Random sweepstakes and viability selection

Bjarki Eldon[1][2]

Abstract

This C++ code generates excursions of the evolution of an advantageous type at a single locus in a diploid population partitioned into three genotypes, and in which two genetic types are segregating, one with viability weight $W = \exp\left(-s(z - z_0)^2\right)$ and the other with viability weight one. The population evolves according to a model of sweepstakes reproduction and randomly occurring bottlenecks and a simple model of viability selection. We are interested in the probability of fixation of the advantageous type and the time to fixation conditional on fixation, and how random sweepstakes and dominance mechanisms and randomly occurring bottlenecks act on these statistics.

1 Copyright

[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; acknowledge Start-up module grants through SPP 1819 with Jere Koskela and Maite Wilke-Berenguer, and with Iulia Dahmer.

January 21, 2022

2 Compilation, output and execution

This CWEB[5] document (the `.w` file) can be compiled with `cweave` to generate a `.tex` file, and with `ctangle` to generate a `.c`[4] 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. 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 the evolution of an advantageous genetic type at a single locus in a diploid population evolving according to random sweepstakes and randomly occurring bottlenecks.

We randomly pair individuals and each pair independently produces a random number of juveniles with probability mass function

$$\mathbb{P}(X_1 = k) = \frac{\mathbb{1}_{\{2 \le k \le \Psi_N\}}}{2^{-\alpha} - (\Psi_N + 1)^{-\alpha}} \left(\frac{1}{k^\alpha} - \frac{1}{(k+1)^\alpha} \right) \tag{1}$$

Write $X_1, \ldots, X_N \sim L(\alpha, \Psi_N)$ if $X_1, \ldots, X_N$ are independent and identically distributed according to Eq (1). Let $1 < \alpha_1 < 2$ and $\alpha_2 > 2$ be fixed and consider, with Ψ_N fixed, the mixture distribution

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

A scaling of ε_N can be identified so that the model converges to a non-trivial limit (1,2) .

Each pair produces at least two juveniles. Each juvenile is then assigned two alleles, one from each parent, sampled independently and uniformly at random. The genotype, denoted g, of a given juvenile is translated into a phenotype, denoted z, by a given map $f(g)$, and a viability weight (fitness value) w is computed

$$w = \exp(-s(z_0 - f(g))) \tag{3}$$

where z_0 is the optimal phenotype and $s > 0$ is the fixed strength of selection. There are obviously many ways to configure this, and this is one of them and corresponds to the traditional way of computing a fitness from a genotype or genotypes. For each juvenile we sample an independent exponential with rate the given viability weight, and the $2N$ juveniles with the smallest exponentials survive and replace the parents.

We introduce randomly occurring bottlenecks. In any given generation there is a fixed probability of a bottleneck, and the size of the bottleneck, denoted N_b, is fixed. When a bottleneck occurs we randomly sample among the current individuals N_b individuals to survive the bottleneck. The surviving individuals then produce juveniles; if the total number of juveniles is less than the given capacity all the juveniles survive, otherwise we assign weights and exponentials and sample the juveniles as just described. If all the surviving individuals are of the optimal type we consider a fixation has occurred since we exclude mutation, and juveniles inherit the type of the parents as described.

Our motivation for including bottlenecks comes partly from a study on the effect of bottlenecks on the evolution of antibiotic resistance(?). The effect of selection gene genealogies has investigated mathematically mostly in haploid populations, but recently also in diploid populations, in particular where heterozygotes have been assumed to be at a selective disadvantage to both homozygotes(3).

4 code

In § 4.2 we collect the main containers, i.e. the number of diploid individuals of each of the three possible genotypes, the juveniles as a pair of genotype and viability weight, and the cumulative density function for sampling the random number of juveniles according to Eq (1) using the inverse CDF method. We use the GSL random number generator § 4.3, in § 4.4 we sample the type of a parent, in § 4.5 we assign a genotype to a juvenile given the genotype of both parents, in § 4.6 we sample a random number of juveniles using the inverse CDF method, in § 4.7 we compute the viability weight, or the rate for the exponential for sorting the juveniles; in § 4.8 we add juveniles produced by a parent pair with given genotypes to the pool of juveniles; in § 4.9 we generate a pool of juveniles for a randomly arranged pairs of parents, if there is an odd number of individuals we assume one individual will be left out; in § 4.10 we sample number of individuals of each type according to the hypergeometric that will survive a bottleneck; in § 4.11 we take one step through an excursion by checking if a bottleneck occurs, and sampling juveniles; in § 4.12 we generate one excursion; in § 4.13 the probability mass function Eq (1) is defined; in § 4.14 we compute the corresponding CDF; and in § 4.15 we generate a specified number of excursions or experiments.

4.1 includes

the included libraries, and the header file containing the settings as global constants

5 ⟨includes 5⟩ ≡

```
#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <functional>
#include <memory>
#include <utility>
#include <algorithm>
#include <cstddef>
#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>
#include "diploid_excursions_random_bottlenecks.hpp"
```

This code is used in chunk 19.

4.2 struct M

6 ⟨struct 6⟩ ≡

```
struct M { private:      /* diploid single locus: 0/0 = 0; 0/1 = 1; 1/1 = 2; the
        population is partitioned into the three genotypes, the vector population records the
        number of each type indexed in the obvious way
        */
  std::vector < double > population
  { }
  ;      /* for each juvenile we record the weight and the genotype
        */
  std::vector < std::pair < size_t , double >> juveniles
  { }
  ;      /* the vectors containing the cumulative density functions for the random number
        of juveniles Eq (1)
        */
  std::vector < double > cdf_one
  { }
  ; std::vector < double > cdf_two
  { }
  ;
public: void removealljuveniles( )
  {
    juveniles.clear( );
    juveniles.shrink_to_fit( );
    assert(juveniles.size( ) < 1);
  }
  size_t totalnumberjuveniles( )
  {
    return juveniles.size( );
  }
  void add_juvenile(const size_t g, const double weight)
  {
```

```
  juveniles.push_back(std::make_pair(g, weight));
}
double number_type(const size_t c_type)
{
  return population[c_type];
}
double current_number_individuals()
{
  return (population[0] + population[1] + population[2]);
}
void updatepopulationafterbottleneck(const double n0, const double n1, const double
        n2)
{
  population[0] = n0;
  population[1] = n1;
  population[2] = n2;
}
void update_count_type(const size_t c_type, const size_t add_subtract)
{
  population[c_type] += (add_subtract < 1 ? 1. : -1.);
}
size_t type_fewest_copy()
{
  return std::distance(population.begin(), std::min_element(population.begin(),
      population.end()));
}
size_t type_most_copies()
{
  return std::distance(population.begin(), std::max_element(population.begin(),
      population.end()));
}
void update_population_all_juveniles() { std::fill(population.begin(), population.end(), 0.);
```

```
    for (const auto &j:juveniles)
  {
    population[std::get < 0 > (j)] += 1;
  }
} double get_cdfone(const size_t i)
{
  return cdf_one[i];
}
double get_cdftwo(const size_t i)
{
  return cdf_two[i];
}
void add_to_cdfone(const double c_x)
{
  cdf_one.push_back(cdf_one.back( ) + c_x);
}
void add_to_cdftwo(const double c_x)
{
  cdf_two.push_back(cdf_two.back( ) + c_x);
}
void init_containers( )
{     /* starting with one copy of 0/1, the rest is homozygous 0/0
        */
  population.reserve(3);
  population[0] = GLOBAL_CONST_I − 1;
  population[1] = 1;
  population[2] = 0;
  cdf_one.reserve(GLOBAL_CONST_CUTOFF_ONE + 2);
  cdf_two.reserve(GLOBAL_CONST_CUTOFF_TWO + 2);
  assert(population[0] + population[1] + population[2] ≡ GLOBAL_CONST_I);
  cdf_one.push_back(0.);
  cdf_one.push_back(0.);
  cdf_two.push_back(0.);
  cdf_two.push_back(0.);
  assert(cdf_one.size( ) ≡ 2);
  assert(cdf_two.size( ) ≡ 2);
}
void init_for_trajectory( )
{
  population[0] = GLOBAL_CONST_I − 1;
  population[1] = 1;
  population[2] = 0;
  assert(population[0] + population[1] + population[2] ≡ GLOBAL_CONST_I);
}
static bool comp ( std::pair < size_t , double > a, std::pair < size_t , double > b
    )
{
  return (std::get < 1 > (a) < std::get < 1 > (b));
}
double nthelm( )
{
  std::nth_element(juveniles.begin( ), juveniles.begin( ) + (GLOBAL_CONST_II − 1),
      juveniles.end( ), comp);
  return (std::get < 1 > (juveniles[GLOBAL_CONST_II − 1]));
}
void sample_juveniles_according_to_weight(const double c_nth)
{
  population[0] = 0;
  population[1] = 0;
  population[2] = 0;
  size_t j = 0;
  while (j < GLOBAL_CONST_II) {
    assert(j < GLOBAL_CONST_II);
    population[std::get < 0 > (juveniles[j])] += std::get < 1 > (juveniles[j]) ≤
```

```
            c_nth ? 1 : 0;
        ++j;
    }
    assert(population[0] + population[2] + population[1] ≡ GLOBAL_CONST_I);
}
void freememory( ) { population.clear( ); std :: vector < double > ( ).swap(population);
      juveniles.clear( ); std :: vector < std :: pair < size_t , double >> ( ).swap(juveniles);
      cdf_one.clear( ); std :: vector < double > ( ).swap(cdf_one);
      cdf_two.clear( ); std :: vector < double > ( ).swap(cdf_two); } } ;
```
This code is used in chunk 19.

4.3 the random number generator

the GSL 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 sample type of parent

Sample the type of a parent by sampling the marginal of the multivariate hypergeometric. There are only three possible genotypes to sample from, and we need to sample only one copy.

8 ⟨ sample parent type 8 ⟩ ≡

```
static int sample_parent_type_mvhyper(M &self, gsl_rng * r)
{
  int i = 0;      /* the nothers is the total number of individuals of the type we are not
      sampling from
      */
  unsigned int nothers = self.current_number_individuals( ) − static_cast⟨unsigned
      int⟩(self.number_type(i));
  unsigned int x = (self.number_type(i) > 0 ? gsl_ran_hypergeometric(r,
      static_cast⟨unsigned⟩(self.number_type(i)), nothers, 1) : 0);
  if (x < 1) {
    ++i;
    nothers −= static_cast⟨unsigned int⟩(self.number_type(i));
    x = self.number_type(i) > 0 ? gsl_ran_hypergeometric(r,
        static_cast⟨unsigned⟩(self.number_type(i)), nothers, 1) : 0;
  }      /* set the sampled genotype to 2 if 0 or 1 not sampled
        */
  i += (x < 1 ? 1 : 0);      /* update the count of the sampled type in the population
      */
  self.update_count_type(i, 1);      /* return the type of the parent
      */
  return i;
}
```

This code is used in chunk 19.

4.5 assign genotype to juvenile

Assign genotype to a juvenile given the genotype of the two parents; the two alleles are sampled independently and uniformly at random, one allele from each parent.

9 ⟨ typejuvenile 9 ⟩ ≡

```
static int assign_type_juvenile(const size_t gone, const size_t gtwo, gsl_rng * r)
{    /* gone and gtwo are the two parent genotypes
       */
  int g
  {}
  ;
  const double u = gsl_rng_uniform(r);
  switch (gone) {
  case 0:
    {    /* a parent genotype is 0/0
           */
      g = (gtwo < 1 ? 0 : (gtwo < 2 ? (u < 0.5 ? 0 : 1) : 1));
      break;
    }
  case 1:
    {    /* a parent genotype is 0/1
           */
      g = (gtwo < 1 ? (u < .5 ? 0 : 1) : (gtwo < 2 ? (u < 0.25 ? 0 : (u < 0.75 ? 1 : 2)) :
           (u < 0.5 ? 1 : 2)));
      break;
    }
  case 2:
    {    /* a parent genotype is 1/1
           */
      g = (gtwo < 1 ? 1 : (gtwo < 2 ? (u < .5 ? 1 : 2) : 2));
      break;
    }
  default: break;
```

```
    }
    **return** $g$;
  }
```
This code is used in chunk 19.

4.6 sample a random number of juveniles

Sample a random number of juveniles; if F denotes the cumulative distribution function and u a random uniform then we return $\min\{j \in \mathbb{N} : F(j) > u\}$

10 ⟨ sample random number juveniles 10 ⟩ ≡

```
static size_t sample_random_number_juveniles(M &self, const size_t c_twoone, gsl_rng *r)
{
  const double u = gsl_rng_uniform(r);
  size_t j = 2;
  if (c_twoone < 2) {
    while (u > self.get_cdfone(j)) {
      ++j;
    }
  }
  else {
    while (u > self.get_cdftwo(j)) {
      ++j;
    }
  }
  assert(j > 1);
  return j;
}
```

This code is used in chunk 19.

4.7 compute viability weight of a juvenile

Compute a viability weight of a juvenile; the canonical form is $\exp(-sf(g))$ where f is the trait function and g the genotype.

```
11 ⟨weigth 11⟩ ≡
   static double computeweight(const double g, gsl_rng ∗ r)
   {      /∗ z(0/1) < z(0/0) < z(1/1), i.e. the heterozygote is worst off
            ∗/      /∗ const double gg = g < 1 ? 1 : (g < 2 ? 2 : 0) ; ∗/
          /∗ linear : z(0/0) < z(0/1) < z(1/1); set gg as 2 − g
            ∗/      /∗ const double gg = static_cast⟨double⟩(2 − g); ∗/
          /∗ r : z(0/0) < z(0/1) = z(1/1) ; set delta as g > 0 ? 0 : 2
            ∗/
     const double gg = (g < 1 ? 2 : (g < 2 ? 1 : 0));
          /∗ return an exponential with rate e^{−sf(g)}
            ∗/
     return (gsl_ran_exponential(r, 1./exp((−GLOBAL_CONST_SELECTION) ∗ pow(gg, 2.))));
   }
```

This code is used in chunk 19.

4.8 adding juveniles

Add juveniles for a parent pair with given genotypes

12 ⟨ add juveniles for a given parent pair 12 ⟩ ≡

```
static void add_juveniles_for_given_parent_pair(M &self, const int gone, const int
        gtwo, const size_t conetwo, gsl_rng * r)
{     /* sample the random number of juveniles § 4.6
       */
  const size_t numberj = sample_random_number_juveniles(self, conetwo, r);
  assert(numberj > 1);
  int g
  {}
  ;
  for (size_t j = 0; j < numberj; ++j) {     /* assign type to each juvenile using § 4.5
        */
    g = assign_type_juvenile(gone, gtwo, r);
       /* add a juvenile to the pool with weight computed in §4.7
        */
    self.add_juvenile(g, computeweight(static_cast⟨double⟩(g), r));
  }
}
```

This code is used in chunk 19.

4.9 generate pool of juveniles

Generate a pool of juveniles

13 ⟨generate pool juveniles 13⟩ ≡

```
static void generate_pool_juveniles(M &self, gsl_rng * r)
{
  int gone, gtwo
  { }
  ;     /* clear the container
      */
  self.removealljuveniles( );     /* sample distribution of number of juveniles
      */
  const size_t conetwo = (gsl_rng_uniform(r) < GLOBAL_CONST_EPSILON ? 1 : 2);
     /* i runs over number of pairs that can be formed from the current number of
      individuals
      */
  assert(self.current_number_individuals( ) < GLOBAL_CONST_I + 1);
  const double currenti = self.current_number_individuals( );
  for (double i = 0; i < floor(currenti/2.); ++i) {
       /* sample the type of the two parents § 4.4
        */
    gone = sample_parent_type_mvhyper(self, r);
    gtwo = sample_parent_type_mvhyper(self, r);
    if ((gone > −1) ∧ (gtwo > −1)) {     /* should always be true
         */     /* add juveniles for a parent pair with given genotypes § 4.8
         */
      add_juveniles_for_given_parent_pair(self, gone, gtwo, conetwo, r);
    }
  }
  assert(self.totalnumberjuveniles( ) ≥ static_cast⟨size_t⟩(currenti));
}
```

This code is used in chunk 19.

4.10 sample individuals surviving a bottleneck

Sample individuals surviving a bottleneck; the bottleneck can occur at any time, even when the population is recovering from a previous one and has not reached full capacity.

14 ⟨bottleneck 14⟩ ≡

```
static void bottleneck(M &self, gsl_rng * r)
{
  unsigned int n0, n1, n2
  {}
  ;     /* sample number of homozygous 0/0 surviving a bottleneck
      */
  n0 = gsl_ran_hypergeometric(r,
      static_cast⟨unsigned int⟩(self.number_type(0)), static_cast⟨unsigned
      int⟩(self.current_number_individuals() − self.number_type(0)),
      GLOBAL_CONST_BOTTLENECK);
     /* sample the number of heterozygotes 0/1 individuals surviving a bottleneck
      */
  n1 = GLOBAL_CONST_BOTTLENECK − n0 > 0 ? gsl_ran_hypergeometric(r,
      static_cast⟨unsigned int⟩(self.number_type(1)), static_cast⟨unsigned
      int⟩(self.current_number_individuals() − self.number_type(0) − self.number_type(1)),
      GLOBAL_CONST_BOTTLENECK − n0) : 0;
     /* set the number of homozygous 1/1 individuals surviving a bottleneck
      */
  n2 = GLOBAL_CONST_BOTTLENECK − n0 − n1;
  assert(n0 + n1 + n2 ≡ GLOBAL_CONST_BOTTLENECK);     /* update population with
      surviving copies; we are tracing the number of homozygous 1/1 individuals
      */
  self.updatepopulationafterbottleneck(static_cast⟨double⟩(n0),
      static_cast⟨double⟩(n1), (n2 < GLOBAL_CONST_BOTTLENECK ?
      static_cast⟨double⟩(n2) : GLOBAL_CONST_I));
}
```

This code is used in chunk 19.

4.11 take one step

Step through one generation in the trajectory

15 ⟨ onestep 15 ⟩ ≡

```
static void onestep(M &self, gsl_rng * r)
{
  double nth
  {}
  ;     /* check if bottleneck
      */
  if (gsl_rng_uniform(r) < GLOBAL_CONST_PROBABILITY_BOTTLENECK) {
        /* bottleneck occurs; sample surviving types and update population § 4.10
         */
    bottleneck(self, r);
  }
  if (self.number_type(1) + self.number_type(2) > 0) {      /* type 1 not lost
         */
    if (self.number_type(2) < GLOBAL_CONST_I) {
          /* not all individuals of type 2 or homozygous 1/1, so sample juveniles § 4.9
           */
      generate_pool_juveniles(self, r);
      if (self.totalnumberjuveniles() ≤ GLOBAL_CONST_II) {
            /* total number of juveniles not over capacity so all survive
             */
        self.update_population_all_juveniles();
      }
      else {      /* need to partial sort juveniles and sample according to weight
             */
        nth = self.nthelm();
        self.sample_juveniles_according_to_weight(nth);
      }
    }     /* mutation has fixed */
  }     /* mutation has been lost */
  }
```

This code is used in chunk 19.

4.12 generate one trajectory

Generate one experiment, recording the count of the optimal type.

16 ⟨ trajectory 16 ⟩ ≡

```
static void trajectory(M &self, gsl_rng * r) {     /* clear containers for a new trajectory
        */
    self.init_for_trajectory( ); std::vector < double > excursion_to_fixation
    { }
    ;     /* clear the containers for the trajectory
        */
    excursion_to_fixation.clear( );
    excursion_to_fixation.shrink_to_fit( );
    assert(excursion_to_fixation.size( ) < 1);
    int timi = 0;
    while ((self.number_type(2) + self.number_type(1) > 0) ∧ (self.number_type(2) <
            GLOBAL_CONST_I)) {
      assert(self.current_number_individuals( ) > 0);
      excursion_to_fixation.push_back(self.number_type(2)/self.current_number_individuals( ));
      ++timi;
      onestep(self, r);
    }
    size_t i = 0; if (self.number_type(2) > 0) {
       /* fixation occurs so print excursion to file
        */
    FILE *fptr = fopen("excurs_prufa.txt", "a");
       /* print the excursion to fixation to the file
        */
    for (const auto &y:excursion_to_fixation)
    {
      fprintf(fptr, "%g␣", y);
      ++i;
    }
    fprintf(fptr, "\n");
    fclose(fptr); }
        /* print out indicator of loss or fixation and the time of the trajectory
         */
    printf("%d␣%d\n", (self.number_type(2) < 1 ? 0 : 1), timi); }
```

This code is used in chunk 19.

4.13 the mass function

Define the probability mass function for the random number of juveniles Eq (1)

17 ⟨ mass function 17 ⟩ ≡

```
static double px(const double k, const double calpha, const double ccutoff)
{
  return ((pow(1./k, calpha) − pow(1./(k + 1.), calpha))/(pow(.5,
      calpha) − pow(1./(ccutoff + 1.), calpha)));
}
```

This code is used in chunk 19.

4.14 initialise CDF

initialise the cumulative density function for sampling random number of juvenile

18 ⟨ initialise cdf 18 ⟩ ≡

```
static void initialise_cdf(M &self)
{
  for (double i = 2; i ≤ GLOBAL_CONST_PSI_ONE; ++i) {
    self.add_to_cdfone(px(i, GLOBAL_CONST_ALPHA_ONE, GLOBAL_CONST_PSI_ONE));
  }
  for (double j = 2; j ≤ GLOBAL_CONST_PSI_TWO; ++j) {
    self.add_to_cdftwo(px(j, GLOBAL_CONST_ALPHA_TWO, GLOBAL_CONST_PSI_TWO));
  }
}
```

This code is used in chunk 19.

4.15 the main module

19
```
⟨ includes 5 ⟩
    ⟨ struct 6 ⟩
    ⟨ gslrng 7 ⟩
    ⟨ sample parent type 8 ⟩
    ⟨ typejuvenile 9 ⟩
    ⟨ sample random number juveniles 10 ⟩
    ⟨ weigth 11 ⟩
    ⟨ add juveniles for a given parent pair 12 ⟩
    ⟨ generate pool juveniles 13 ⟩
    ⟨ bottleneck 14 ⟩
    ⟨ onestep 15 ⟩
    ⟨ trajectory 16 ⟩
    ⟨ mass function 17 ⟩
    ⟨ initialise cdf 18 ⟩
    int main(int argc, char *argv[ ])
    {
      M d
      { }
      ;
      d.init_containers( );
      initialise_cdf (d);
      setup_rng(static_cast⟨unsigned long⟩(atoi(argv[1])));
      int z = GLOBAL_CONST_NUMBER_EXPERIMENTS + 1;
      while (−−z > 0) {
        trajectory(d, rngtype);
      }
      d.freememory( );
      gsl_rng_free(rngtype);
      return GSL_SUCCESS;
    }
```

5 examples of results

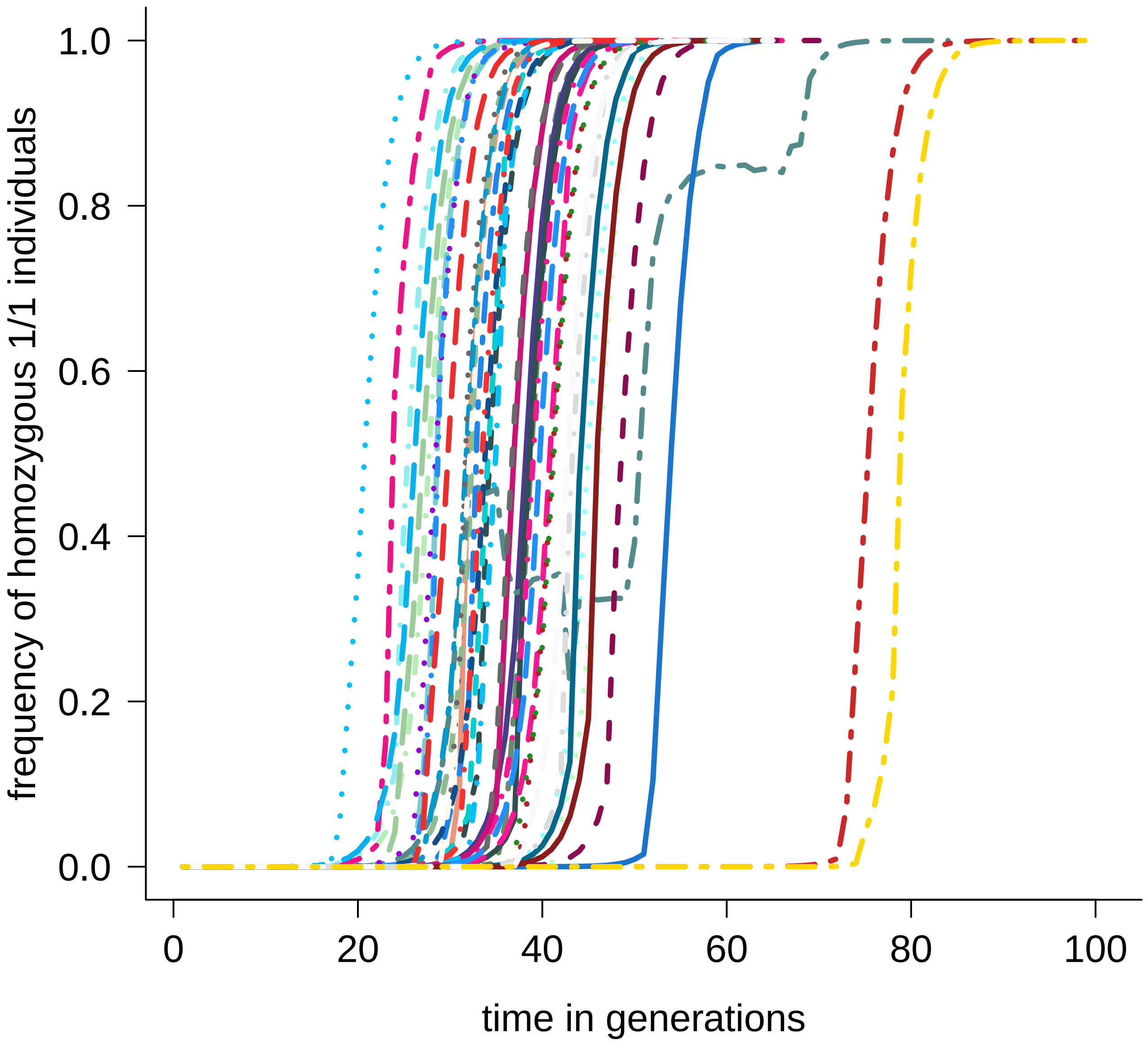

Figure 1: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 0.1$, selection strength $s = 1$, bottleneck 10^2 and probability of a bottleneck in a given generation 0.01; results from 10^2 experiments with 42 fixations and average time 57.9 and stdev 12.8

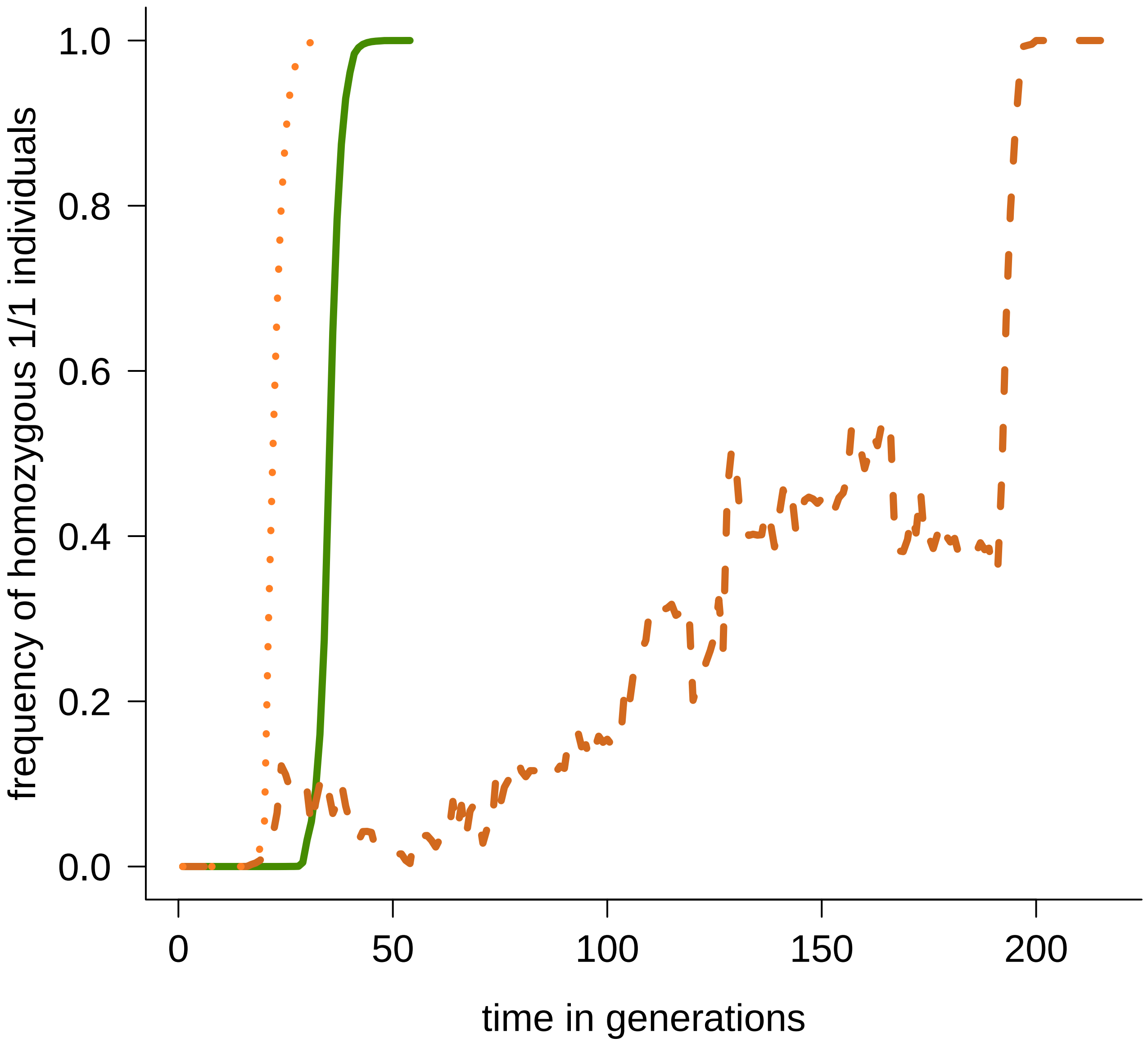

Figure 2: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 0.1$, selection strength $s = 1$, bottleneck 10^2 and probability of a bottleneck in a given generation 0.1; results from 10^2 experiments with 3 fixations and average time 100.7 and stdev 99.6

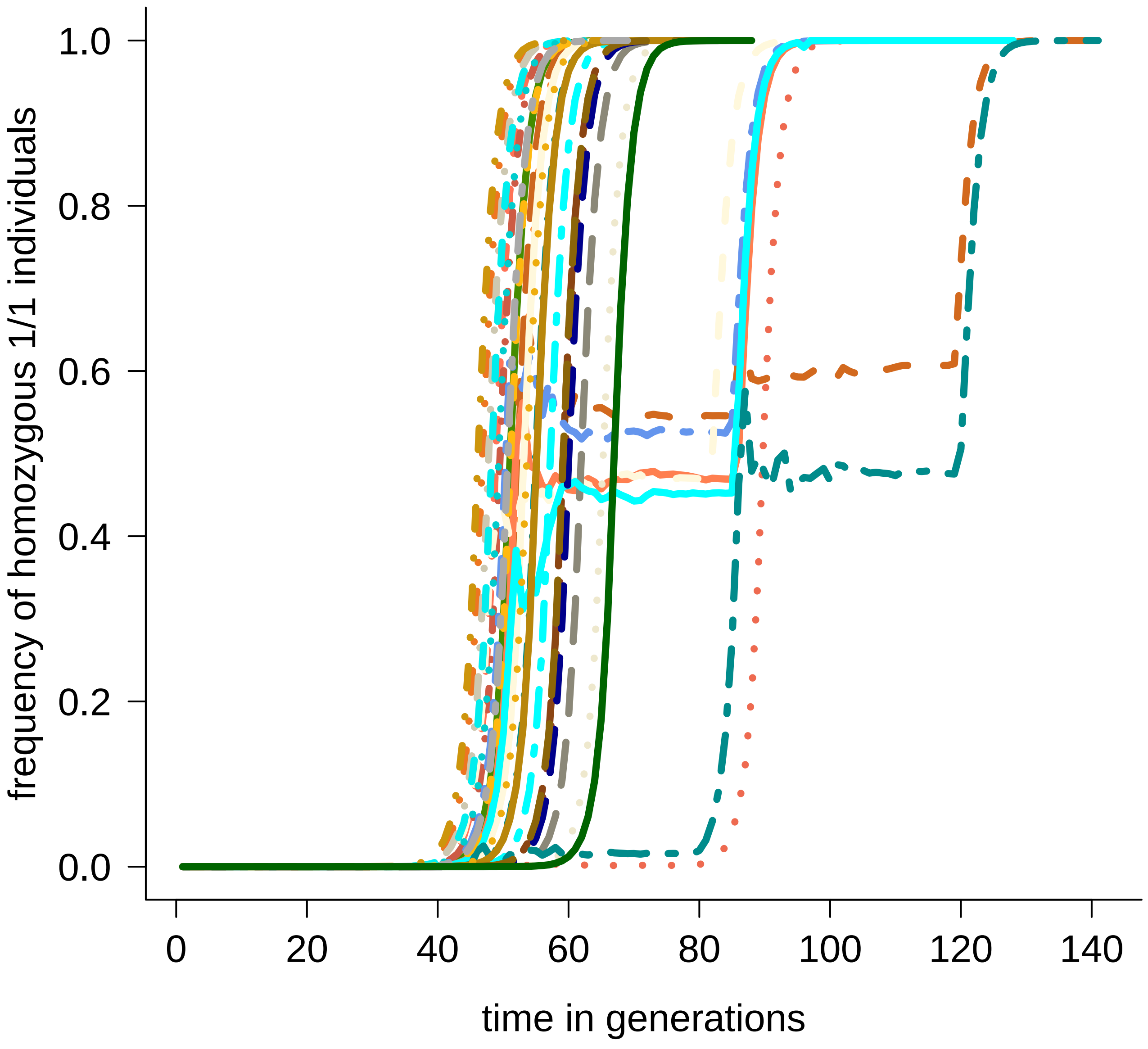

Figure 3: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 1$, bottleneck 10^2 and probability of a bottleneck in a given generation 0.01; results from 10^2 experiments with 31 fixations and average time 85.4 and stdev 21.0

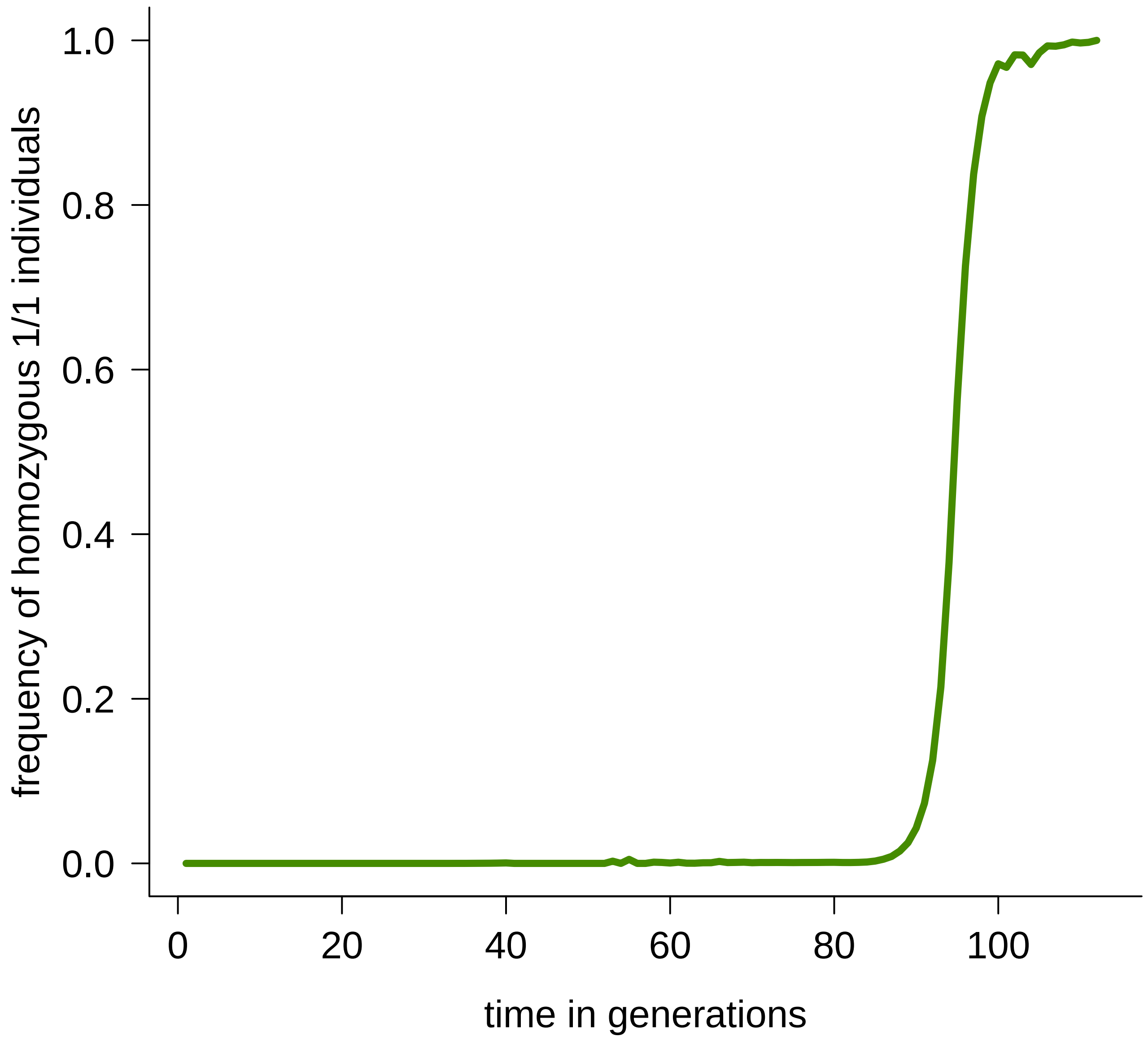

Figure 4: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 1$, bottleneck 10^2 and probability of a bottleneck in a given generation 0.1; results from 10^2 experiments with one fixation and average time 112 and stdev −−

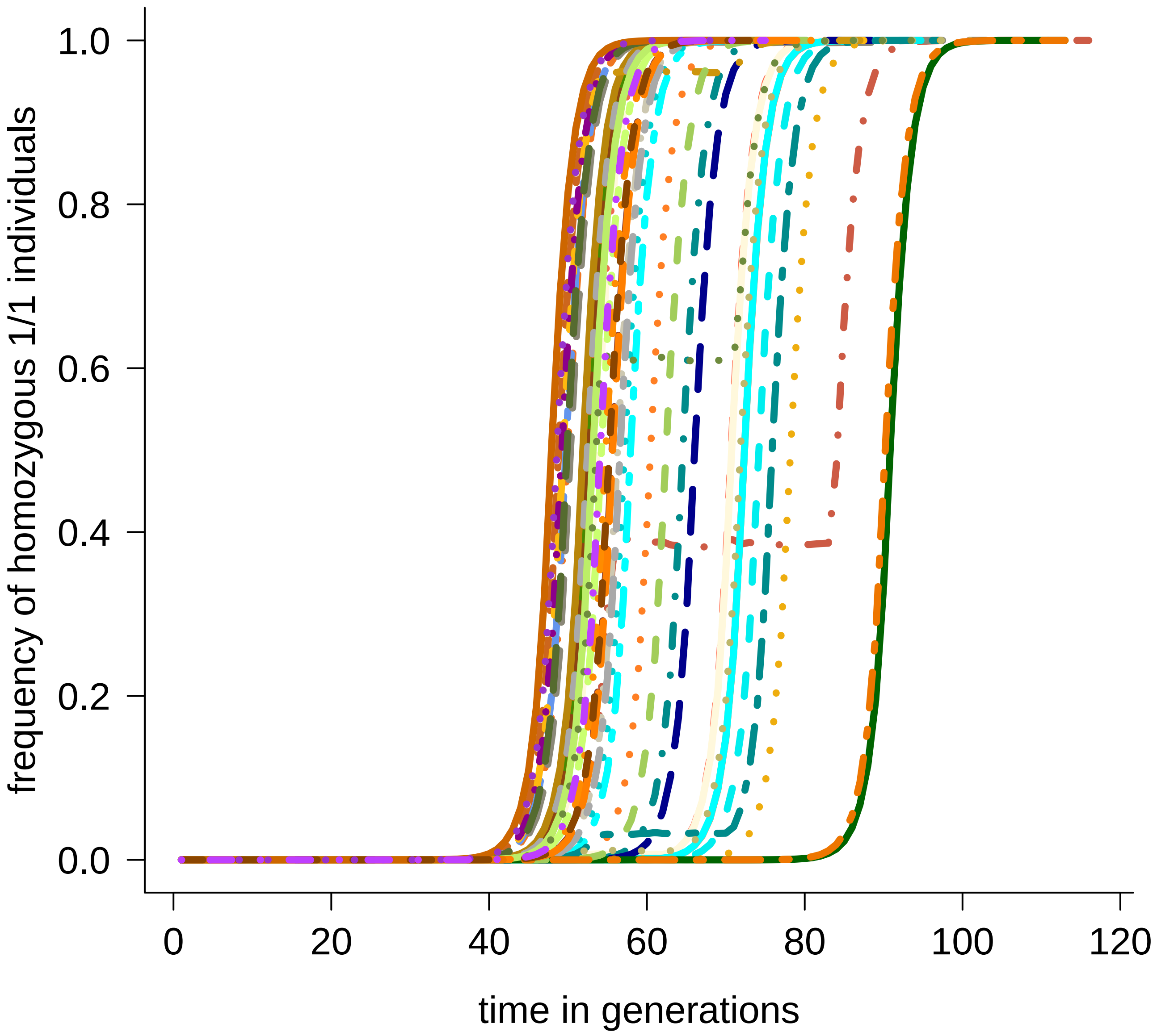

Figure 5: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.01; the dominance mechanism is linear with $z = g$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; results from 10^2 experiments with 46 fixations and average time 83.8 and stdev 12.5

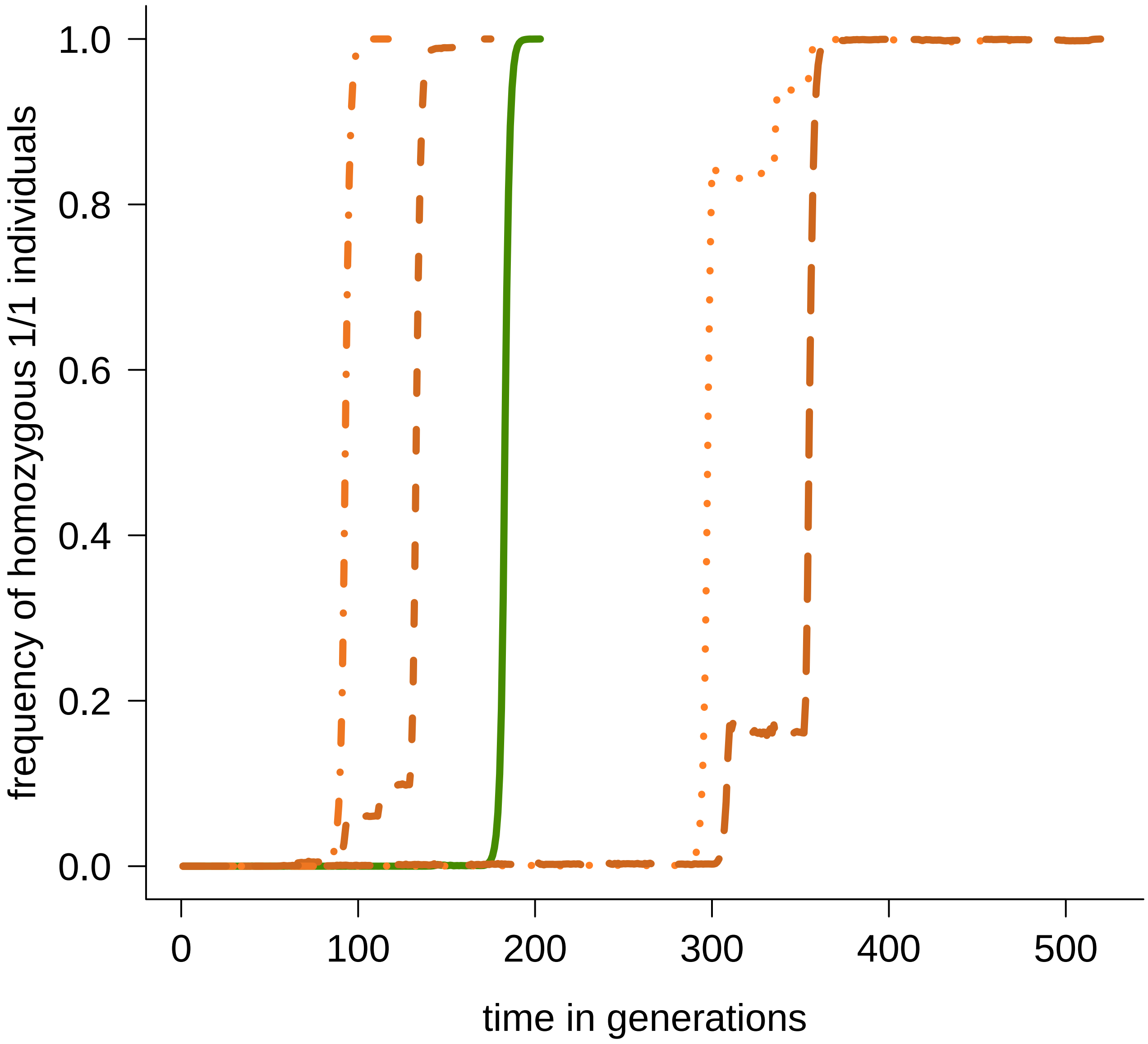

Figure 6: examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.1; the dominance mechanism is linear with $z = g$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; results from 10^2 experiments with five fixations and average time 299.4 and stdev 187.2

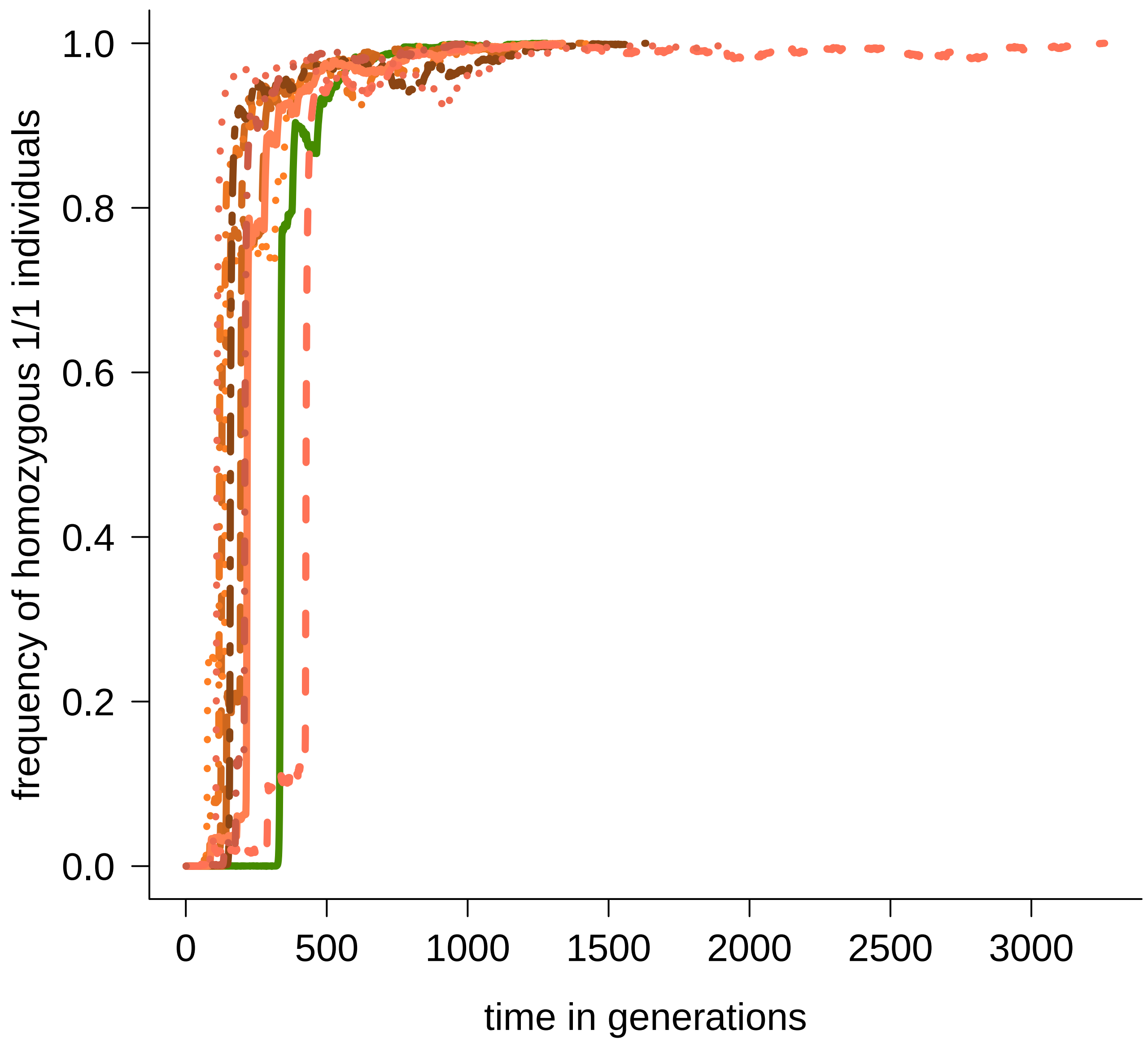

Figure 7: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.1; the dominance mechanism is r-shape with $z = 2\mathbb{1}_{\{g>0\}}$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; ; results from 10^2 experiments with ten fixations and average time to fixation 1600.2 and stdev 631.3

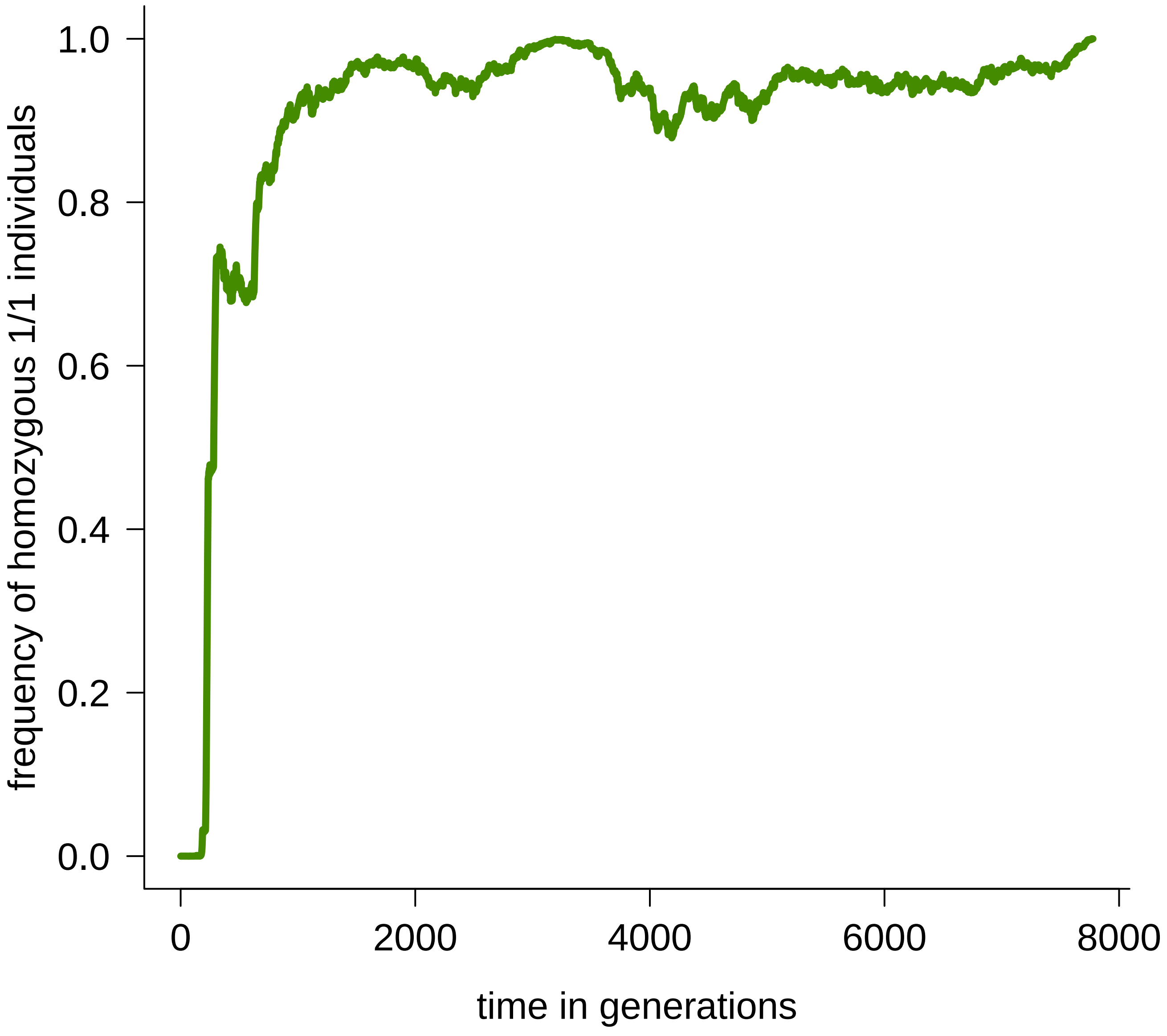

Figure 8: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 0.1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.1; the dominance mechanism is r-shape with $z = 2\mathbb{1}_{\{g>0\}}$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; ; results from 10^2 experiments with one fixation

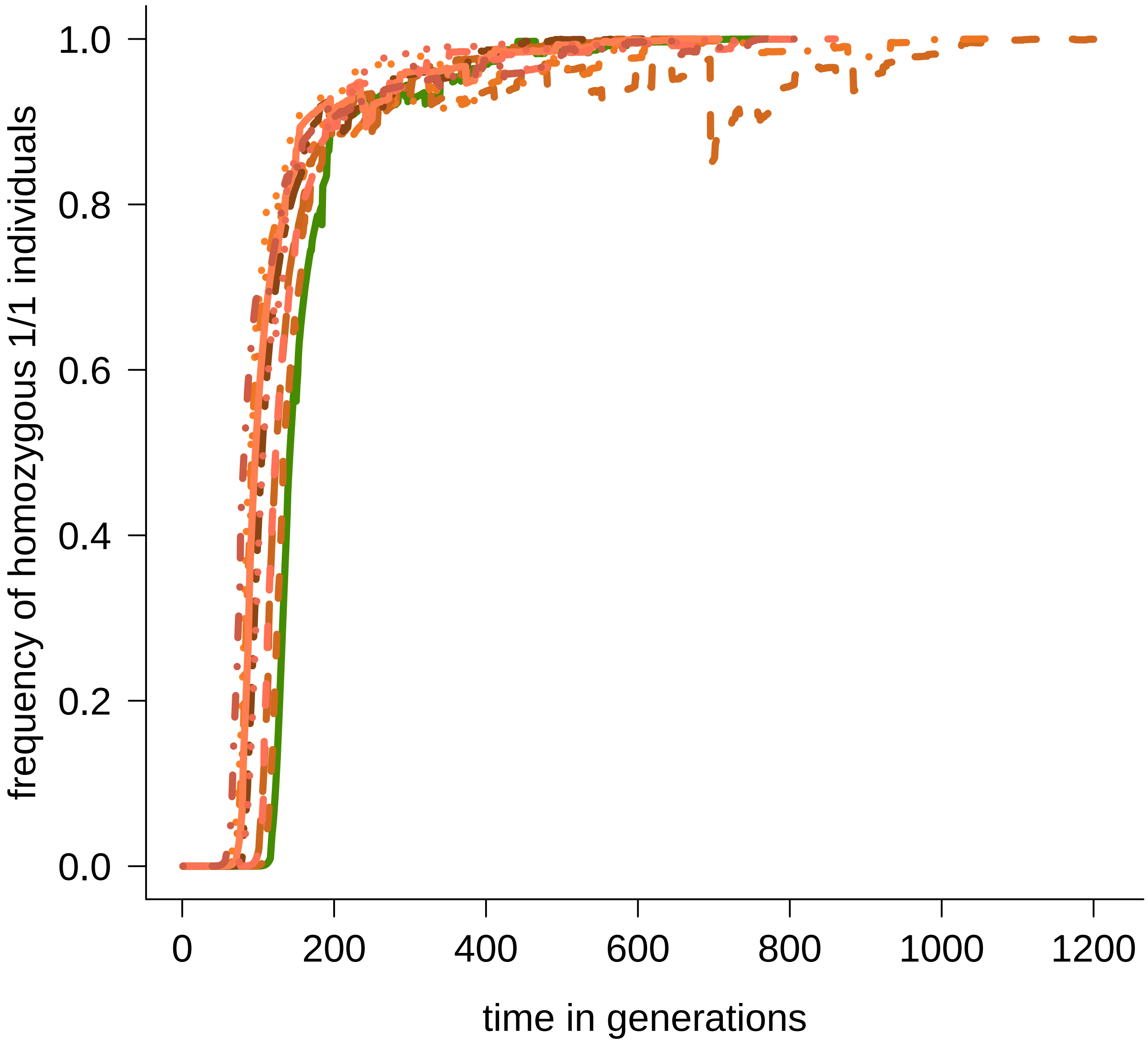

Figure 9: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 0.1$, selection strength $s = 0.1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.01; the dominance mechanism is r-shape with $z = 2\mathbb{1}_{\{g>0\}}$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; ; results from 10^2 experiments with ten fixations and average time to fixation 818.8 and stdev 191.6

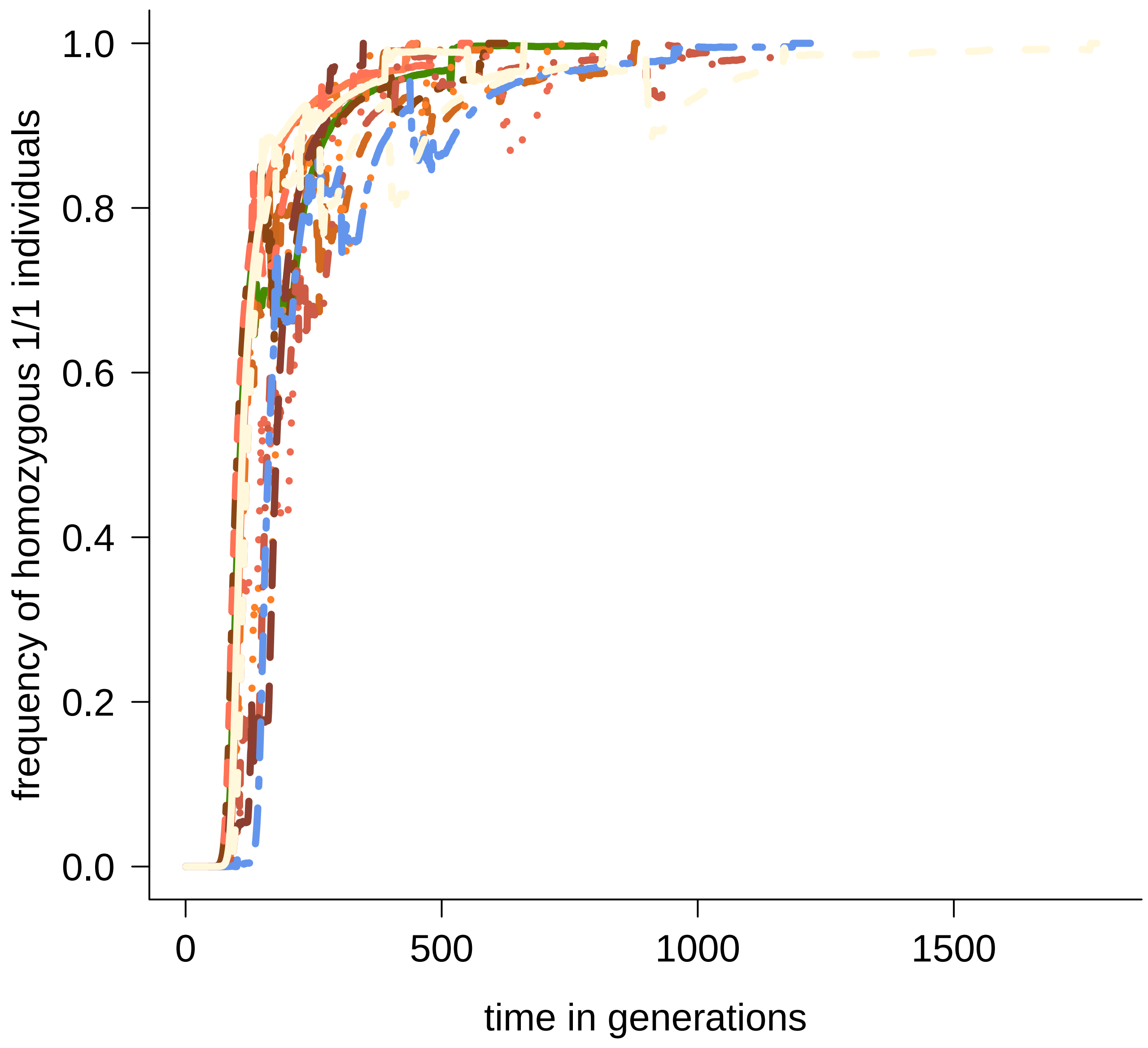

Figure 10: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 0.1$, bottleneck 10^2 and probability of a bottleneck in a given generation 0.01; the dominance mechanism is r-shape with $z = 2\mathbb{1}_{\{g>0\}}$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; ; results from 10^2 experiments ; fourteen fixations with 801.0000 380.229

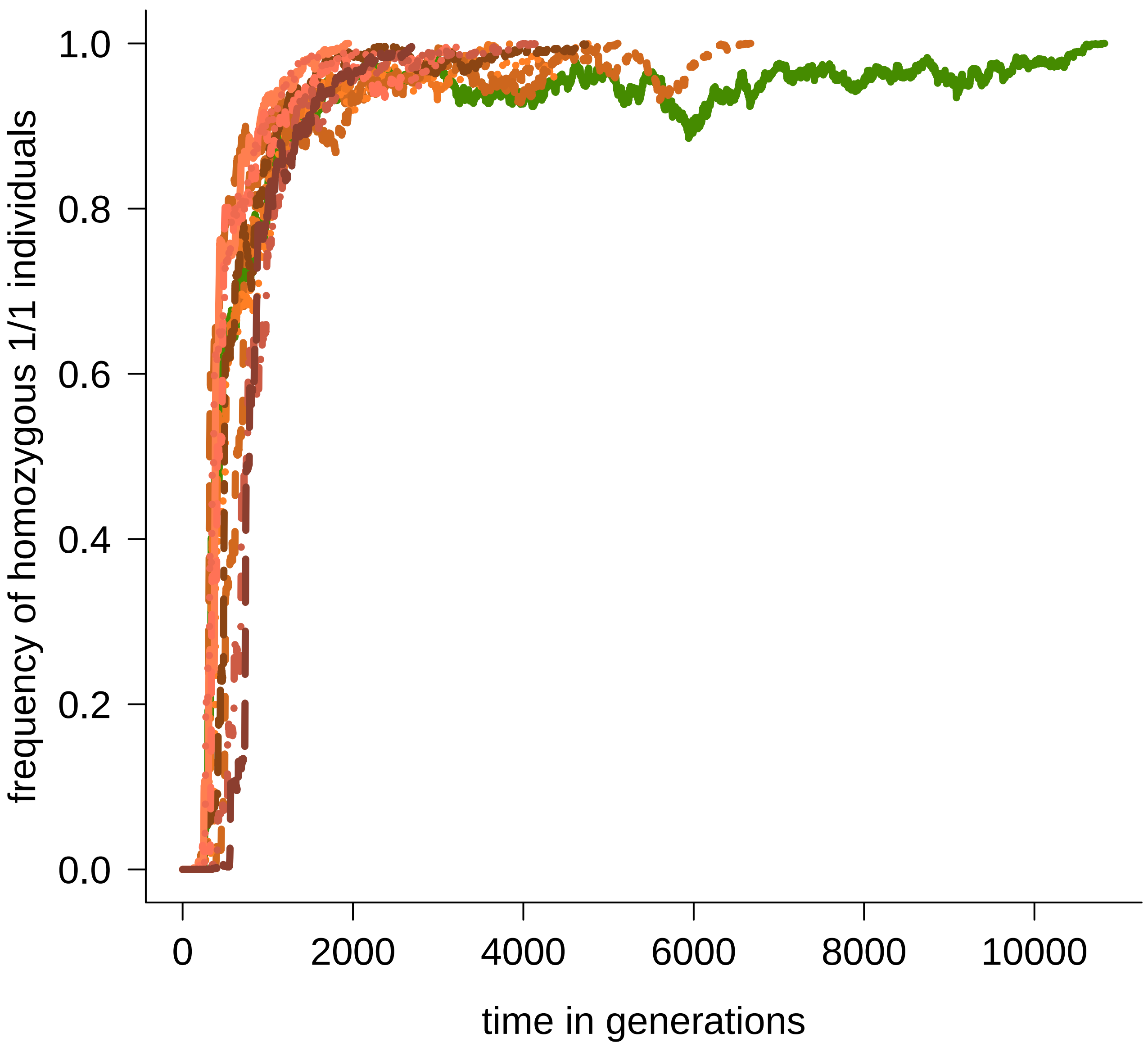

Figure 11: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 1.05$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 0.1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.1; the dominance mechanism is r-shape with $z = 2\mathbb{1}_{\{g>0\}}$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; eleven fixations out of 10^3 experiments with average time to fixation 4692.2 and stdev 2403.0

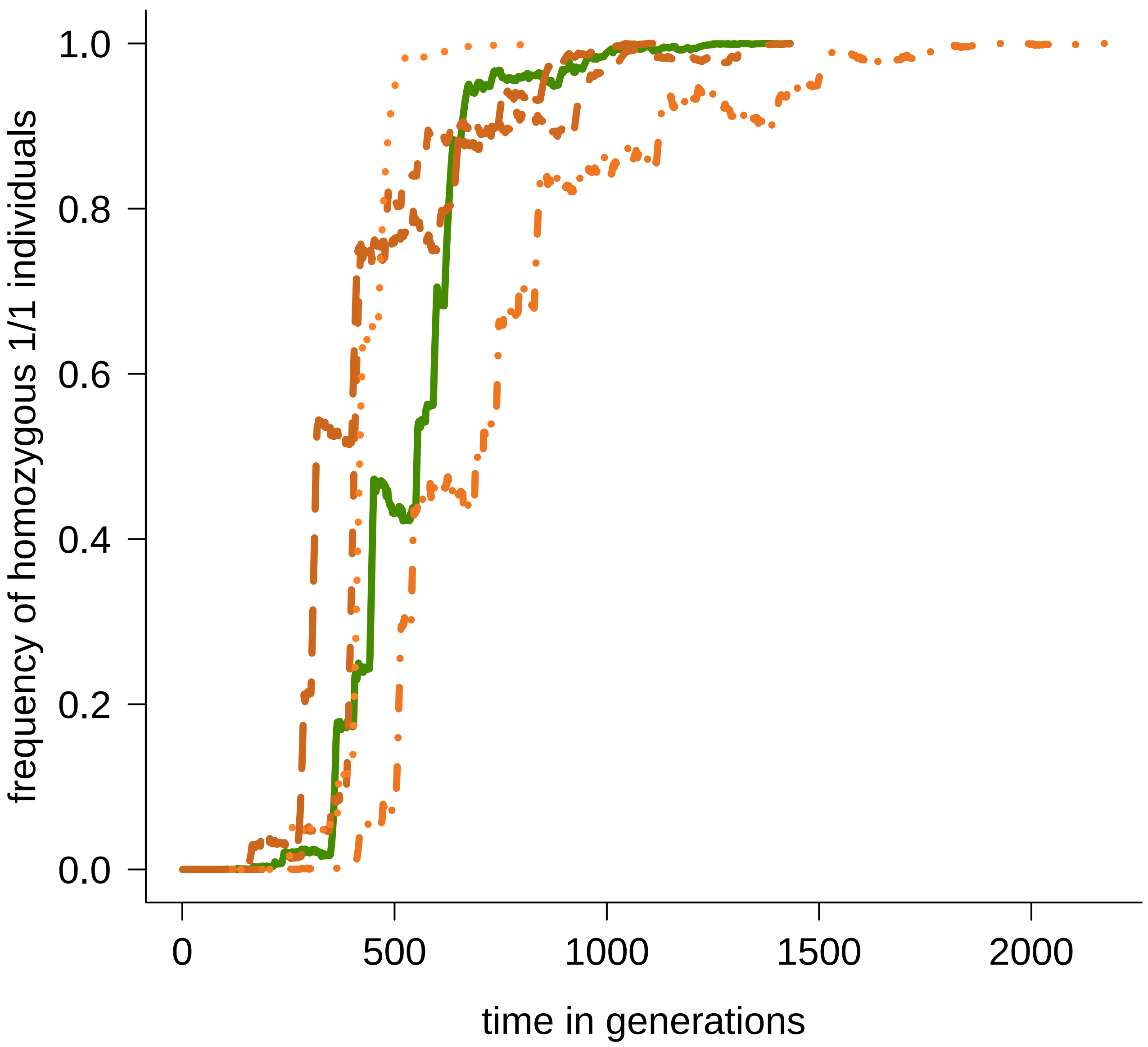

Figure 12: Examples of excursions to fixation of the homozygous type 1/1 at a single locus in a diploid population evolving according to random sweepstakes Eq (1) and randomly occurring bottlenecks with fixed capacity $2N = 10^6$, cutoff $\Psi_N = 2N$, $\alpha_1 = 0.75$, $\alpha_2 = 3$, $\varepsilon_N = 1/N$, selection strength $s = 0.1$, bottleneck 10^4 and probability of a bottleneck in a given generation 0.1; the dominance mechanism is linear with $z = g$ for $g \in \{0, 1, 2\}$, optimal type 1/1 or $z_0 = 2$; five fixations out of 10^3 experiments with average time to fixation 1397.2 and stdev 508.9

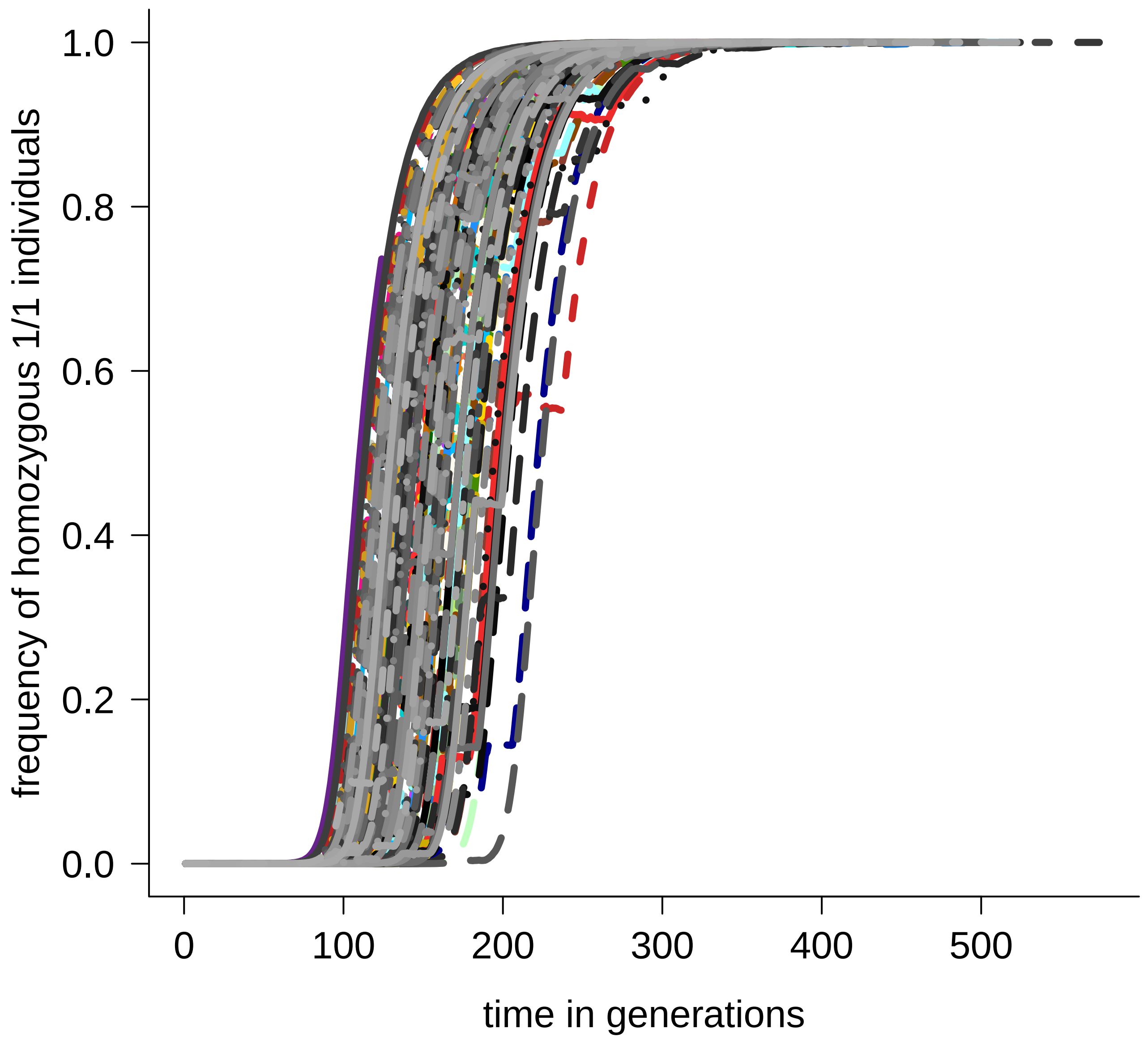
frequency of homozygous 1/1 individuals
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500
time in generations

Figure 13: ex

6 conclusion

[3] consider a model of selection in a diploid population in which the homozygous aa individuals are at a selective disadvantage to AA individuals and heterozygous aA individuals are at a selective disadvantage to both homozygotes.

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] Alison Etheridge and Sarah Penington. Genealogies in bistable waves. *arXiv preprint arXiv:2009.03841*, 2020.

[4] Brian W Kernighan and Dennis M Ritchie. The c programming language, 1988.

[5] 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

⟨ add juveniles for a given parent pair 12 ⟩ Used in chunk 19.

⟨ bottleneck 14 ⟩ Used in chunk 19.

⟨ generate pool juveniles 13 ⟩ Used in chunk 19.

⟨ gslrng 7 ⟩ Used in chunk 19.

⟨ includes 5 ⟩ Used in chunk 19.

⟨ initialise cdf 18 ⟩ Used in chunk 19.

⟨ mass function 17 ⟩ Used in chunk 19.

⟨ onestep 15 ⟩ Used in chunk 19.

⟨ sample parent type 8 ⟩ Used in chunk 19.

⟨ sample random number juveniles 10 ⟩ Used in chunk 19.

⟨ struct 6 ⟩ Used in chunk 19.

⟨ trajectory 16 ⟩ Used in chunk 19.

⟨ typejuvenile 9 ⟩ Used in chunk 19.

⟨ weigth 11 ⟩ Used in chunk 19.