsl_rng_free(rngtype);
  return 0;
}
```

5 example excursions

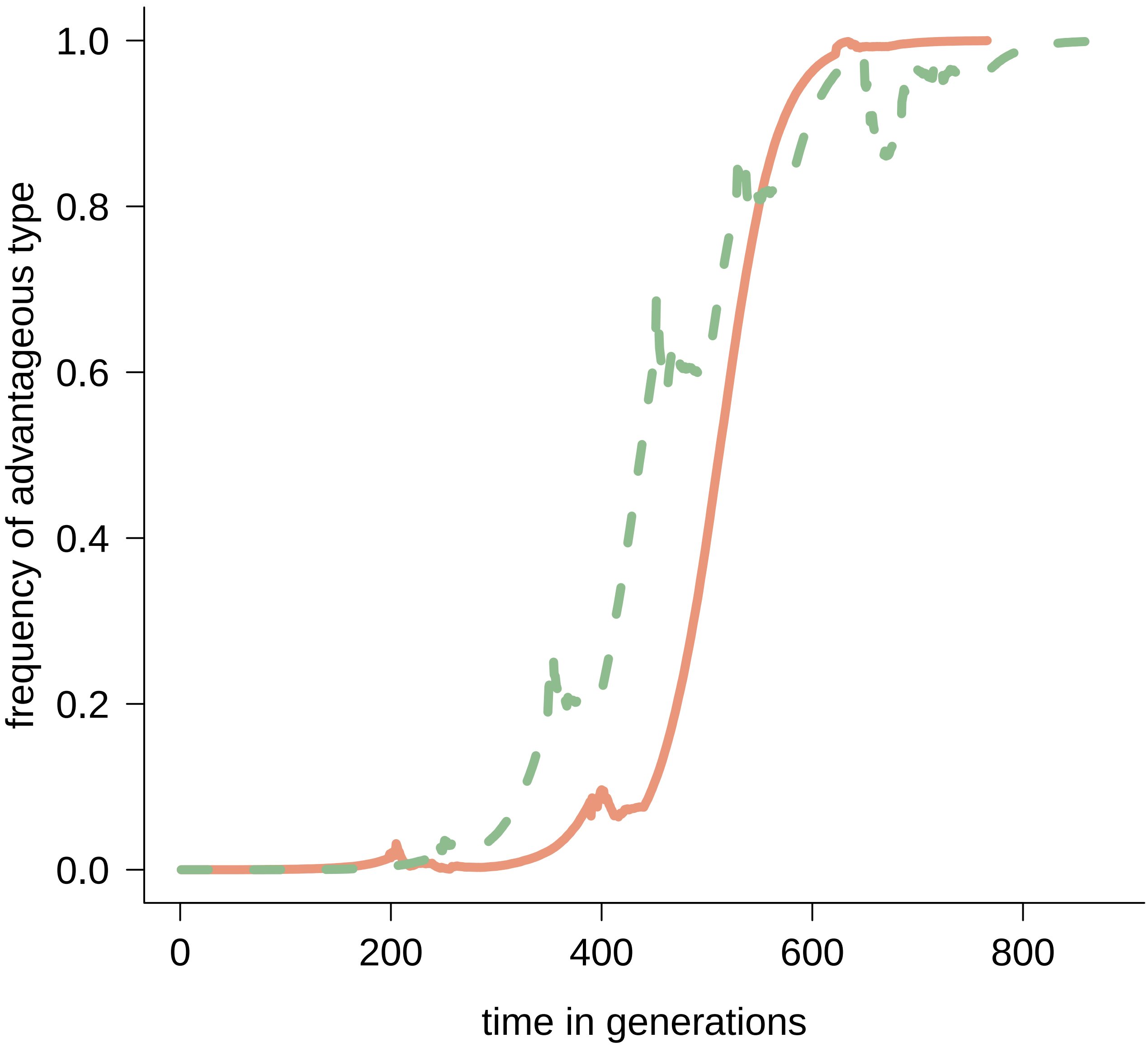

Figure 1: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.1$, size of bottleneck 10^2, probability of a bottleneck in any given generation 0.01

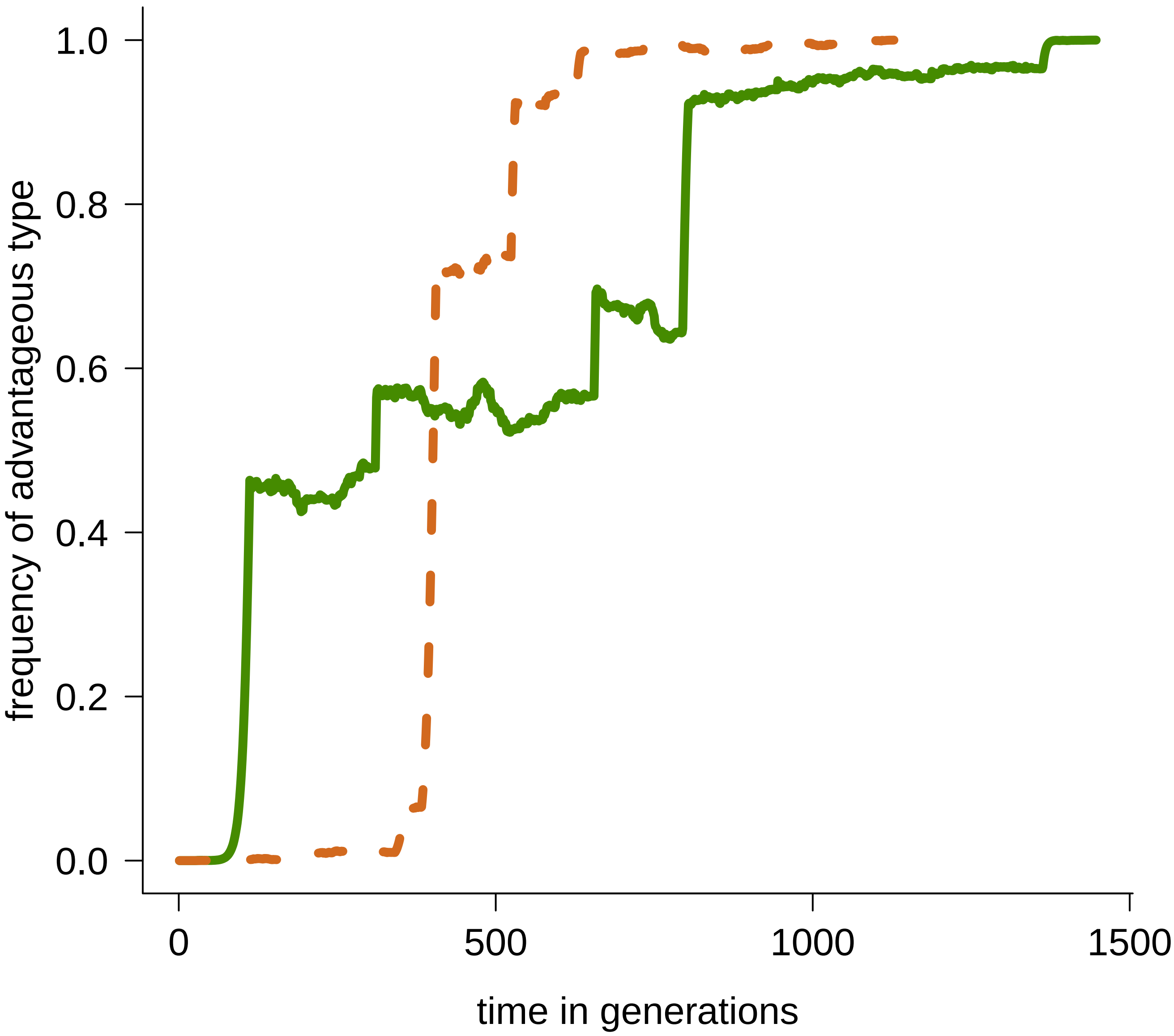

Figure 2: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1

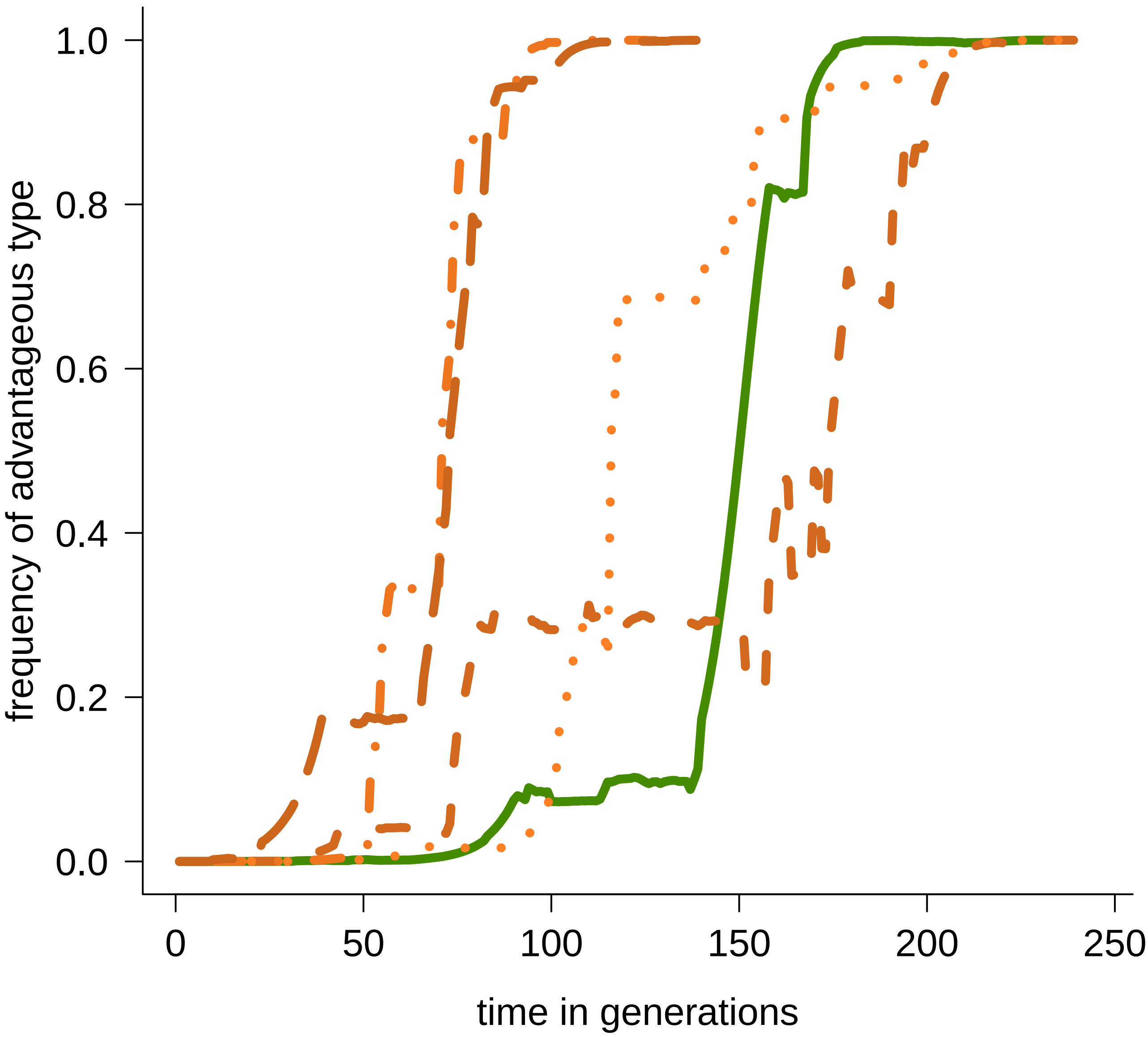

Figure 3: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 0.1$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

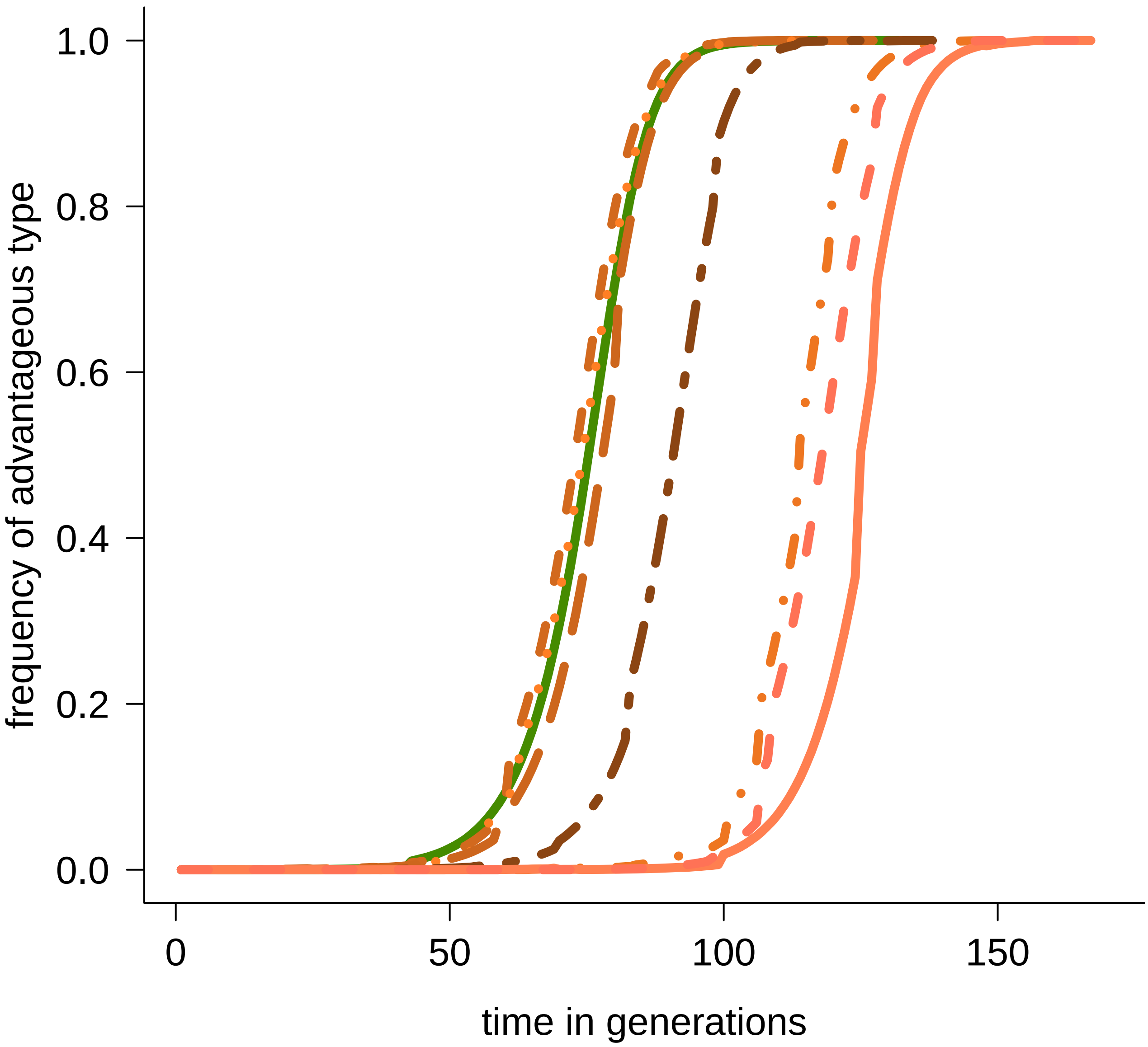

Figure 4: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 0.1$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.01; results from 10^2 experiments

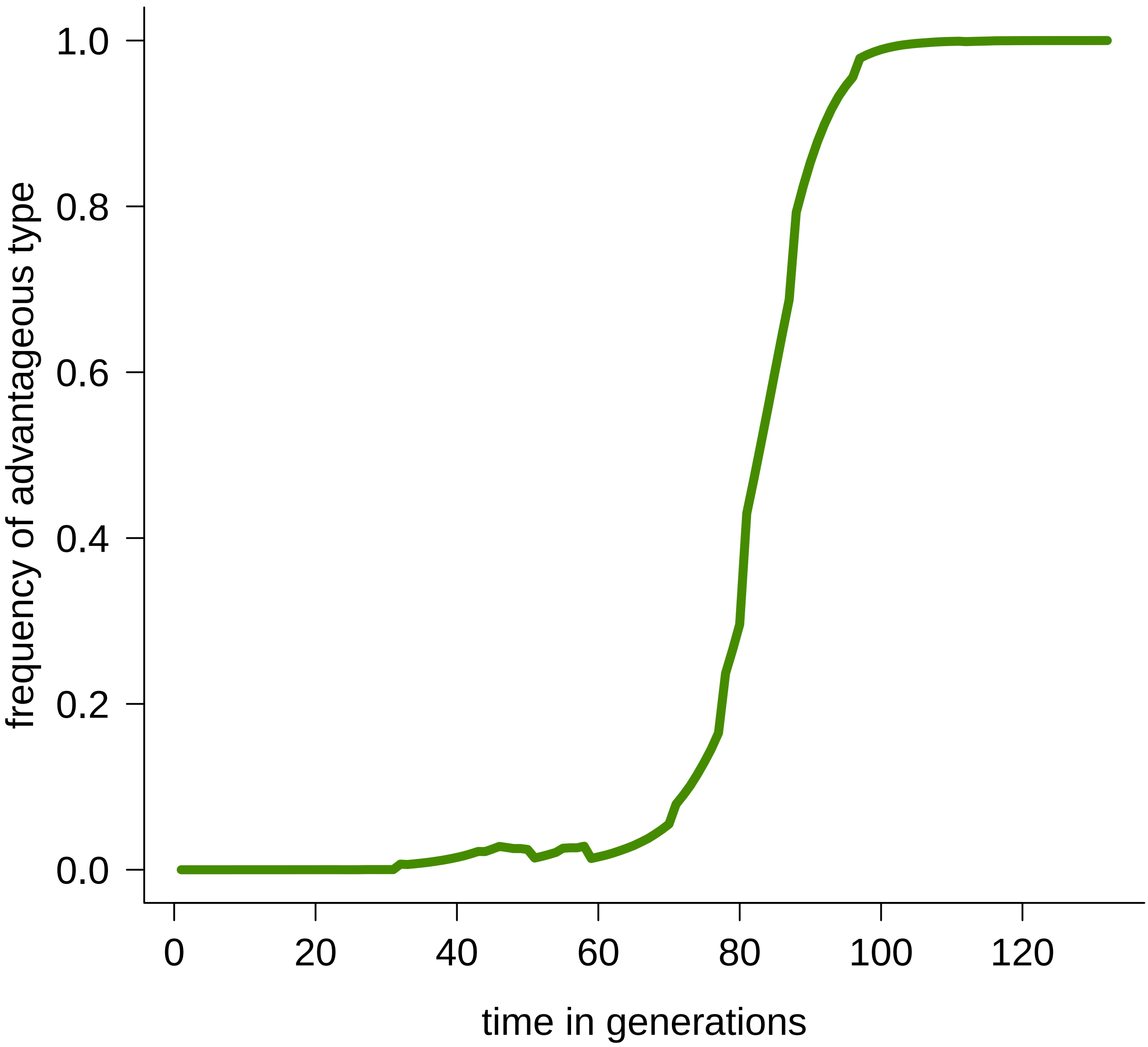

Figure 5: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = .75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 0.1$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1; results from 10^2 experiments

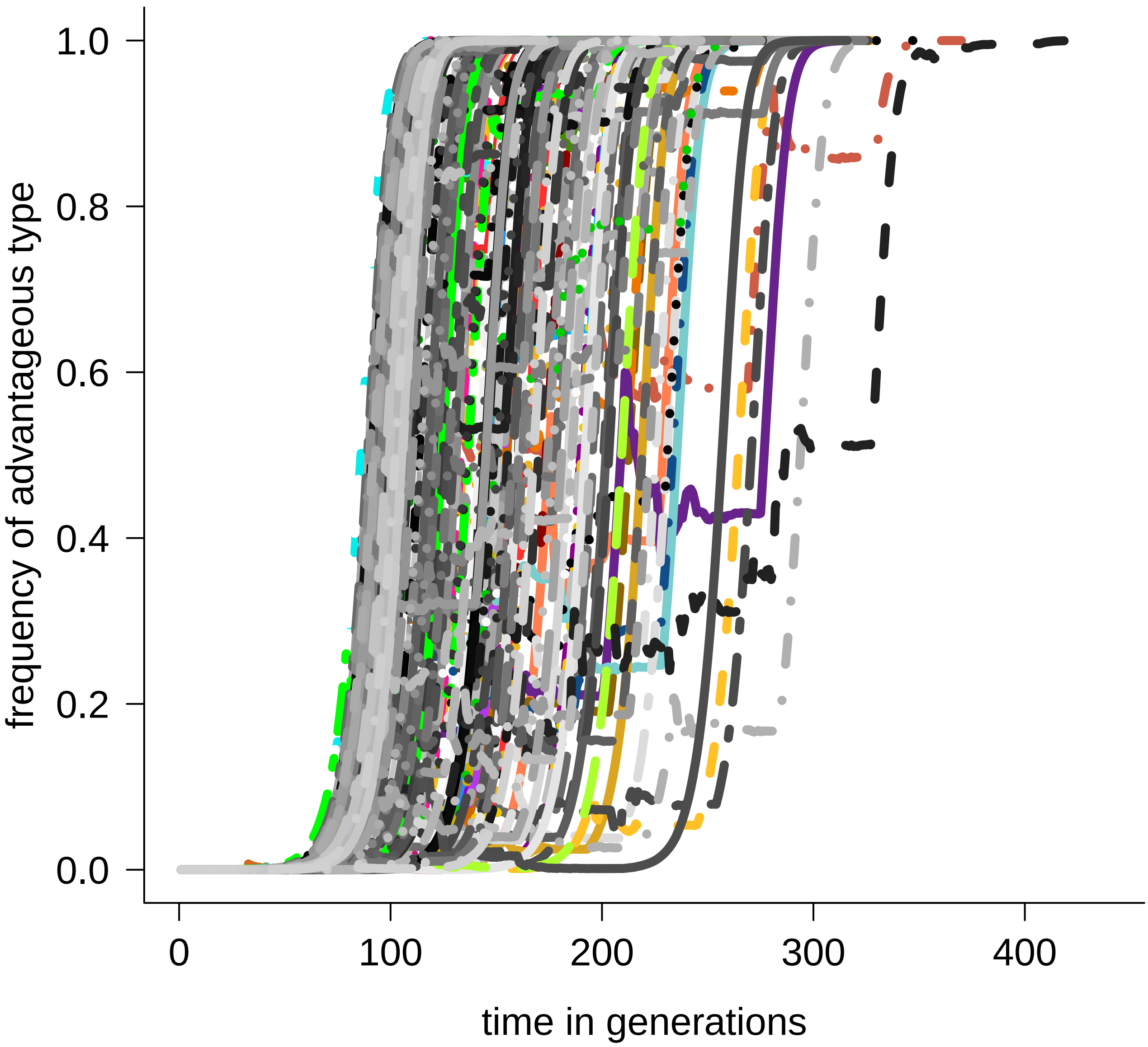

Figure 6: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^2, probability of a bottleneck in any given generation 0.01; results from 10^2 experiments

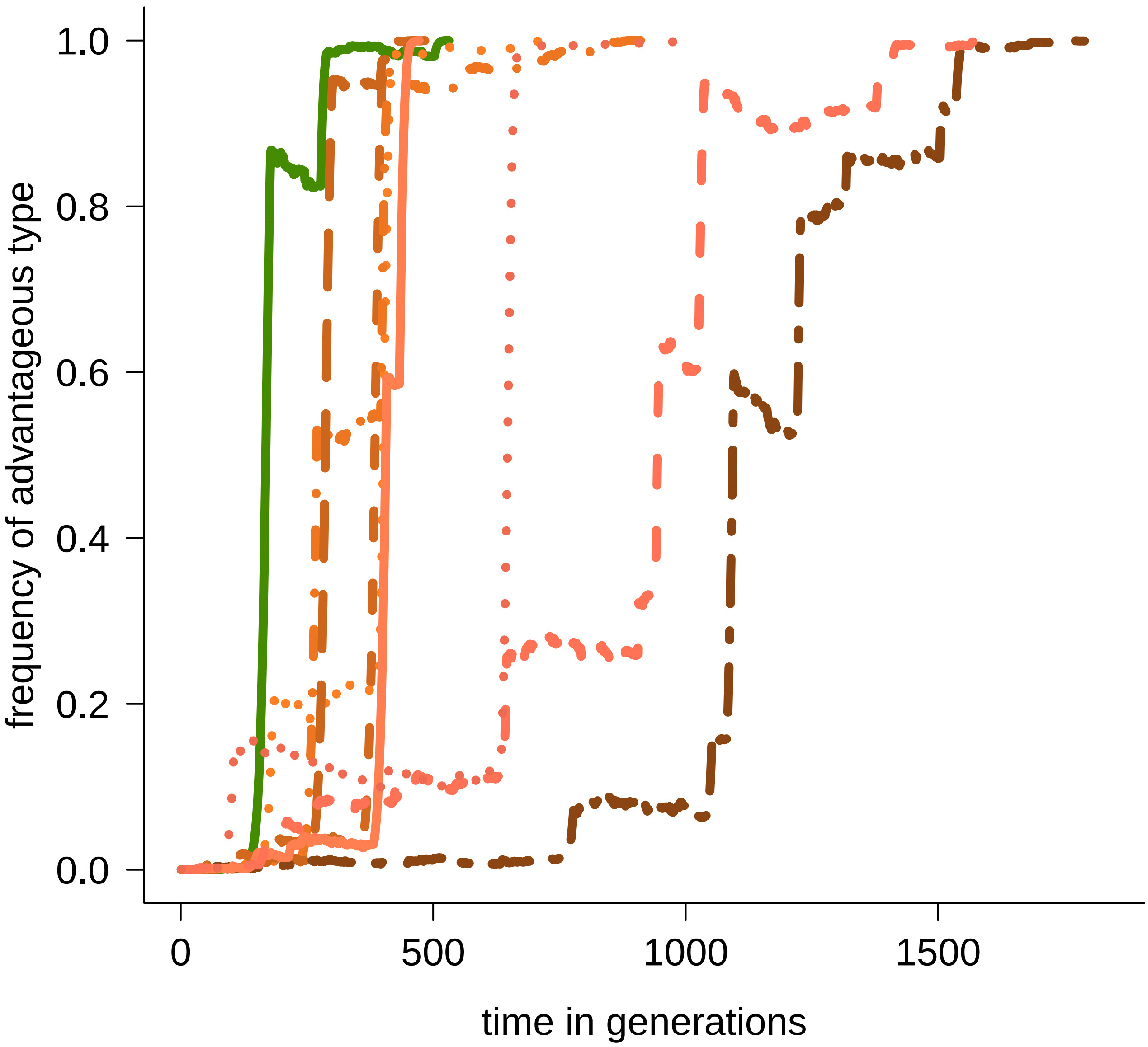

Figure 7: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

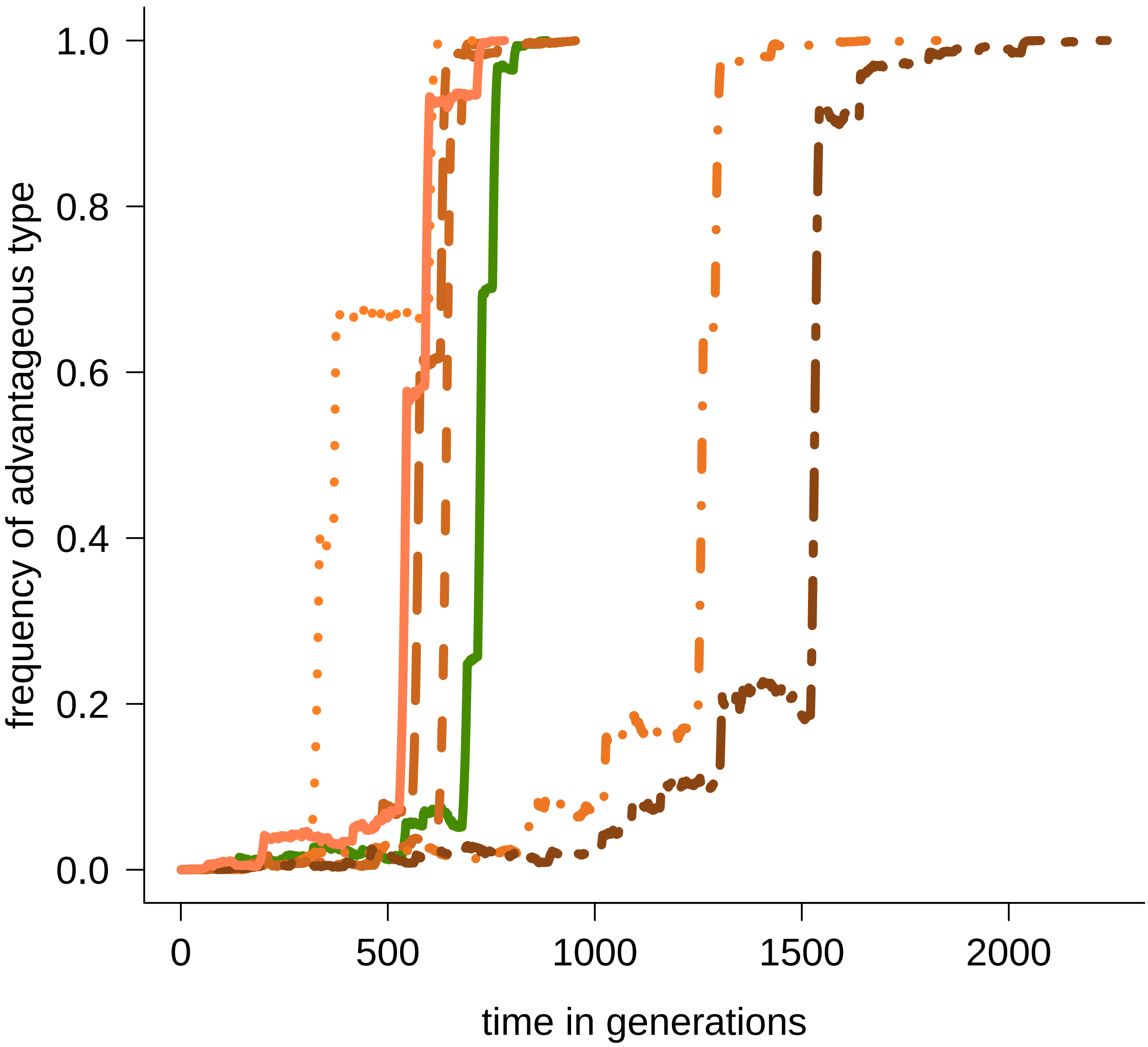

Figure 8: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

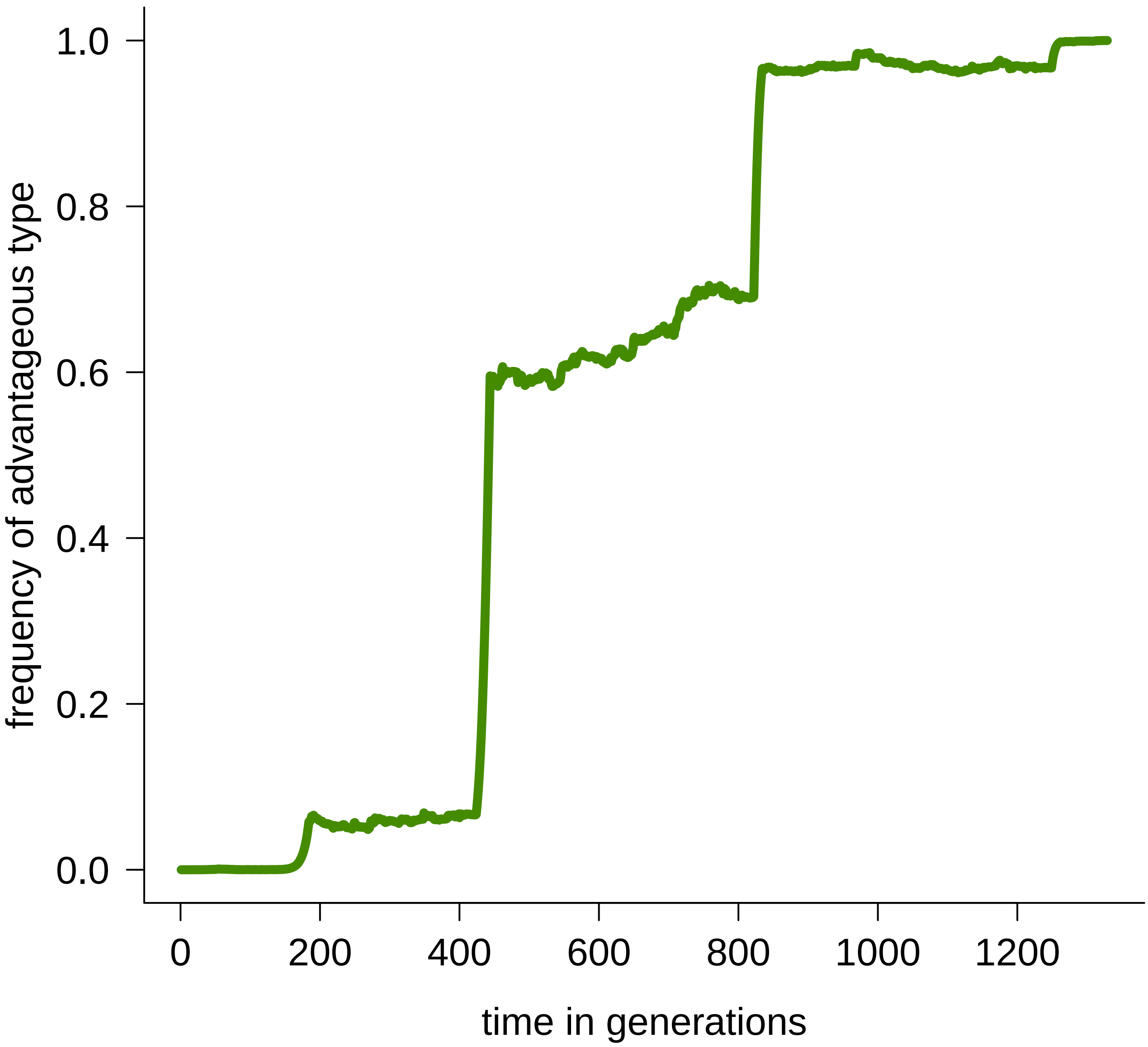

Figure 9: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^4, probability of a bottleneck in any given generation 0.1; results from 10^2 experiments

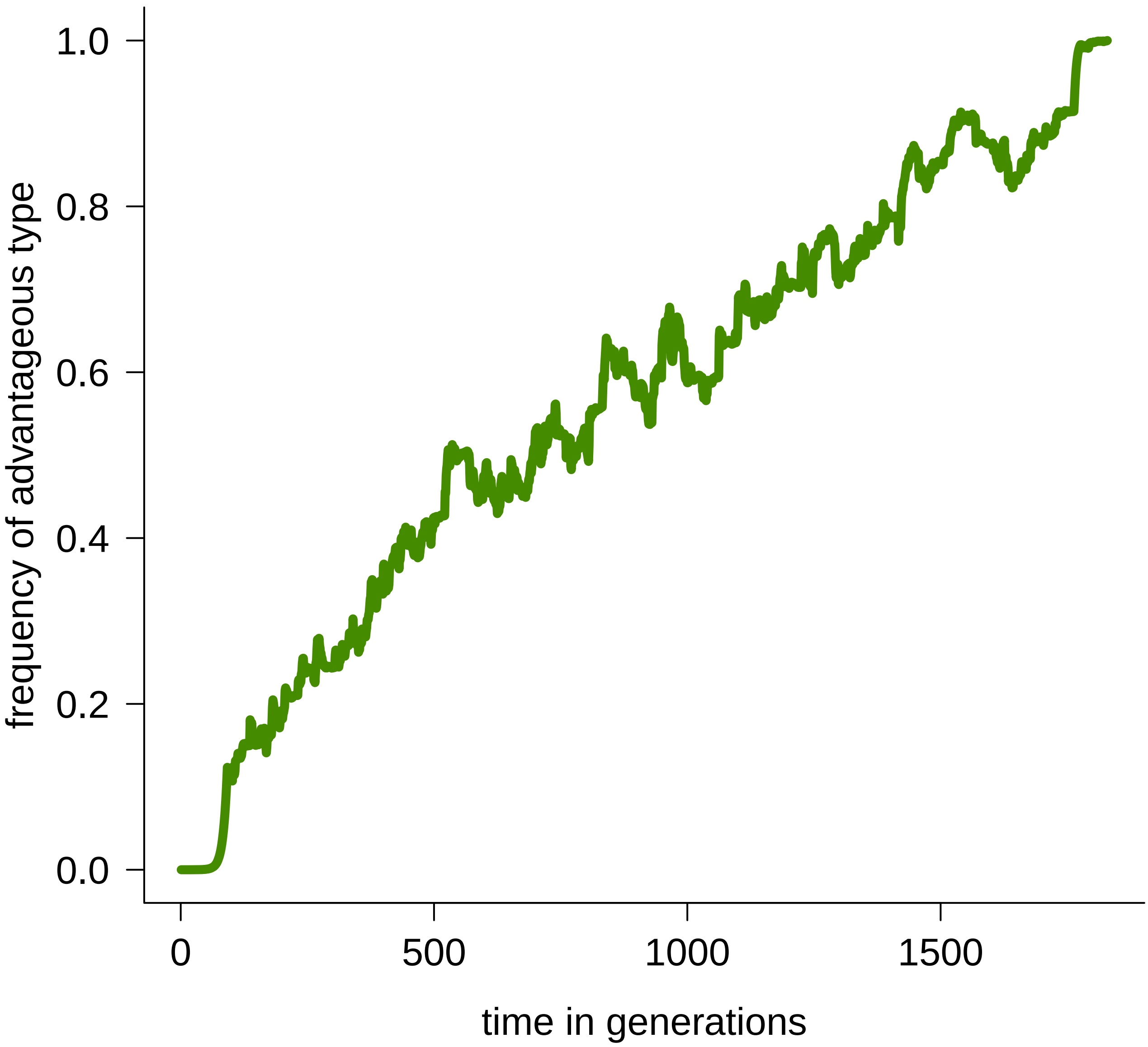

Figure 10: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 1/N$, selection strength $s = 0.5$, size of bottleneck 10^3, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

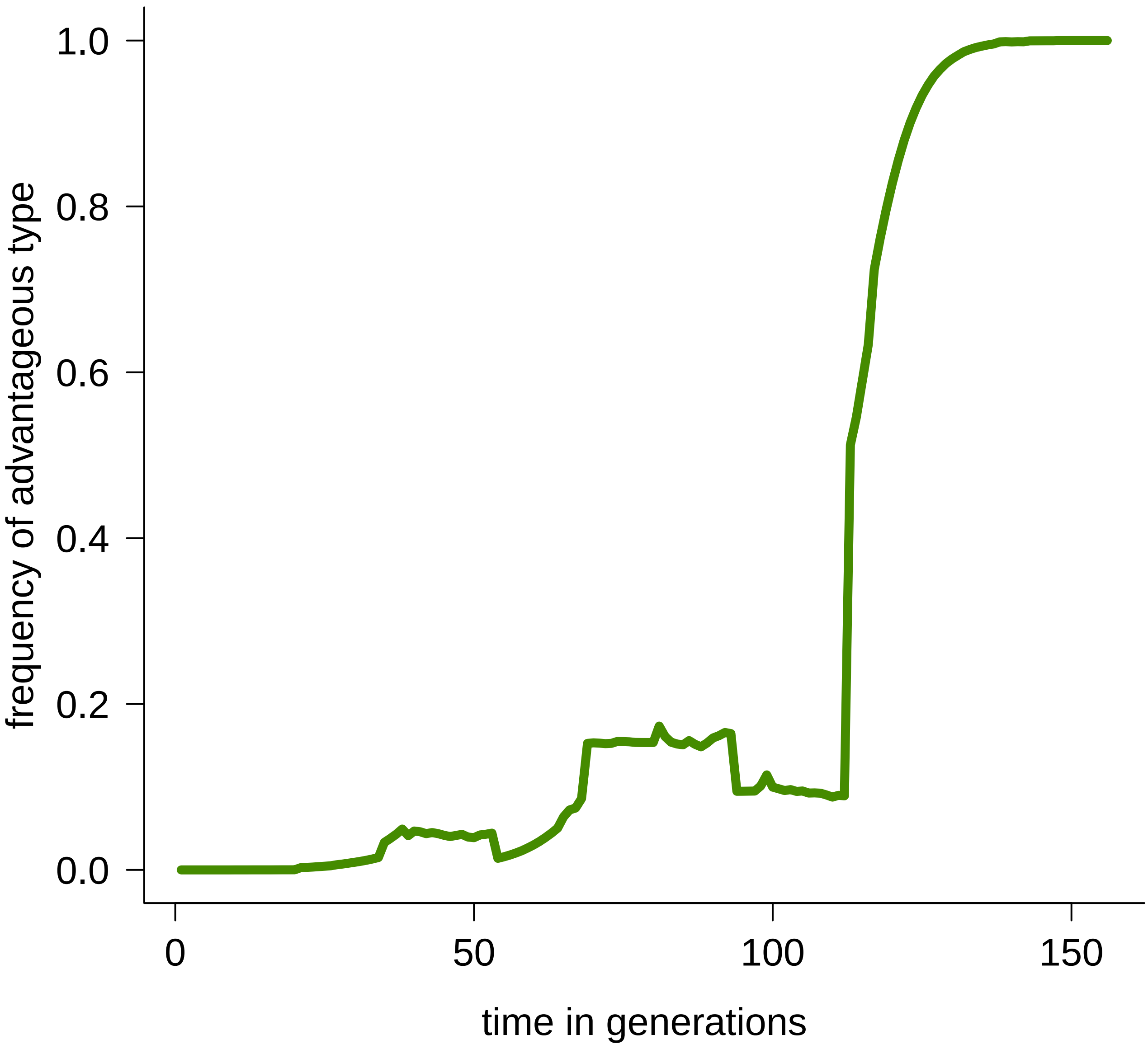

Figure 11: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 0.1$, selection strength $s = 0.5$, size of bottleneck 10^3, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

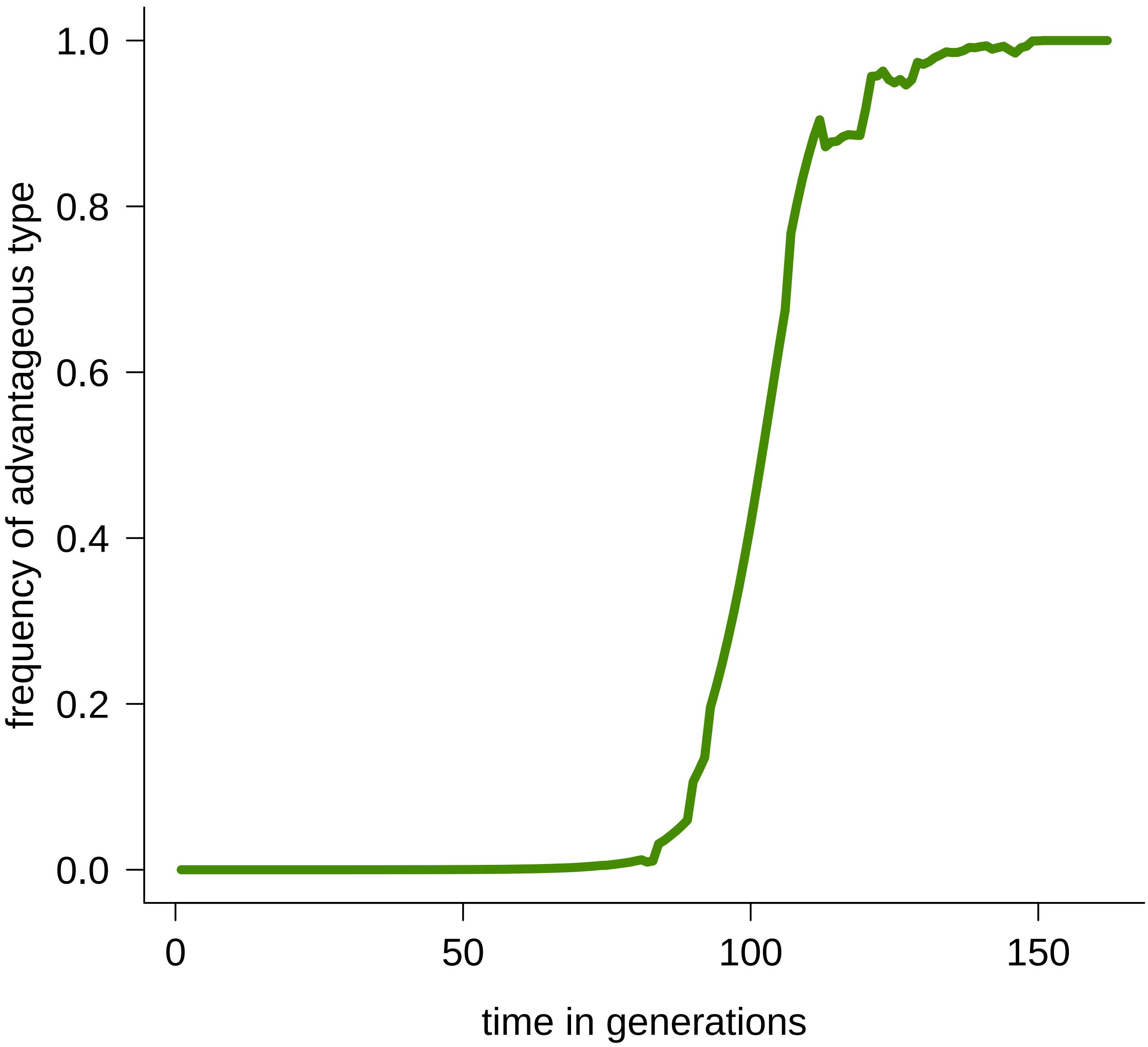

Figure 12: Examples of excursions to fixation for $N = 10^6$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, cutoff $\Psi_N = N$ Eq (1), $\varepsilon_N = 0.1$, selection strength $s = 0.5$, size of bottleneck 10^2, probability of a bottleneck in any given generation 0.1; results from 10^3 experiments

6 conclusion

We are interested in understanding how random sweepstakes and randomly occurring bottlenecks affect viability selection in a haploid population.

In the absence of selection and bottlenecks the transition of Y_t is given by a hypergeometric, for $0 \le x \le N$

$$\mathbb{P}\left(Y_{t+1} = x : Y_t = y\right) = \mathbb{E}\left[\mathbb{P}\left(Y_{t+1} = x : Y_t = y, S_y, S_{N-y}\right)\right] = \mathbb{E}\left[\frac{\binom{S_y}{x}\binom{S_{N-y}}{N-x}}{\binom{S_N}{N}}\right] \tag{7}$$

where S_y is the random number of juveniles produced by y parents of the advantageous type, and S_{N-y} is the random number of juveniles produced by $N-y$ parents of the wild type and $S_N = S_{Y_t} + S_{N-Y_t}$. Given our model of random sweepstakes Eq (2) and Eq (1) we have $S_N \ge N$ almost surely.

In the presence of bottlenecks let N_t denote the population size at time t, i.e. with B the bottleneck size $B \le N_t \le N$ and $1 < B < N$. Then, with $S_{N_t} = S_y + S_{N_t - y}$

$$\mathbb{P}\left(Y_{t+1} = x : Y_t = y\right) = \mathbb{E}\left[\mathbb{1}_{\{S_y = x, S_{N_t} \le N\}}\right] + \mathbb{E}\left[\frac{\binom{S_y}{x}\binom{S_{N_t - y}}{N-x}}{\binom{S_{N_t}}{N}}\mathbb{1}_{\{S_{N_t} > N\}}\right] \tag{8}$$

Including selection corresponds to weighted sampling where the probability of sampling a juvenile with weight one is

$$\frac{S_y}{S_y + e^{-s} S_{N_t - y}} \tag{9}$$

when y parents are of type with weight one.

A mathematical investigation of our model will have to be postponed for future study.

7 references

References

[1] JA Chetwyn-Diggle, Bjarki Eldon, and Alison M Etheridge. Beta-coalescents when sample size is large. in preparation.

[2] Iulia Dahmer and Bjarki Eldon. Coalescent processes from models of random sweepstakes. in preparation.

[3] Brian W Kernighan and Dennis M Ritchie. The C programming language, 1988.

[4] Donald Ervin Knuth and Silvio Levy. *The CWEB system of structured documentation: version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Reading, Massachusetts, 1994.

Index

List of Refinements

⟨ Bottleneckypergeometric 15 ⟩ Used in chunk 19.

⟨ assignweight 13 ⟩ Used in chunk 19.

⟨ byweight 16 ⟩ Used in chunk 19.

⟨ cdf 8 ⟩ Used in chunk 19.

⟨ comp 10 ⟩ Used in chunk 19.

⟨ gslrng 7 ⟩ Used in chunk 19.

⟨ includes 5 ⟩ Used in chunk 19.

⟨ nth 11 ⟩ Used in chunk 19.

⟨ onestep 17 ⟩ Used in chunk 19.

⟨ samplej 9 ⟩ Used in chunk 19.

⟨ samplepooljuveniles 12 ⟩ Used in chunk 19.

⟨ structM 6 ⟩ Used in chunk 19.

⟨ surviving 14 ⟩ Used in chunk 19.

⟨ traject 18 ⟩ Used in chunk 19.