Forward-time simulations using simuPOP, a tutorial

Bo Peng, Ph.D.

Department of Epidemiology
U.T. M.D. Anderson Cancer Center
Houston, TX

June 6th, 2007
Programmers' Cross Training
U.T. M.D. Anderson Cancer Center

THE UNIVERSITY OF TEXAS MD ANDERSON CANCER CENTER Making Cancer History®

outline

1 What is simuPOP

2 An example

3 simuPOP components

4 A real-world application

Outline

1 What is simuPOP

- Forward- and backward-time simulation
- Features of simuPOP
- Applications
- Availability

simuPOP is ...

A forward-time population genetics simulation environment

simuPOP is ...

A forward-time population genetics simulation environment

- A population genetics simulation program

simuPOP is ...

A forward-time population genetics simulation environment

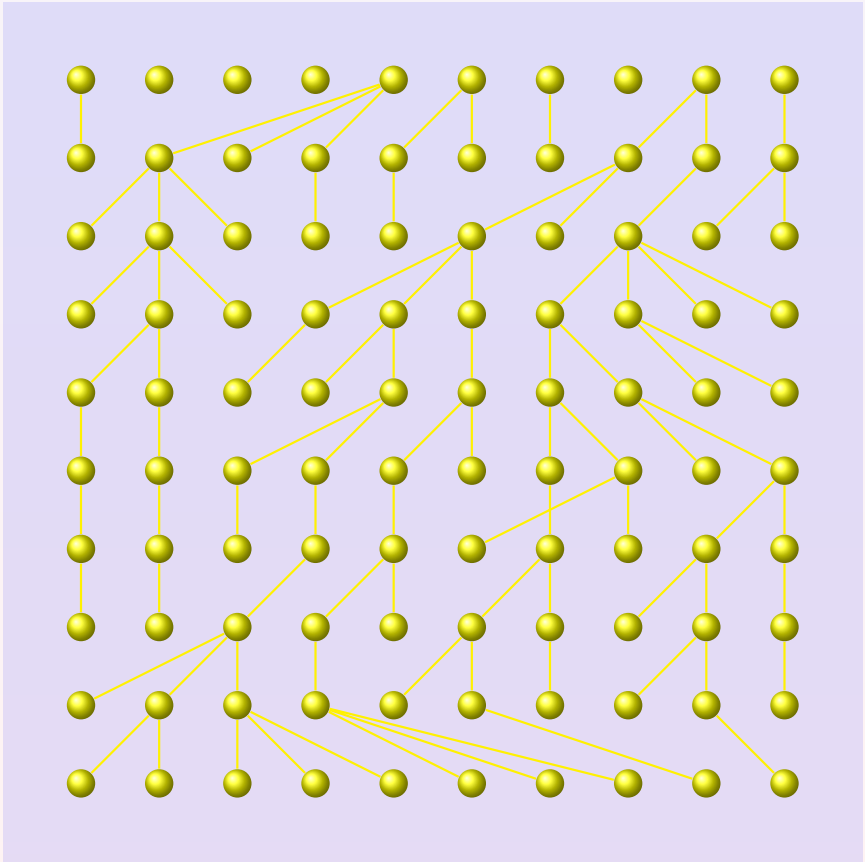
- A population genetics simulation program
- Not coalescent-based

simuPOP is ...

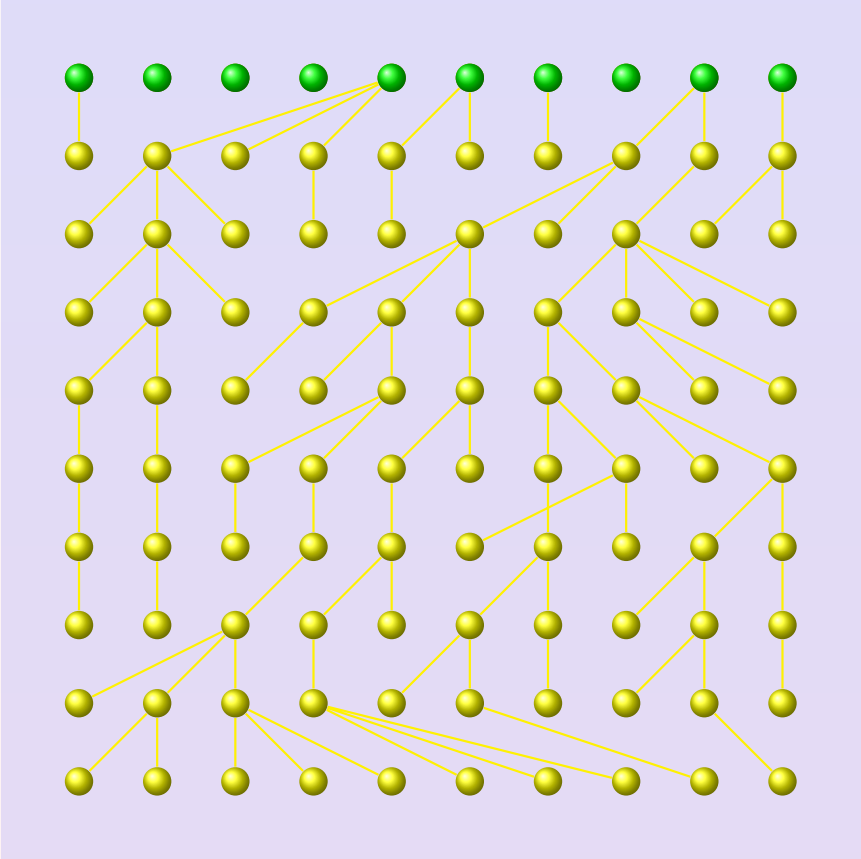
A forward-time population genetics simulation environment

- A population genetics simulation program
- Not coalescent-based
- Based on an object-oriented scripting language (Python)

Forward-time simulation

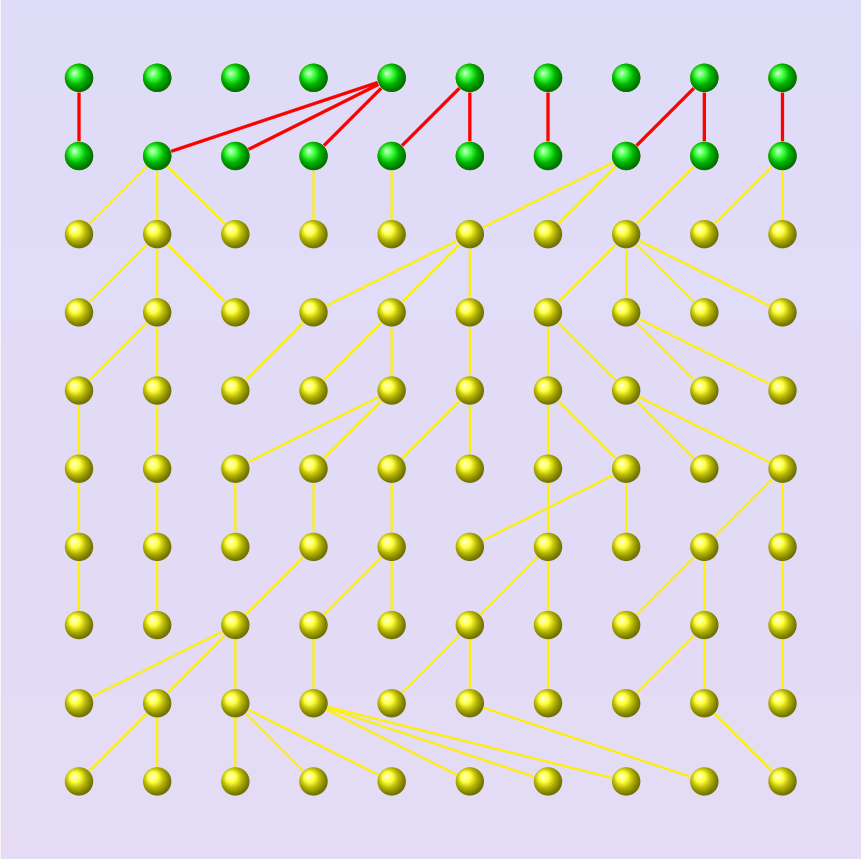


Forward-time simulation



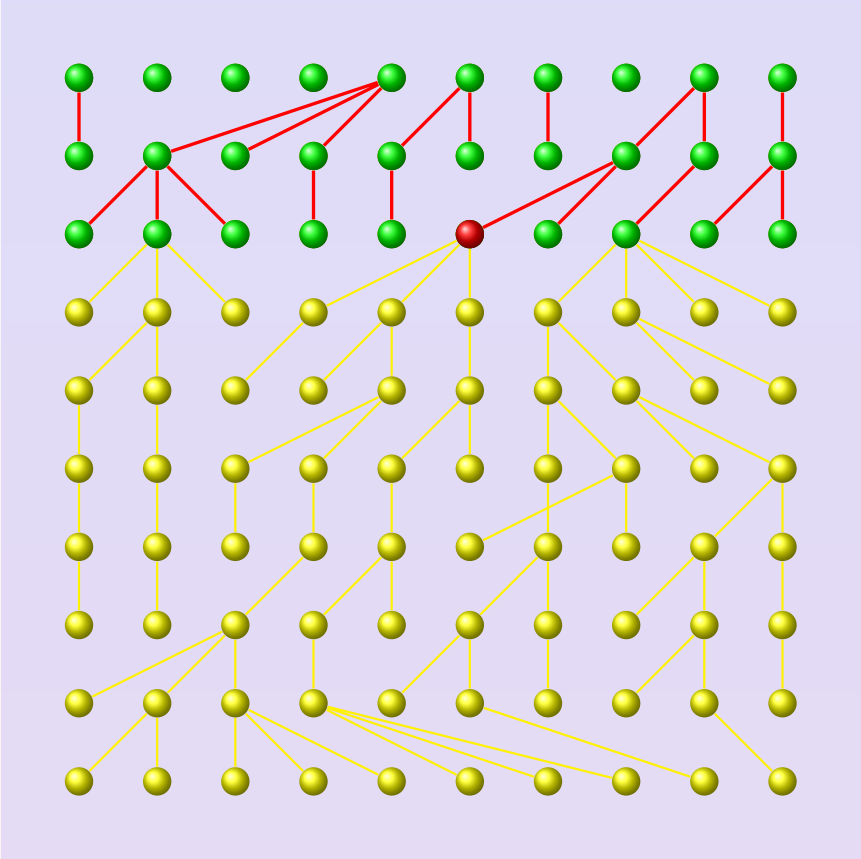
- Start from an initial population

Forward-time simulation



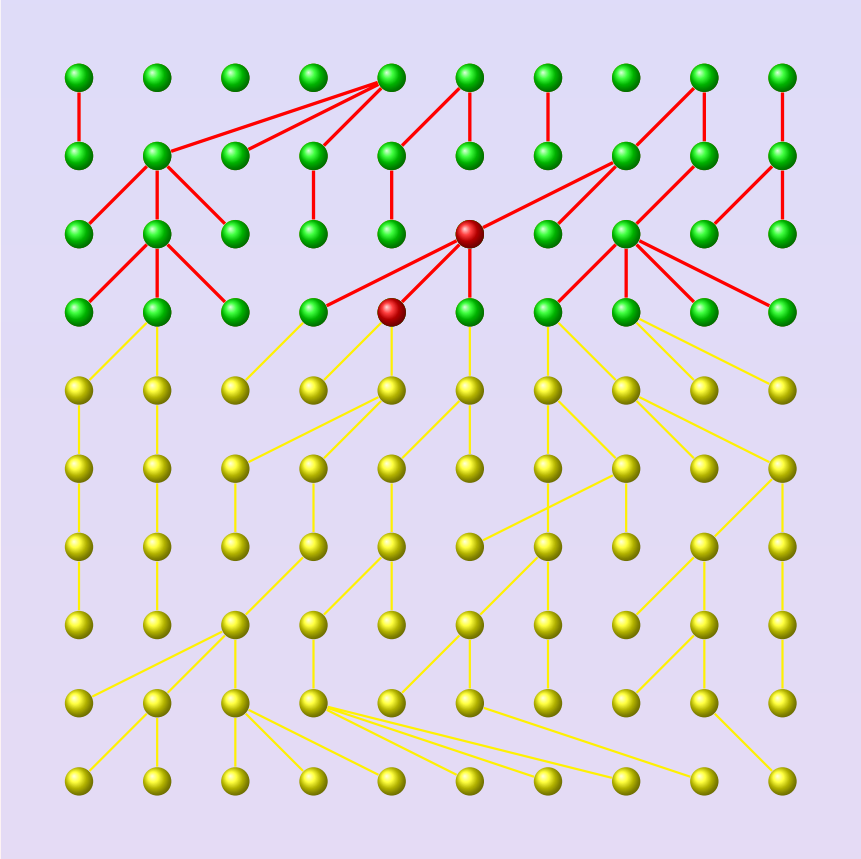
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



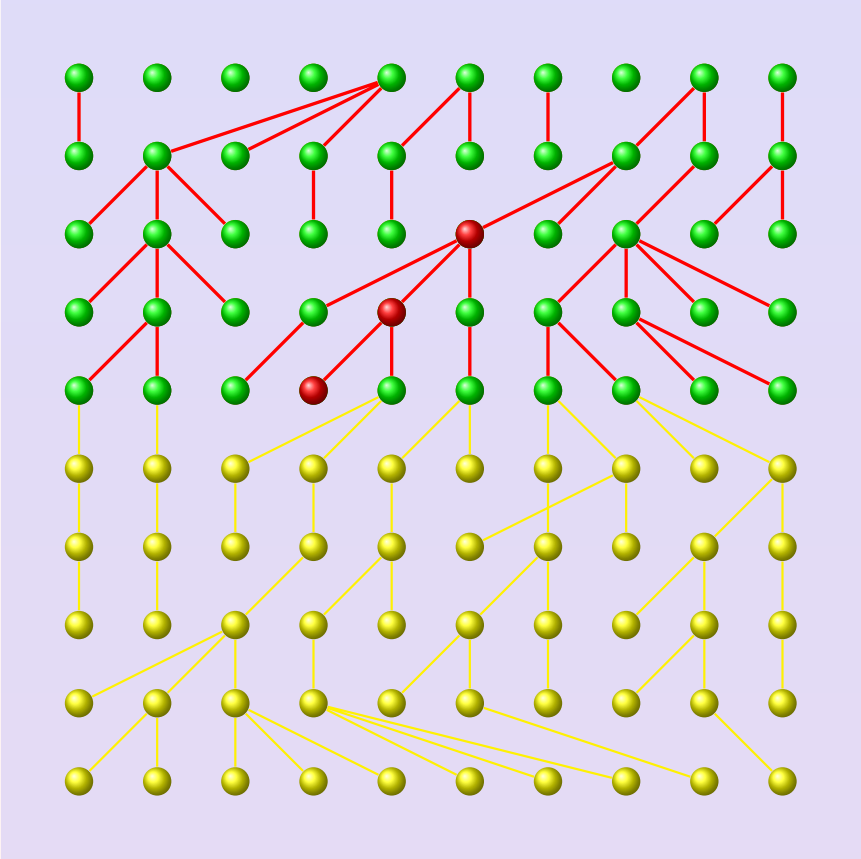
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



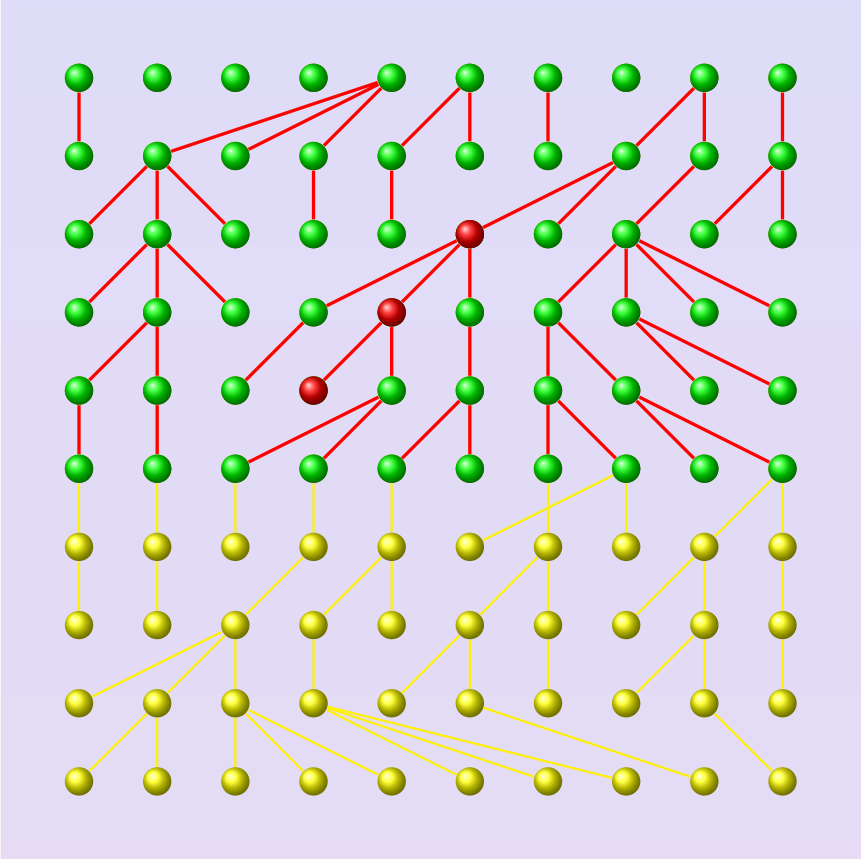
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



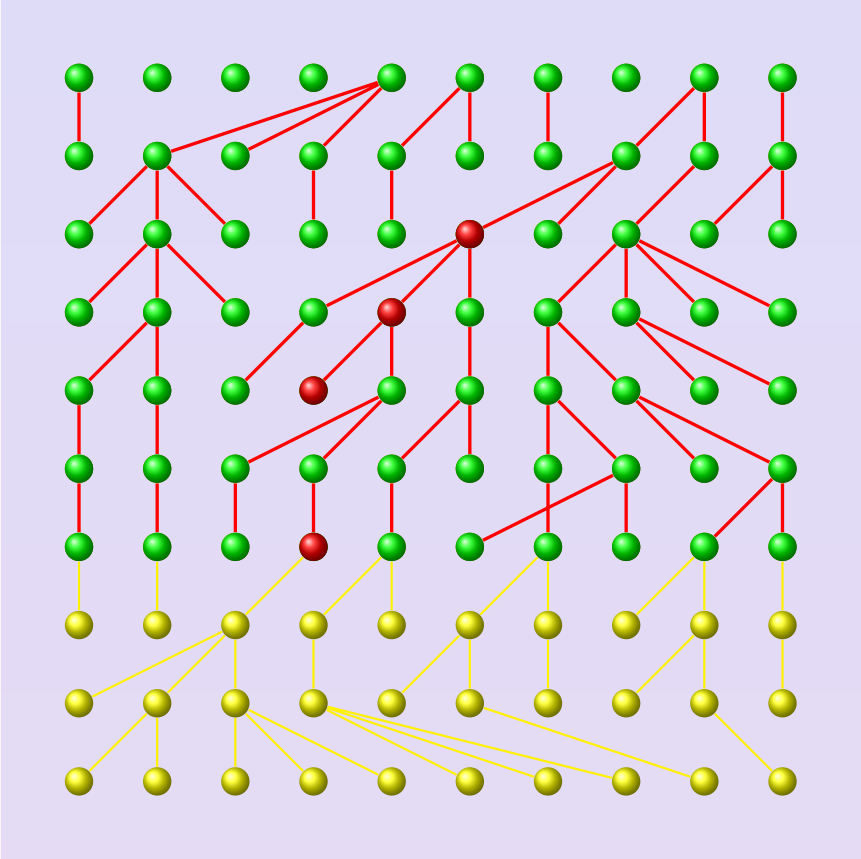
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



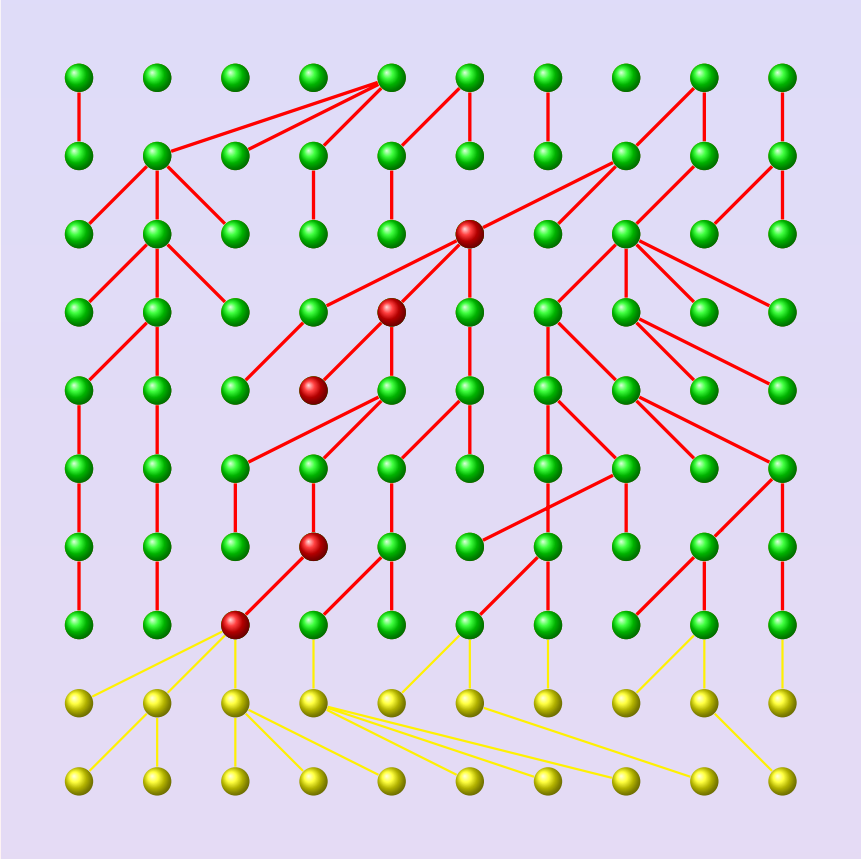
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



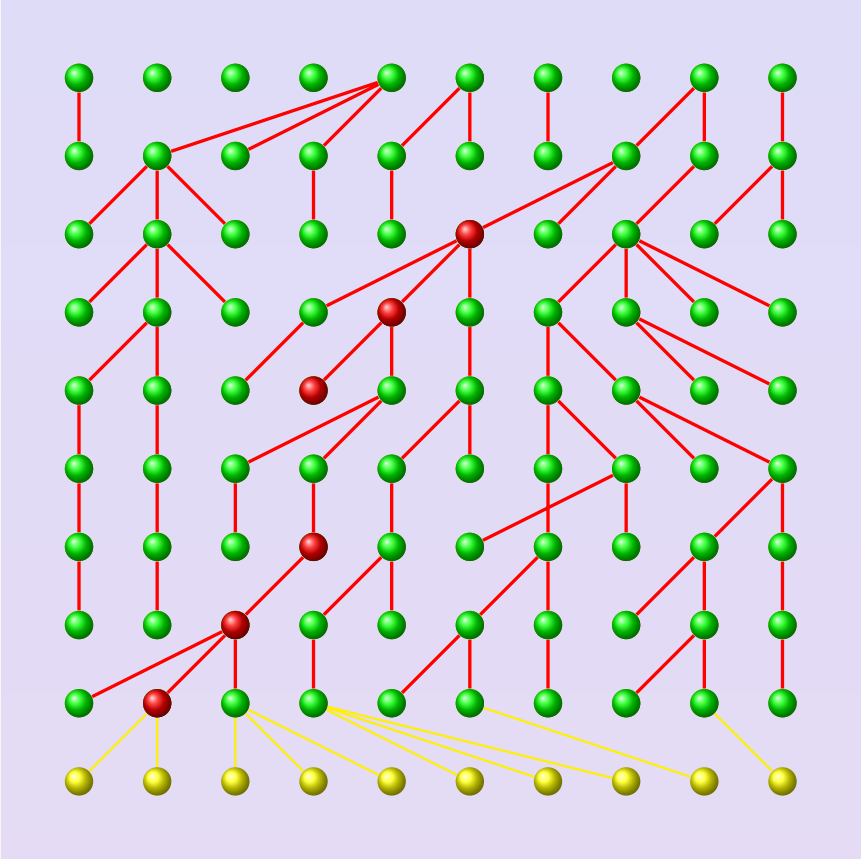
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



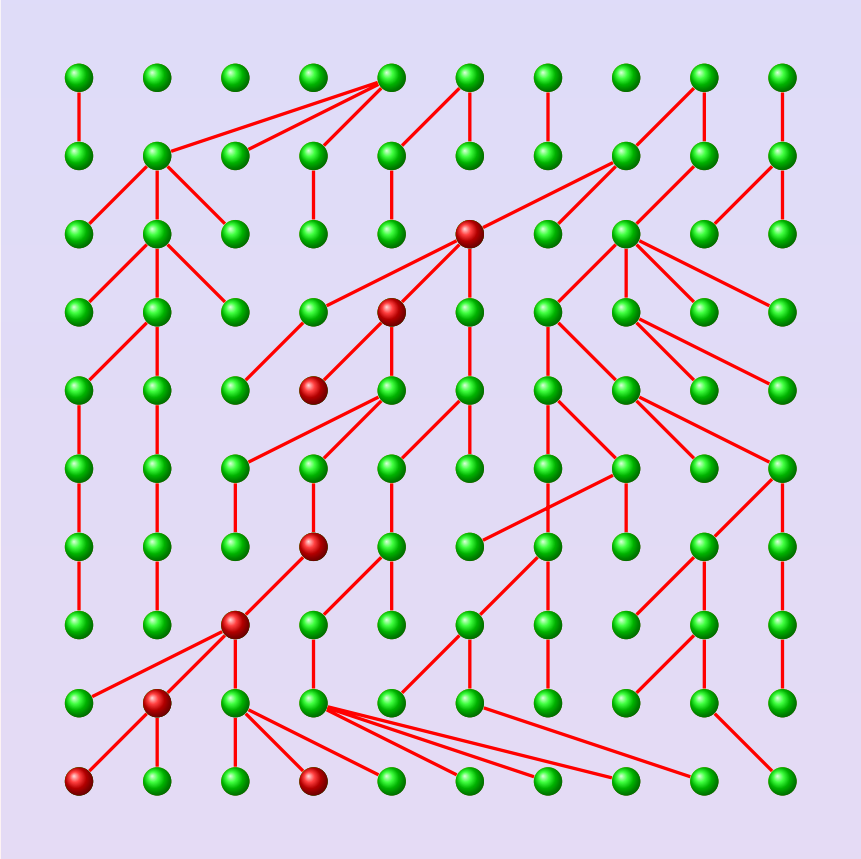
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



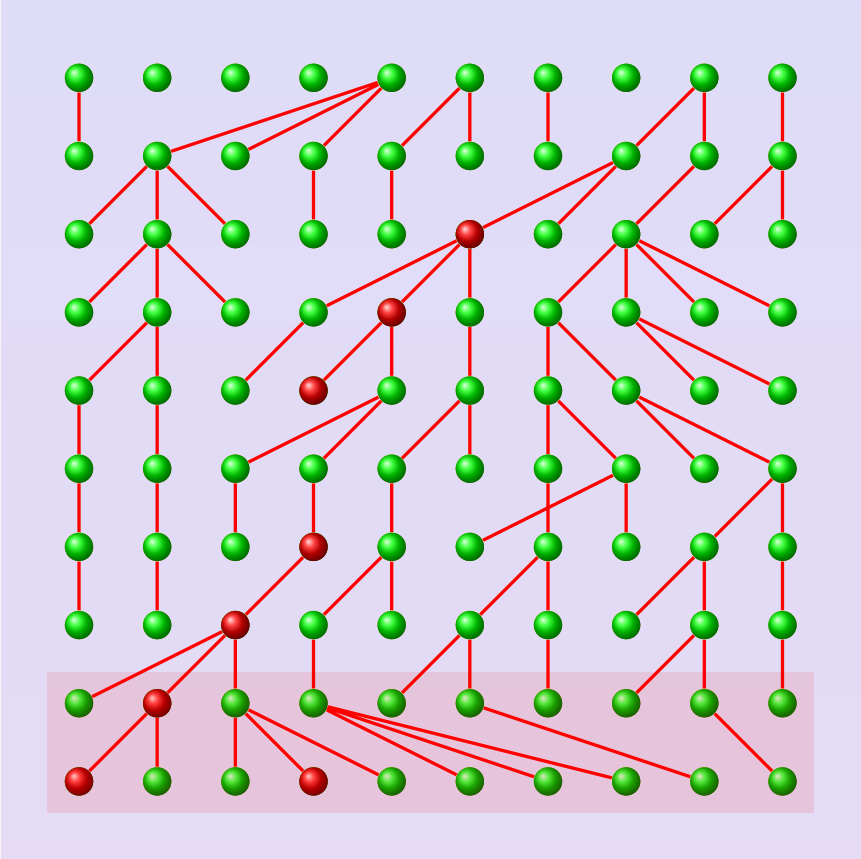
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation



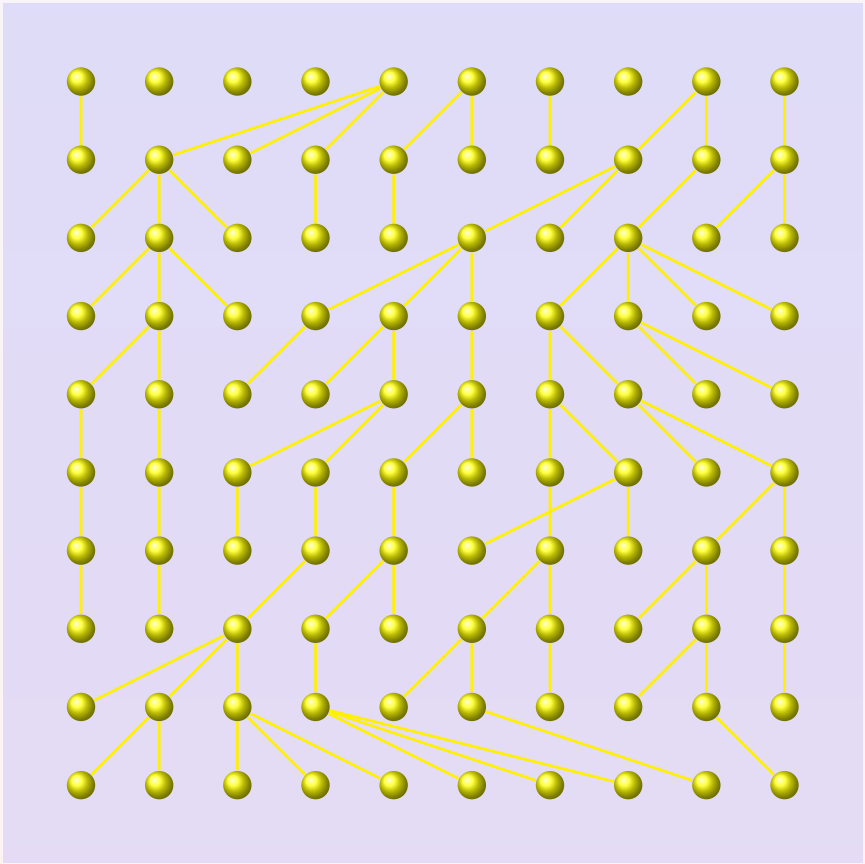
- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects

Forward-time simulation

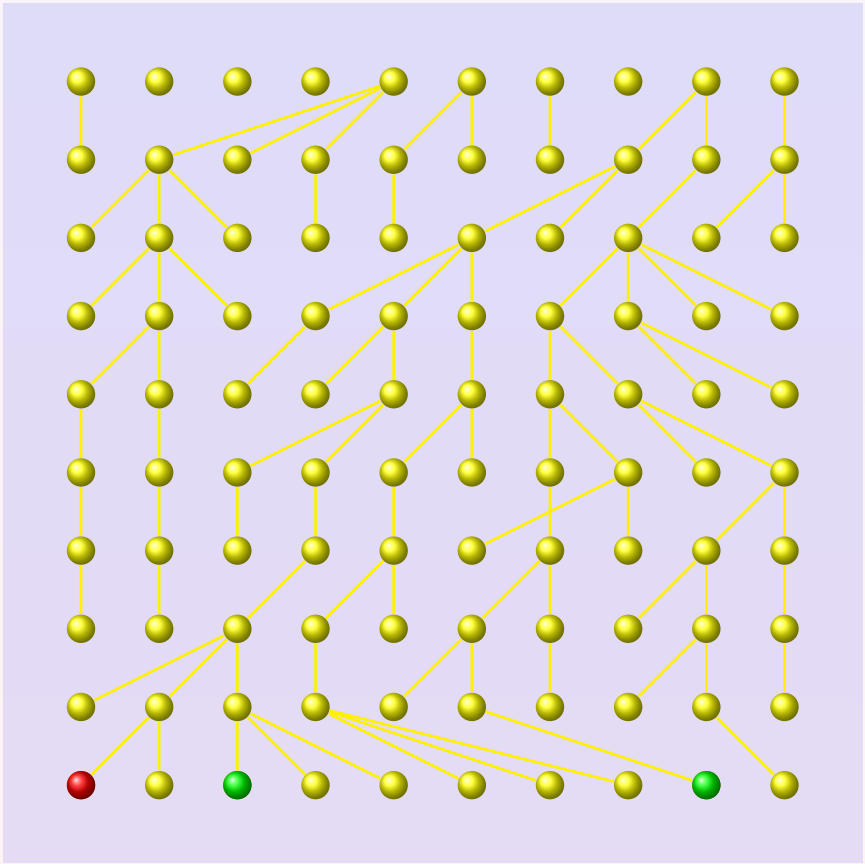


- Start from an initial population
- Evolve forward in time, generation by generation, subject to certain number of genetic and/or demographic effects
- Samples are collected from the last several generations

Backward-time simulation

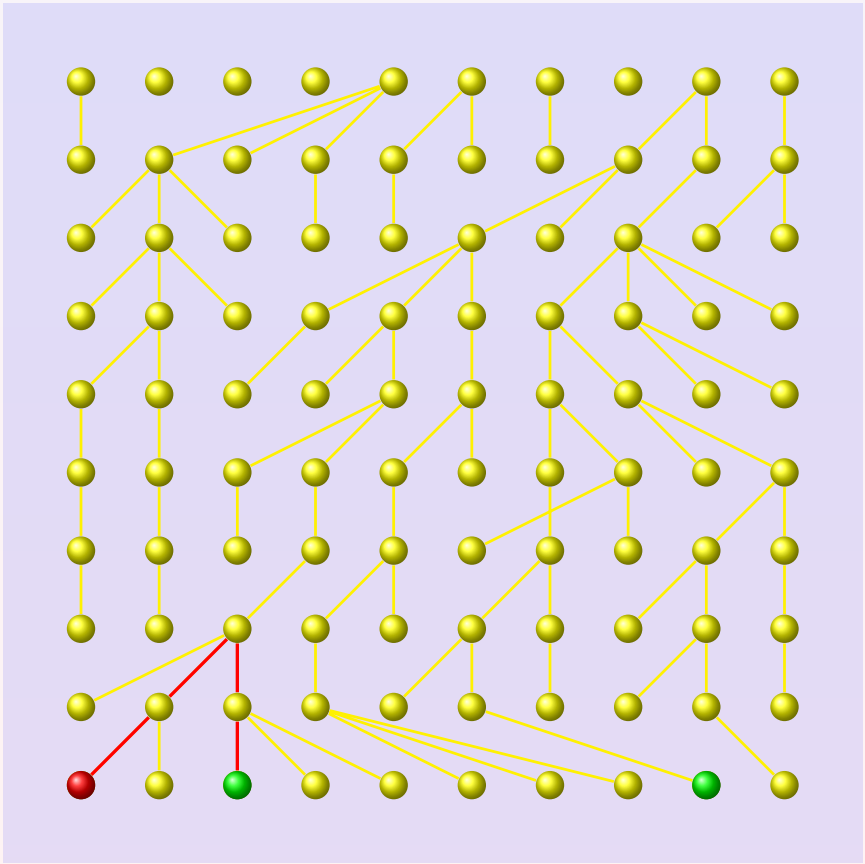


Backward-time simulation



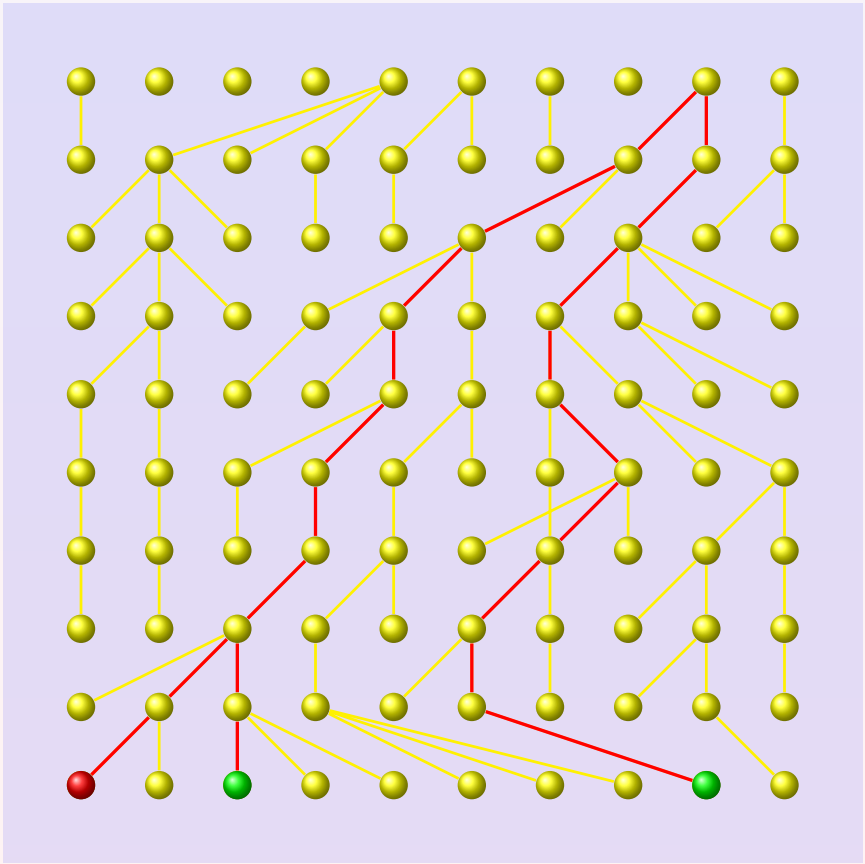
- Start from a sample with unknown genotype

Backward-time simulation



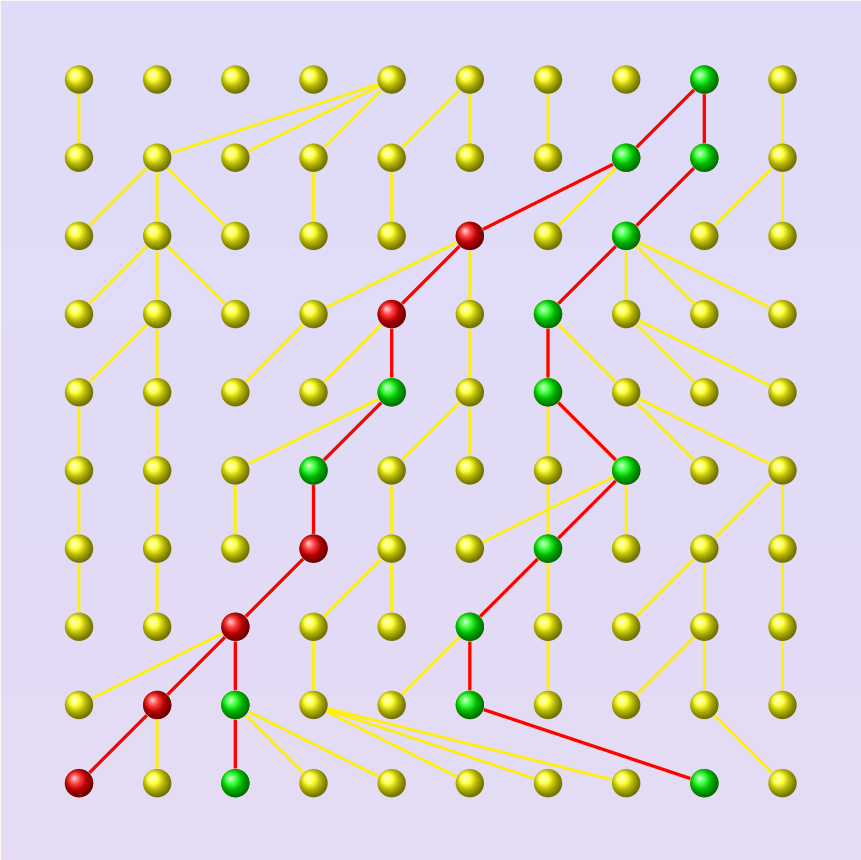
- Start from a sample with unknown genotype
- Coalesce individuals until the most recent common ancestor of all individuals is found

Backward-time simulation



- Start from a sample with unknown genotype
- Coalesce individuals until the most recent common ancestor of all individuals is found

Backward-time simulation



- Start from a sample with unknown genotype
- Coalesce individuals until the most recent common ancestor of all individuals is found
- Starting from the MRCA, proceed forward in time and fill the genotype of each individual

Forward vs. backward-time simulations

Backward-time

- Sample based, efficient

Forward-time

- Population based, inefficient

Forward vs. backward-time simulations

Backward-time

- Sample based, efficient
- Limited selection, recombination models and mating schemes

Forward-time

- Population based, inefficient
- Can simulate almost arbitrary evolutionary scenarios

Forward vs. backward-time simulations

Backward-time

- Sample based, efficient
- Limited selection, recombination models and mating schemes
- Can not study population properties, or properties of ancestral generations

Forward-time

- Population based, inefficient
- Can simulate almost arbitrary evolutionary scenarios
- Can study population properties and ancestral generations

Forward vs. backward-time simulations

Backward-time

- Sample based, efficient
- Limited selection, recombination models and mating schemes
- Can not study population properties, or properties of ancestral generations
- Used mostly for sample generation

Forward-time

- Population based, inefficient
- Can simulate almost arbitrary evolutionary scenarios
- Can study population properties and ancestral generations
- Not limited to sample generation

Forward-time simulation programs

For specific applications

- Easy to write simple simulations
- Difficult to write complicated simulations
- A few programs are available (EasyPOP, FPG, Nemo, ...), easy to use if they happen to fit your need

Forward-time simulation programs

For specific applications

- Easy to write simple simulations
- Difficult to write complicated simulations
- A few programs are available (EasyPOP, FPG, Nemo, ...), easy to use if they happen to fit your need

For general purposes

- Difficult to write
- Easy to set up complicated simulations
- simuPOP fits in this category

What simuPOP does

simuPOP provides

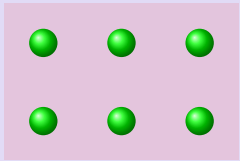
- a large number of functions to manipulate populations
  copy, split, merge, manipulate individual genotypes, determine affection status, save to and load from various formats, generate sample, ...

What simuPOP does

simuPOP provides

- a large number of functions to manipulate populations copy, split, merge, manipulate individual genotypes, determine affection status, save to and load from various formats, generate sample, ...
- and a mechanism to evolve populations forward in time subject to arbitrary demographic and genetic forces such as population size changes, mutation, migration, recombination, selection, ...

Illustration of the evolutionary process



Parental generation

Illustration of the evolutionary process

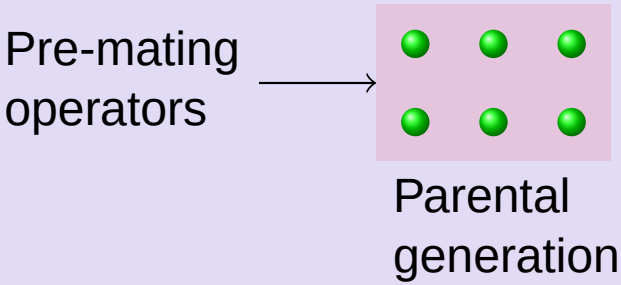


Illustration of the evolutionary process

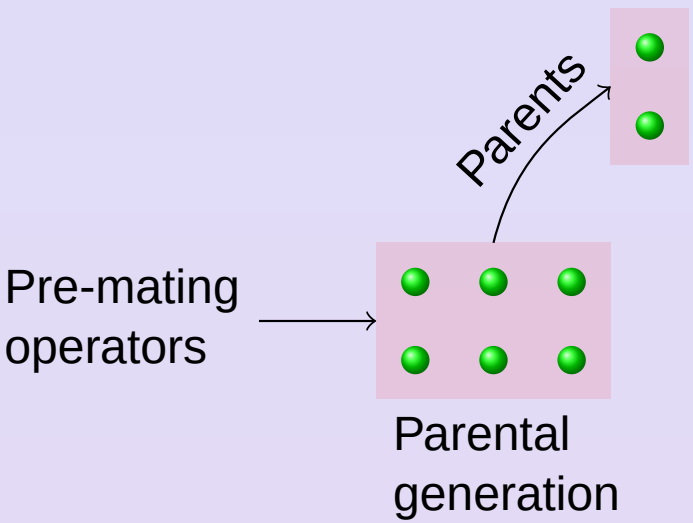


Illustration of the evolutionary process

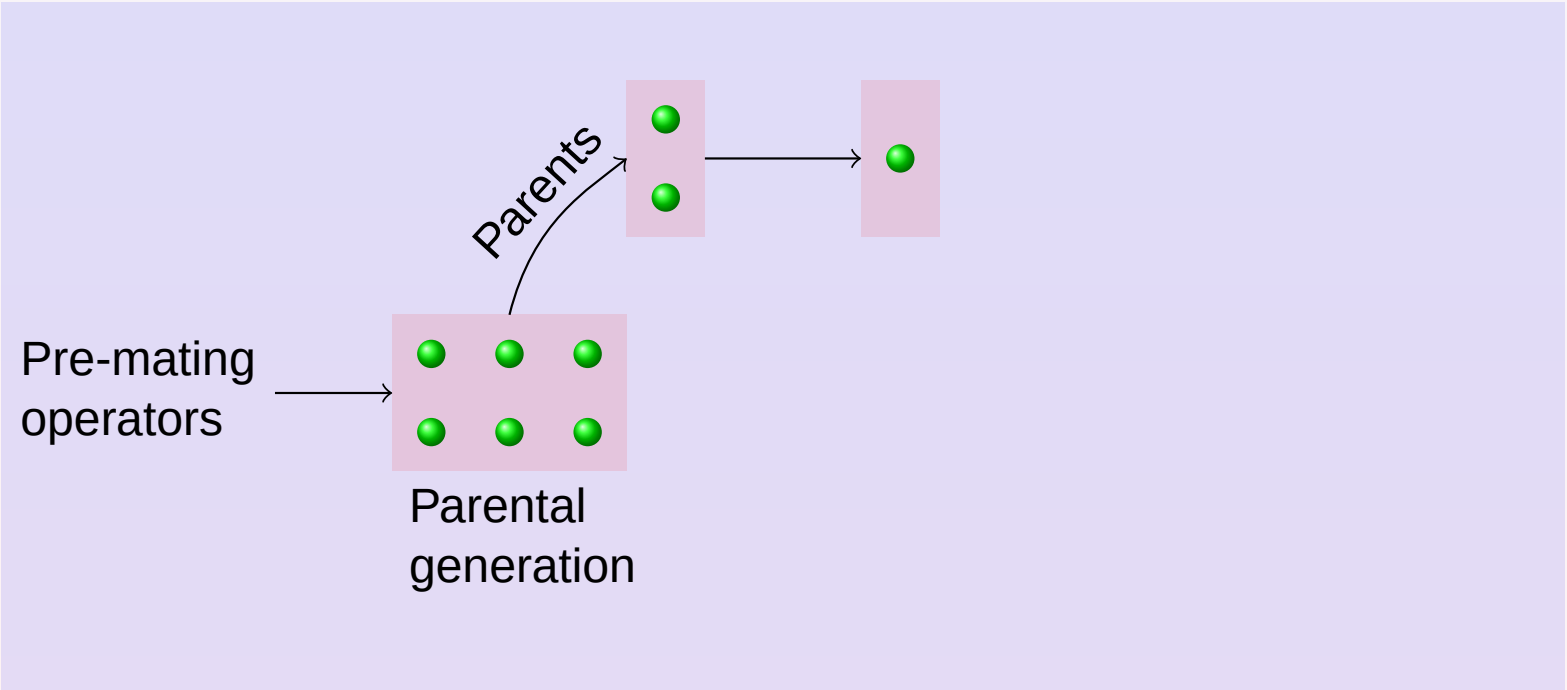


Illustration of the evolutionary process

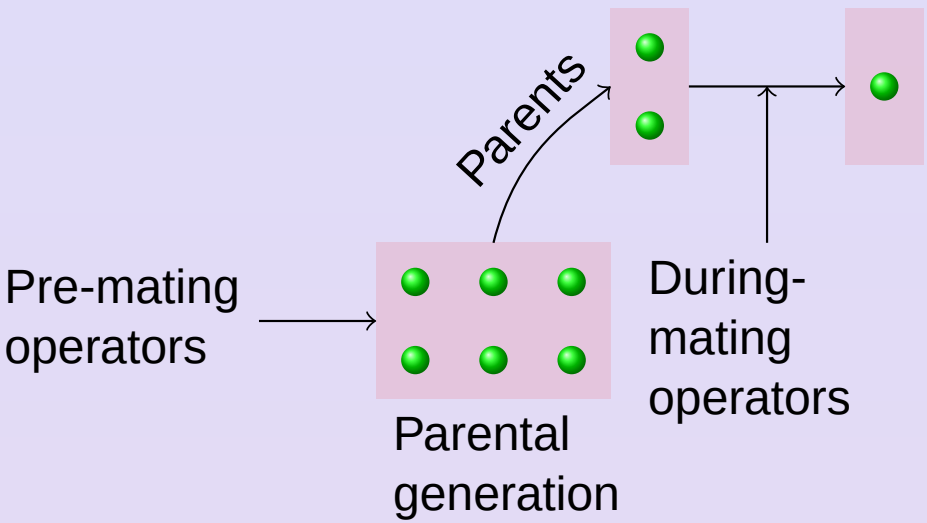


Illustration of the evolutionary process

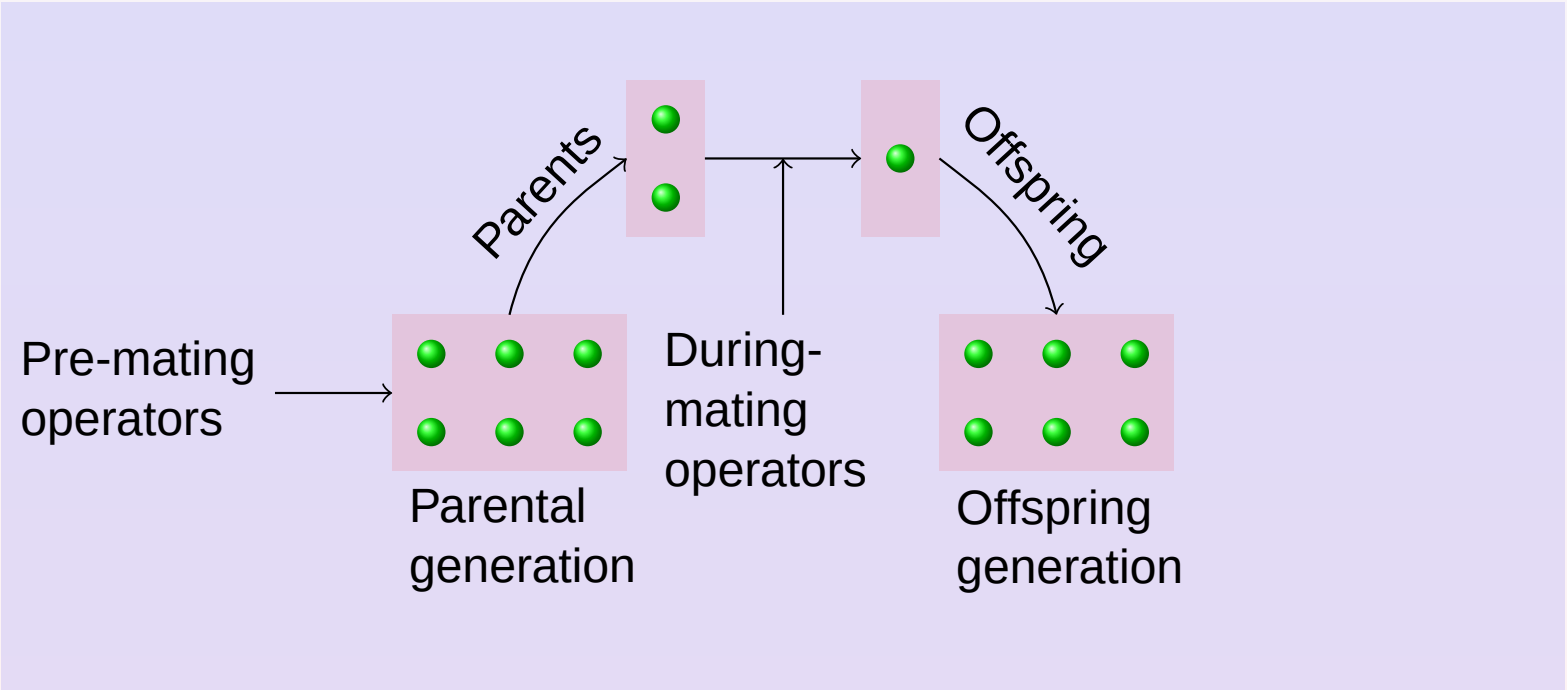
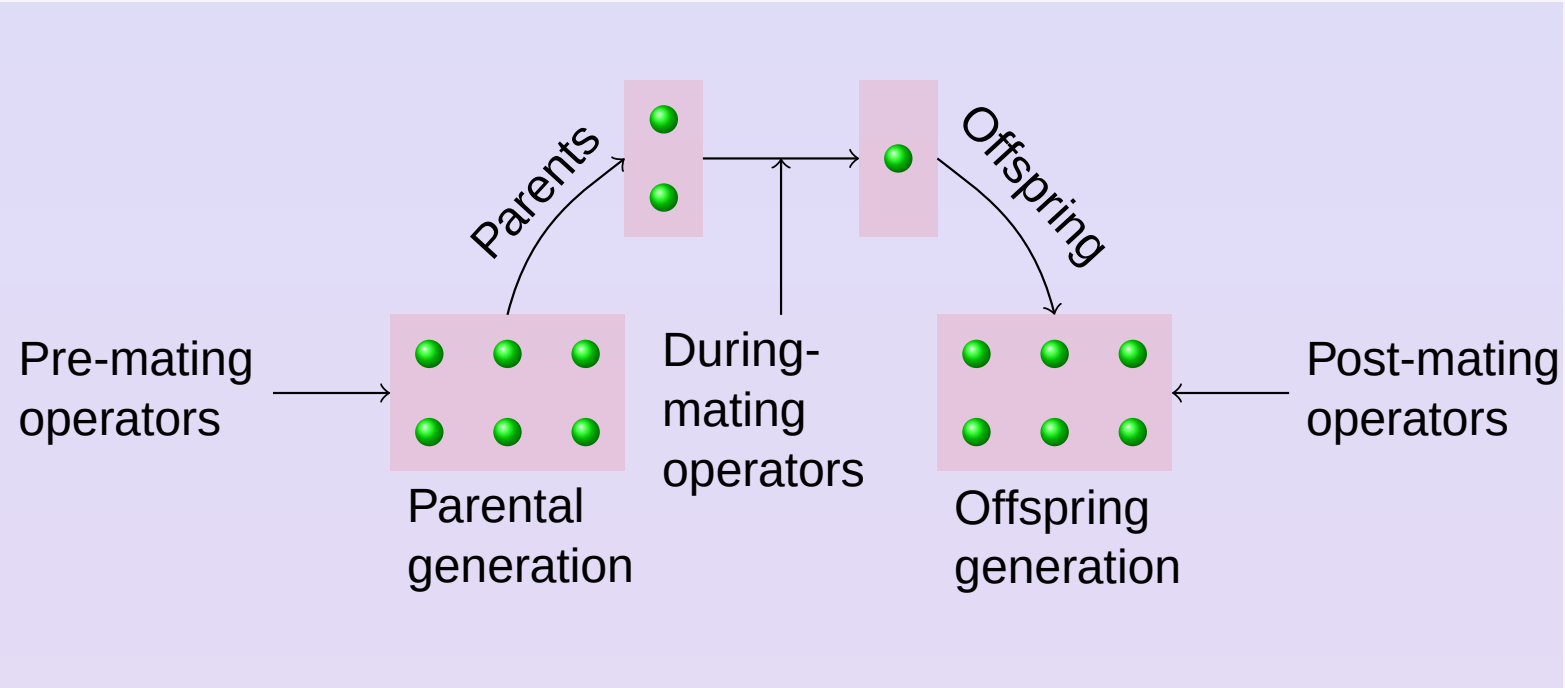


Illustration of the evolutionary process



What distinguishes simuPOP from others

scripting simuPOP is provided as a set of Python modules, and is therefore backed by a full-blown object-oriented programming language.

flexibility simuPOP does not impose any limit on the size of genome, population, demographic model, etc. Using a large number of standard and hybrid (Python-assisted) operators, users can simulate almost arbitrarily complex evolutionary processes.

integration Owing to the 'glue language' nature of Python, it is easy to integrate simuPOP with other languages and programs.

I like it, but, oohm, why Python??

- The core of simuPOP is written in C++ for efficiency
- Python is the glue language, a wrapper of the core
- Python is used to write simuPOP extensions (user interface etc)
- The core sometimes calls Python (Python operators) for maximum flexibility

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena
- Study the impact of genetic and demographic forces on the evolution of a population

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena
- Study the impact of genetic and demographic forces on the evolution of a population
- Study the evolution of (complex) genetic diseases

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena
- Study the impact of genetic and demographic forces on the evolution of a population
- Study the evolution of (complex) genetic diseases
- Simulate samples to validate gene-mapping methods

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena
- Study the impact of genetic and demographic forces on the evolution of a population
- Study the evolution of (complex) genetic diseases
- Simulate samples to validate gene-mapping methods
- Study ascertainment methods in simulated populations

This is fun, but is it useful?

simuPOP can simulate the change of the genetic composition of a population in a complicated evolutionary process. It can be used to

- Demonstrate population genetics phenomena
- Study the impact of genetic and demographic forces on the evolution of a population
- Study the evolution of (complex) genetic diseases
- Simulate samples to validate gene-mapping methods
- Study ascertainment methods in simulated populations
- ...

Simulations of complex human diseases

Backward-time

- Haploid only

Forward-time

- No limit on ploidy

Simulations of complex human diseases

Backward-time

- Haploid only
- Additive selection and penetrance models

Forward-time

- No limit on ploidy
- Arbitrary selection and penetrance models

Simulations of complex human diseases

Backward-time

- Haploid only
- Additive selection and penetrance models
- One disease susceptibility locus

Forward-time

- No limit on ploidy
- Arbitrary selection and penetrance models
- Multiple DSL with interaction

Simulations of complex human diseases

Backward-time

- Haploid only
- Additive selection and penetrance models
- One disease susceptibility locus
- Generate independent samples of fixed format

Forward-time

- No limit on ploidy
- Arbitrary selection and penetrance models
- Multiple DSL with interaction
- Generate multi-generation populations

Availability

- simuPOP website: `http://simupop.sourceforge.net`
- Mailing list: `simupop-list@lists.sourceforge.net`
- License: GPL 2.0
- Platforms: `all OS on which Python is available`
- Monthly release, currently at 0.7.10
- Documentation: *simuPOP User's Guide* and *simuPOP Reference Manual*

Outline

2 An example

- An example
- Visualization with R

A simple example


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         pyEval(r"'%3d    ' % gen", rep=0, step=10),
...         pyEval(r"'%f    ' % LD[0][1]", step=10),
...         pyEval(r"'\n'", rep=REP_LAST, step=10)
...     ],
...     end=100
... )
```

Output of the example


```
  0    0.202805    0.198518    0.200676
 10    0.068618    0.057934    0.092528
 20    0.031660    0.014256    0.033041
 30    0.010710    0.002449    0.006295
 40    0.031548    0.000453    0.011609
 50    0.004170    0.003946    0.005345
 60    0.012041    0.015075    0.007308
 70    0.008850    0.014041    0.012417
 80    0.017006    0.012987    0.013742
 90    0.013991    0.000250    0.005159
100    0.010028    0.021751    0.009032
```

simuPOP modules


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
```

Import the default simuPOP module

population


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
```

Create a `population` of 1000 `diploid` individuals, each having two `loci` on the first chromosome

simulator and mating scheme


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
```

Create a `simulator` that has one replicate of this population, and a random mating scheme

Operators!


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         pyEval(r"'%3d    ' % gen", rep=0, step=10),
...         pyEval(r"'%f    ' % LD[0][1]", step=10),
...         pyEval(r"'\n'", rep=REP_LAST, step=10)
...     ],
...     end=100
... )
```

`initByValue` is applied before evolution

Operators!


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         pyEval(r"'%3d    ' % gen", rep=0, step=10),
...         pyEval(r"'%f    ' % LD[0][1]", step=10),
...         pyEval(r"'\n'", rep=REP_LAST, step=10)
...     ],
...     end=100
... )
```

`recombinator` is applied at every generation when an offspring is produced

Operators!


```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         pyEval(r"'%3d   ' % gen", rep=0, step=10),
...         pyEval(r"'%f    ' % LD[0][1]", step=10),
...         pyEval(r"'\n'", rep=REP_LAST, step=10)
...     ],
...     end=100
... )
```

`stat` is applied to the offspring generation at every generation

Operators!

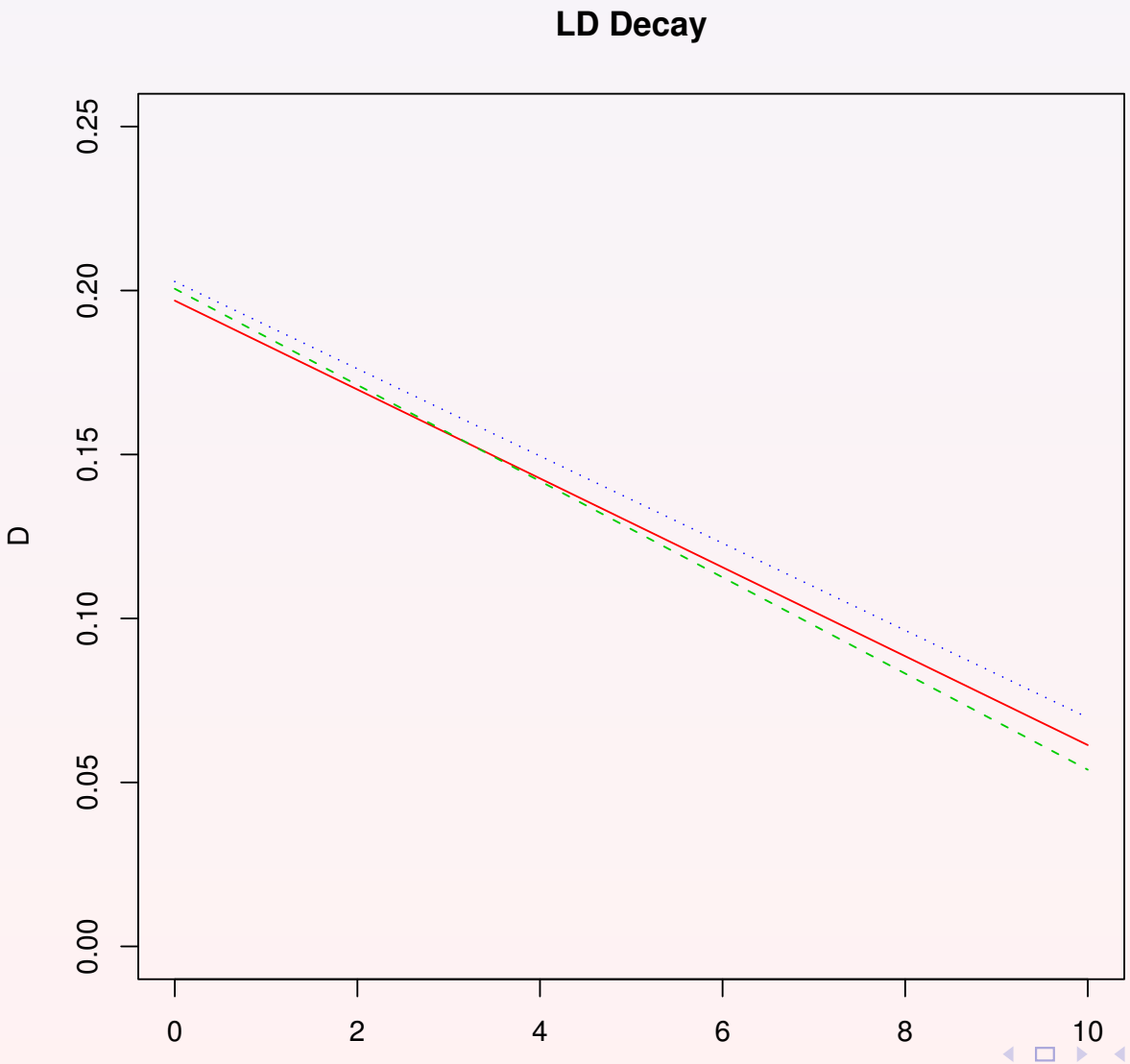

```
>>> from simuPOP import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         pyEval(r"'%3d    ' % gen", rep=0, step=10),
...         pyEval(r"'%f    ' % LD[0][1]", step=10),
...         pyEval(r"'\n'", rep=REP_LAST, step=10)
...     ],
...     end=100
... )
```

`pyEval` is applied every 10 generations

Use R to plot

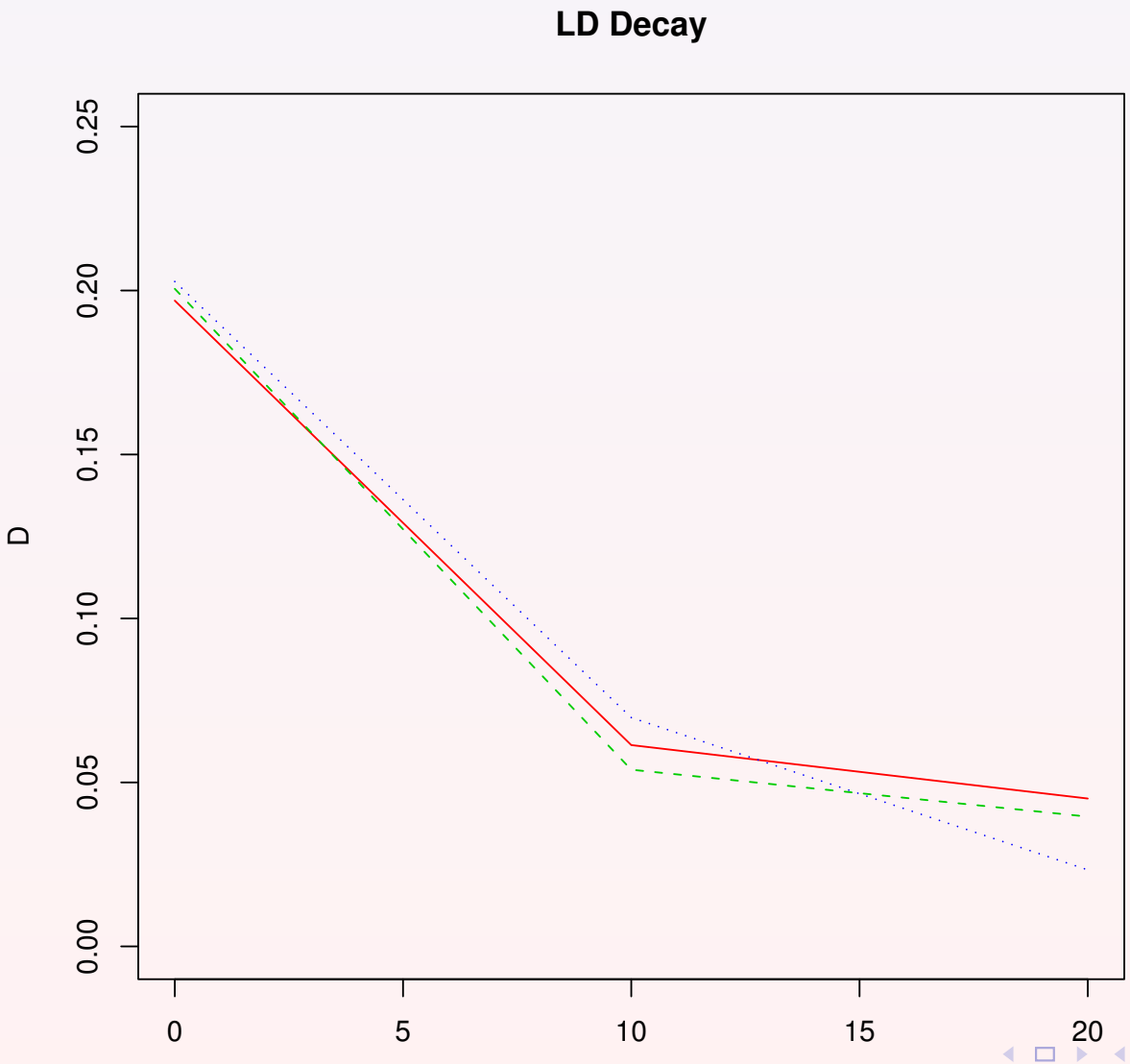

```
>>> from simuPOP import *
>>> from simuRPy import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         varPlotter('LD[0][1]', numRep=3, step=10,
...             saveAs='ld', ylim=[0,.25],
...             lty=range(1, 4), col=range(2, 5),
...             xlab='generation', ylab='D',
...             title='LD Decay'),
...     ],
...     end=100
... )
True
>>>
```

Evolve!



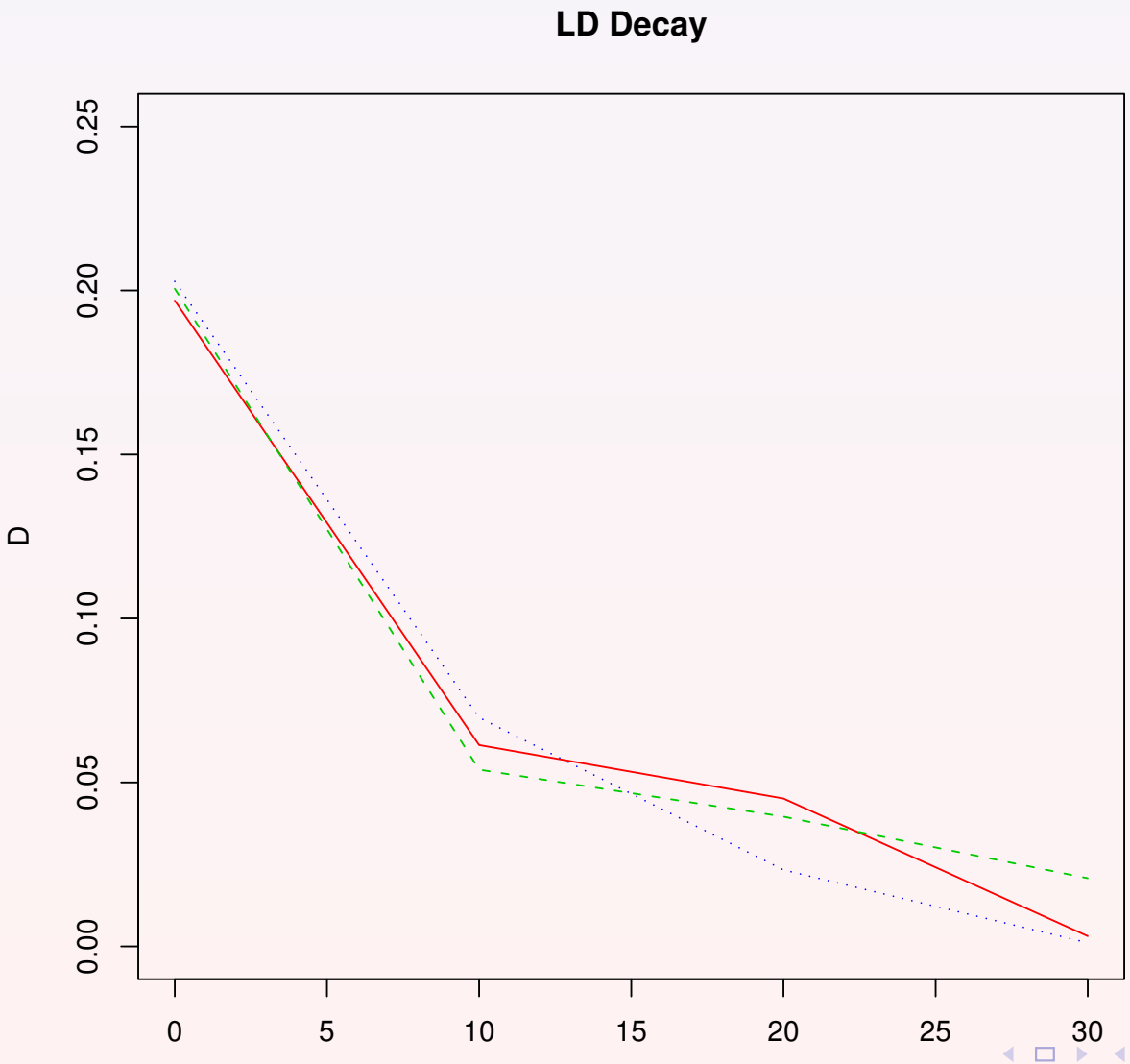
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



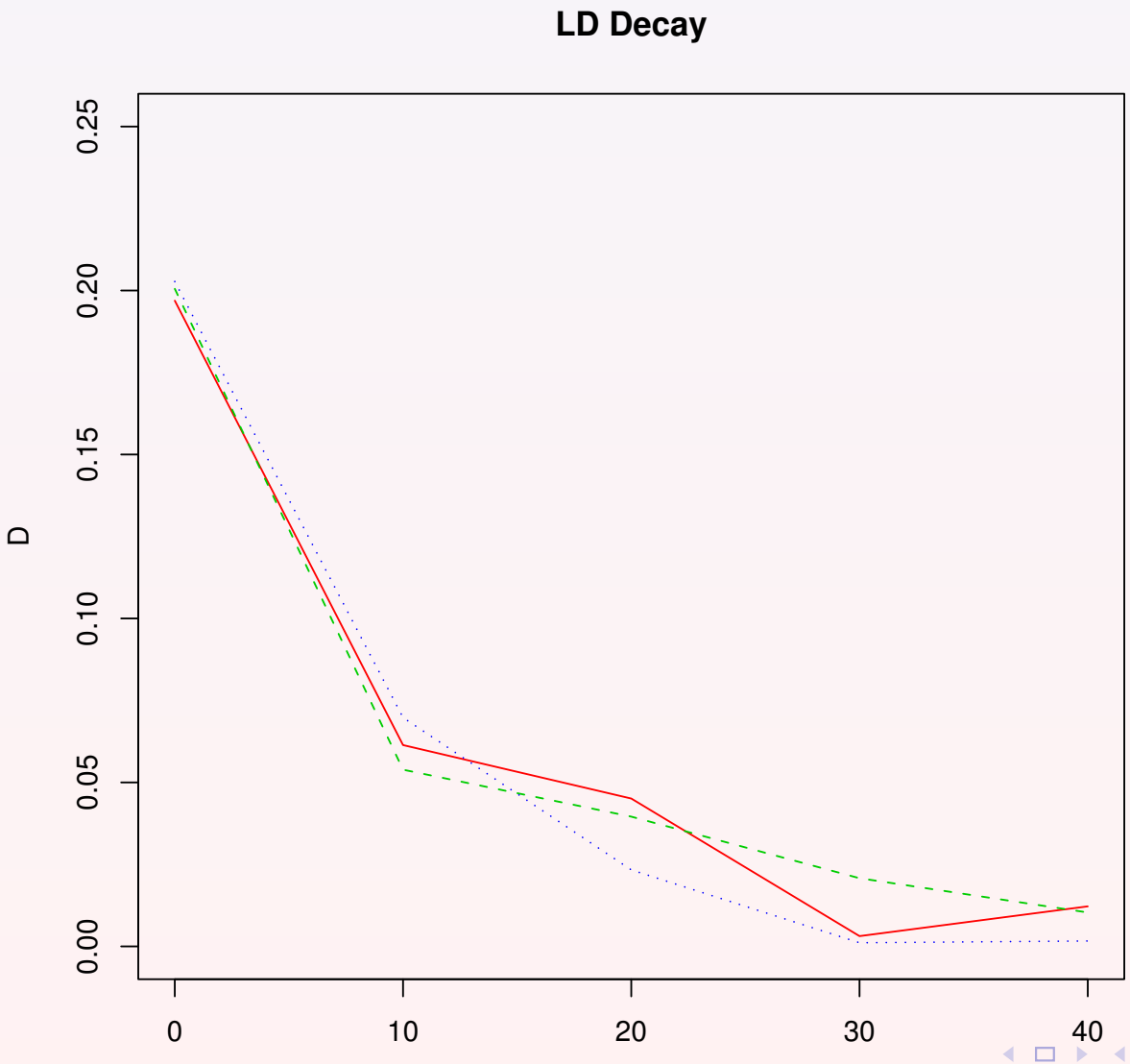
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



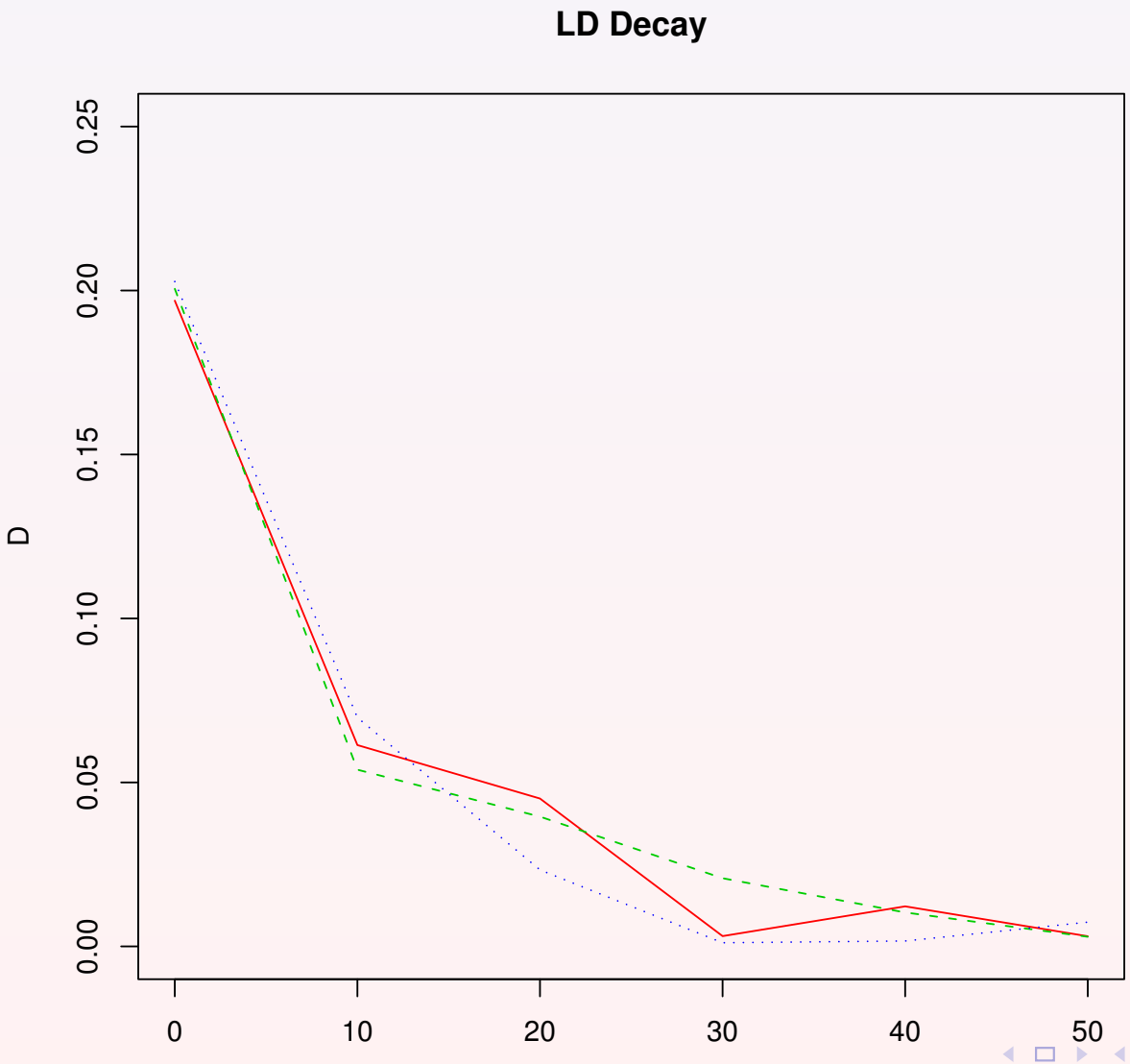
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



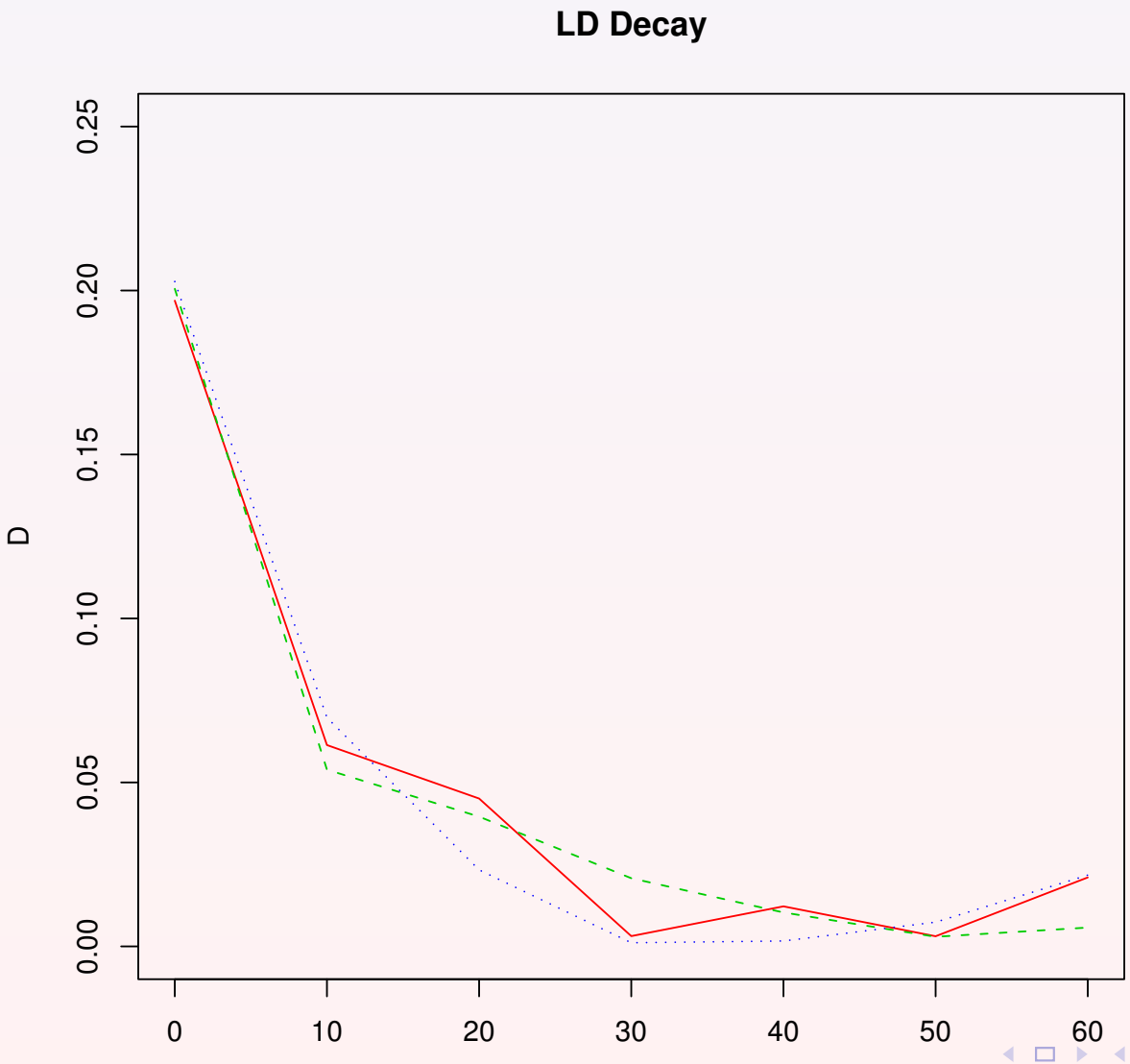
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



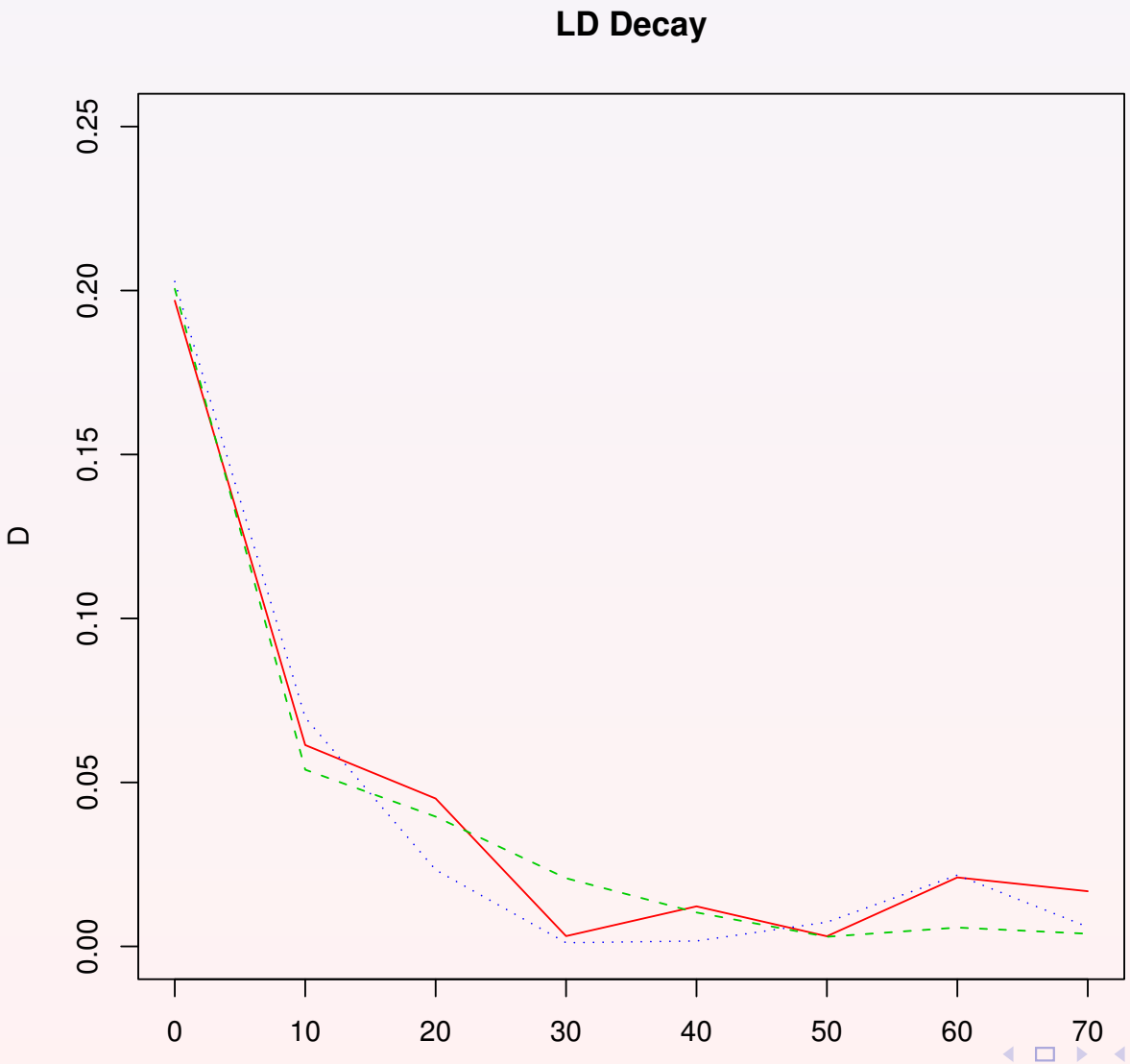
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



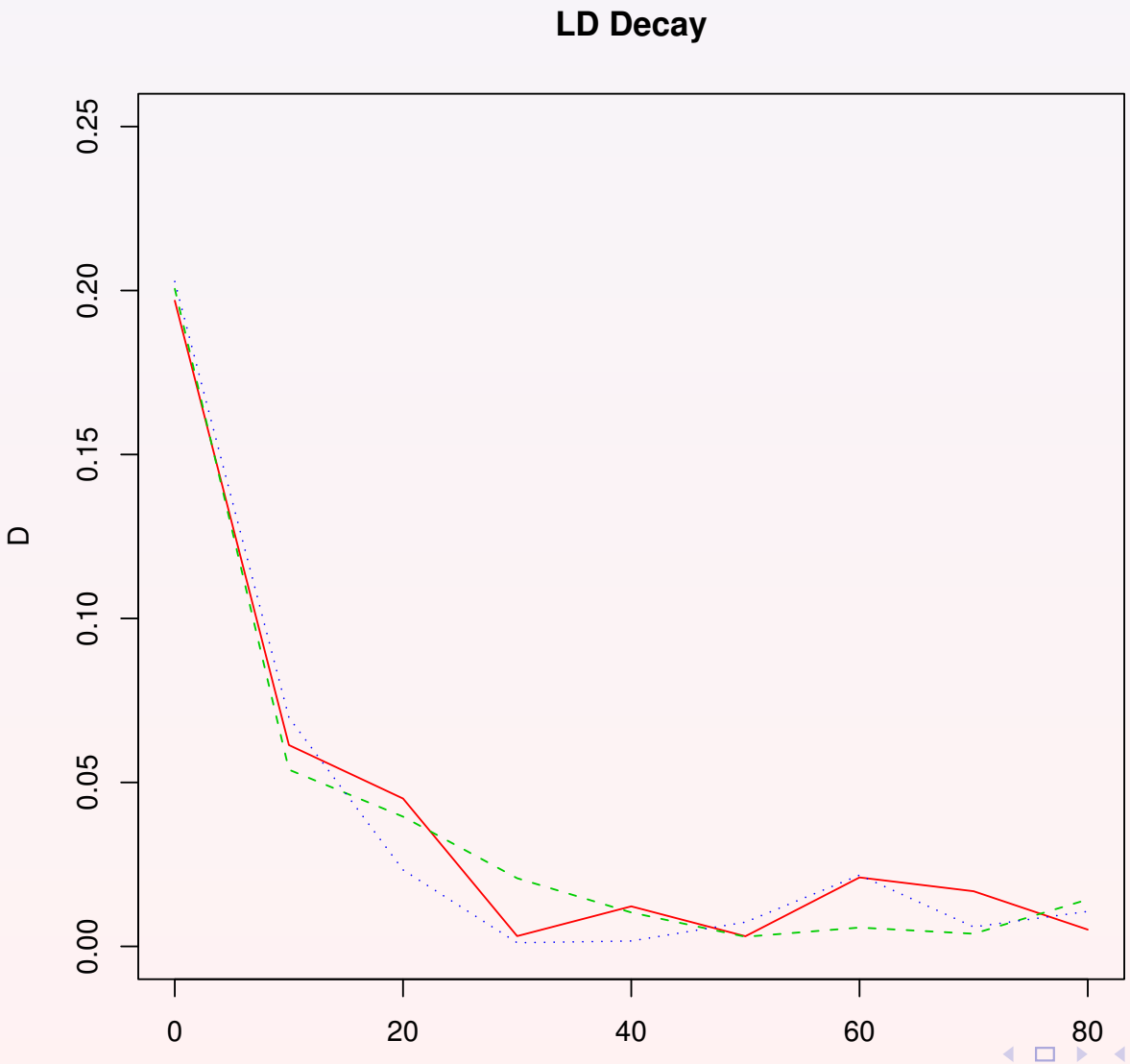
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



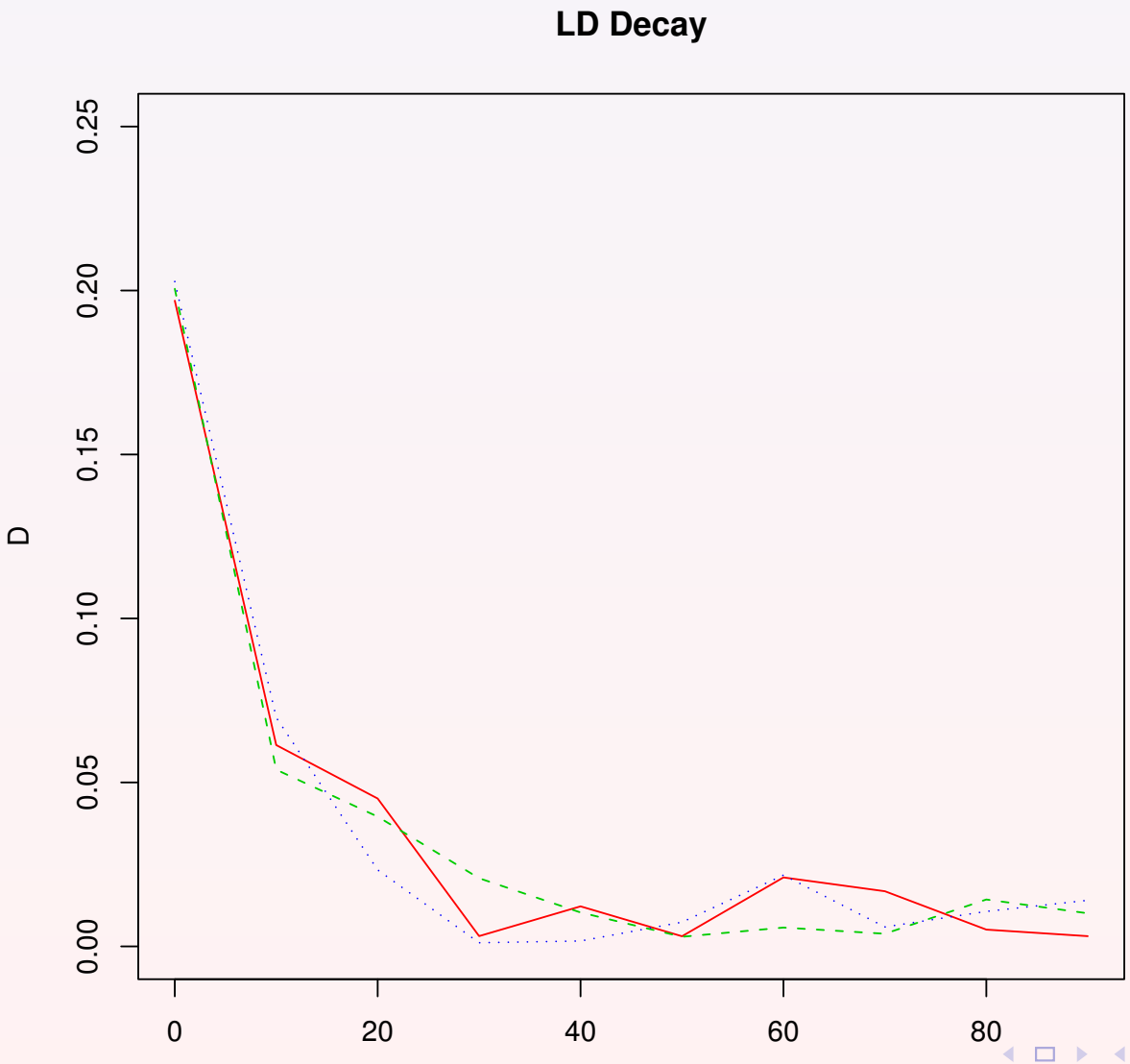
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



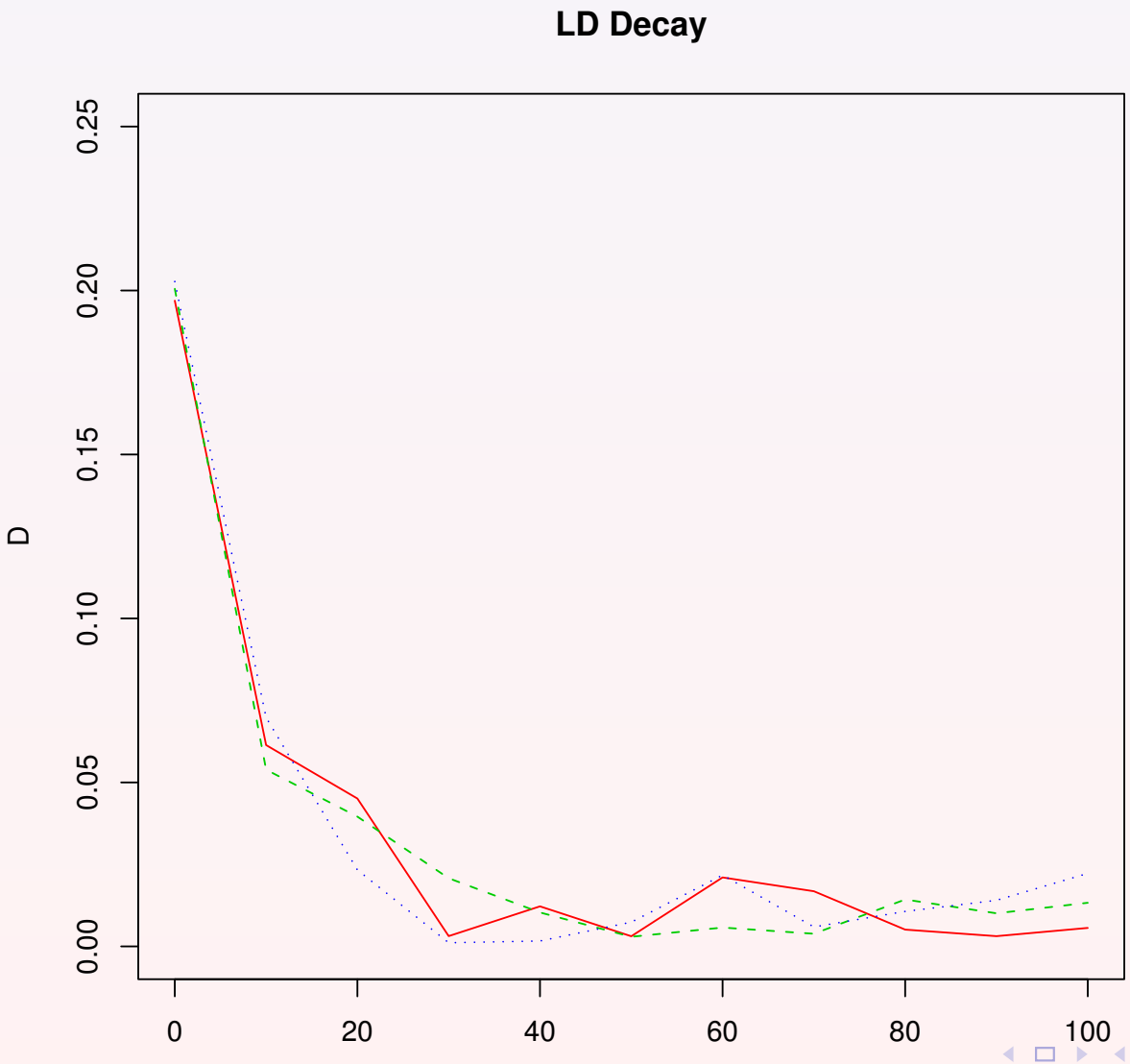
- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Evolve!



- Update at every 10 generations
- LD=0.25 before generation 0
- LD is calculated at the end of each generation

Exercise time

- Start python
- Load simuPOP
- Create a population and run

```
pop.ploidyName()
```

- run `tutorial_example1.py`

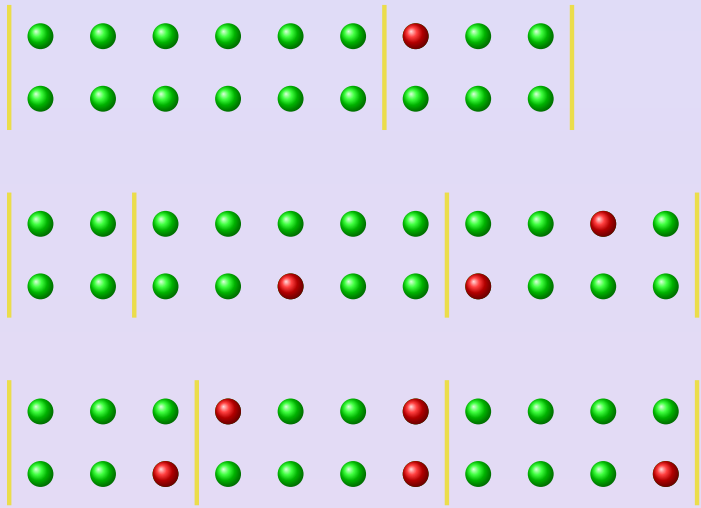
Outline

3 simuPOP components

- Population
- Individual
- Operator
- Mating scheme
- Simulator
- Other utilities

Structure of a population

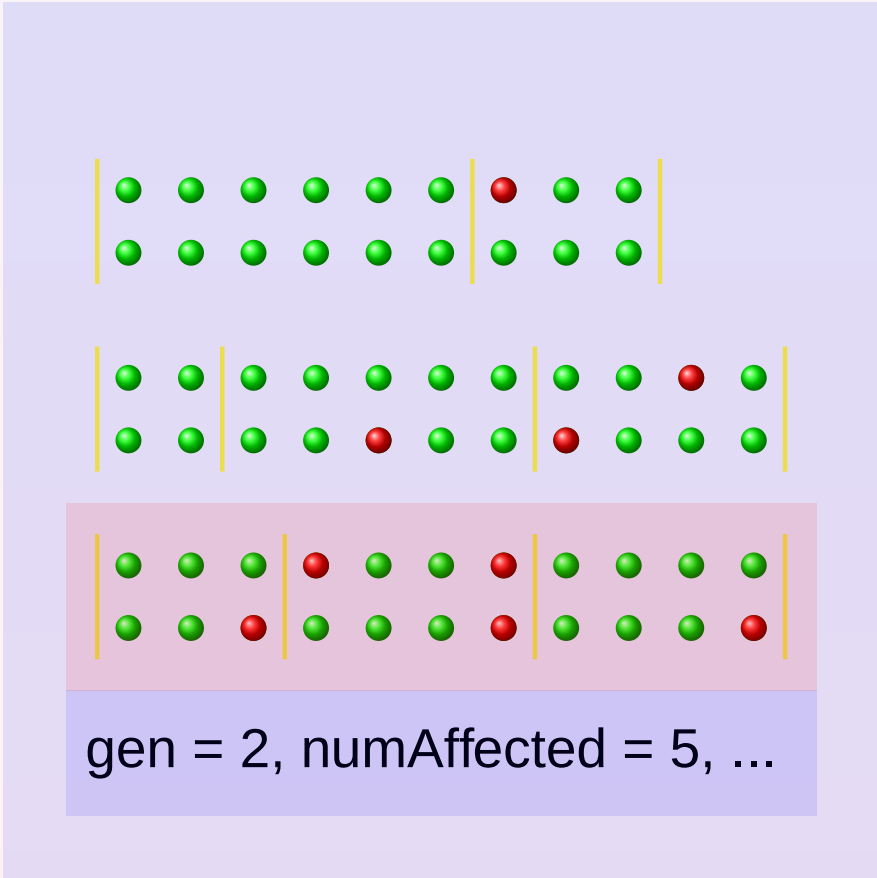
- Unaffected
- Affected



gen = 2, numAffected = 5, ...

Structure of a population

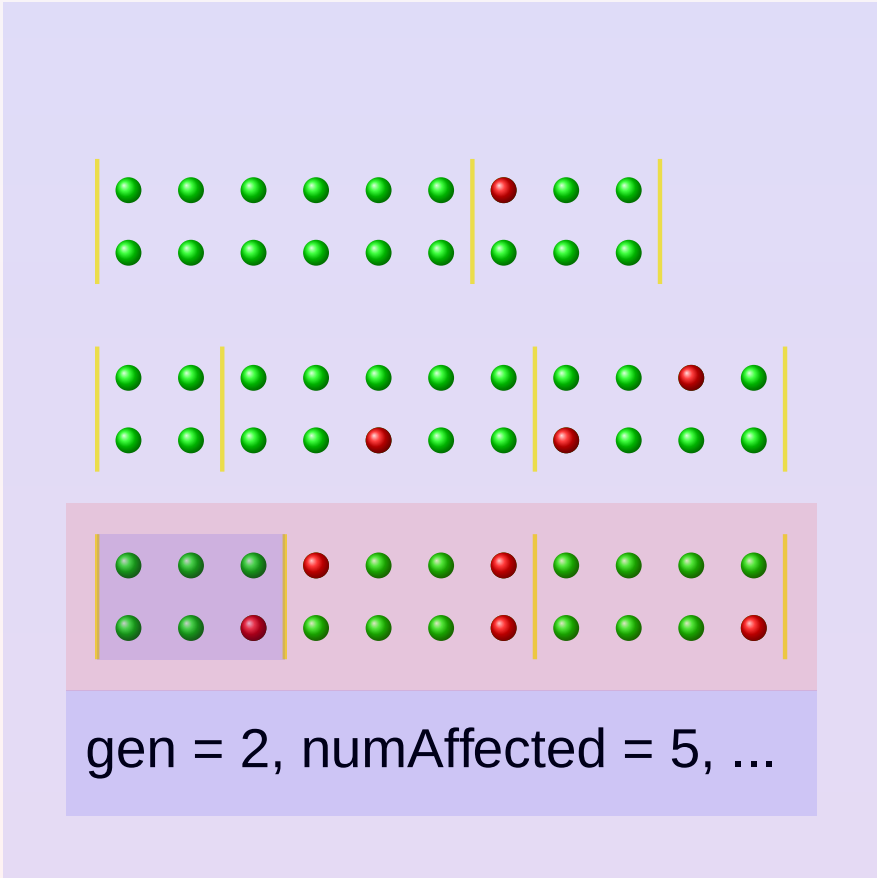
- Unaffected
- Affected



Current generation

Structure of a population

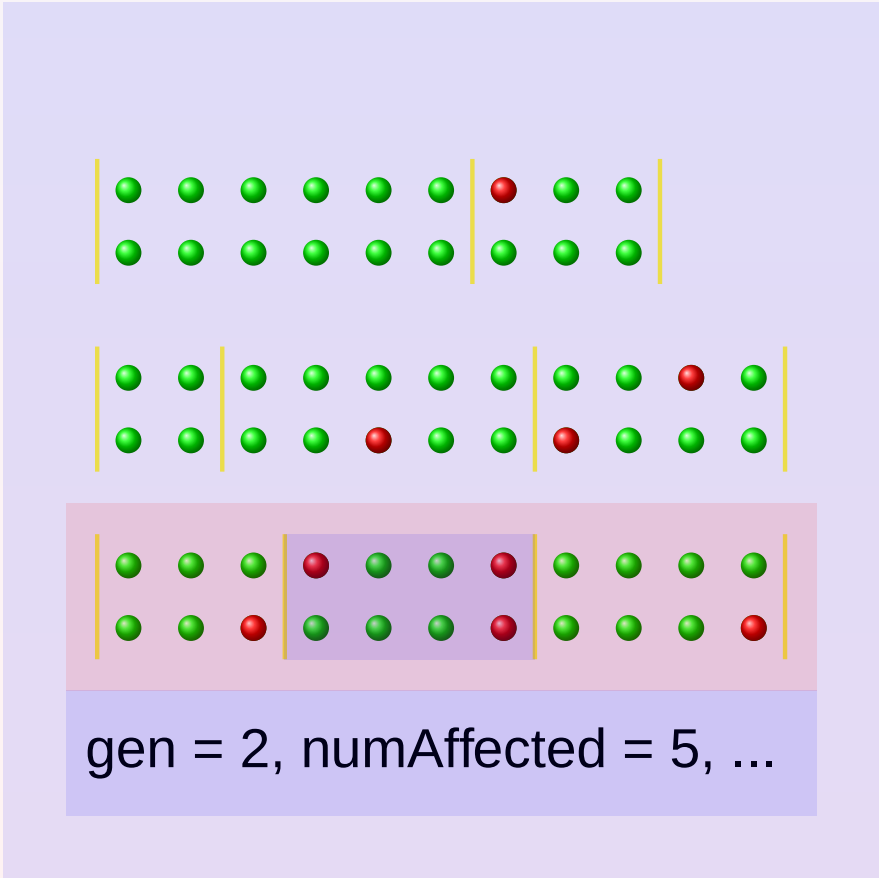
- Unaffected
- Affected



Current generation

Structure of a population

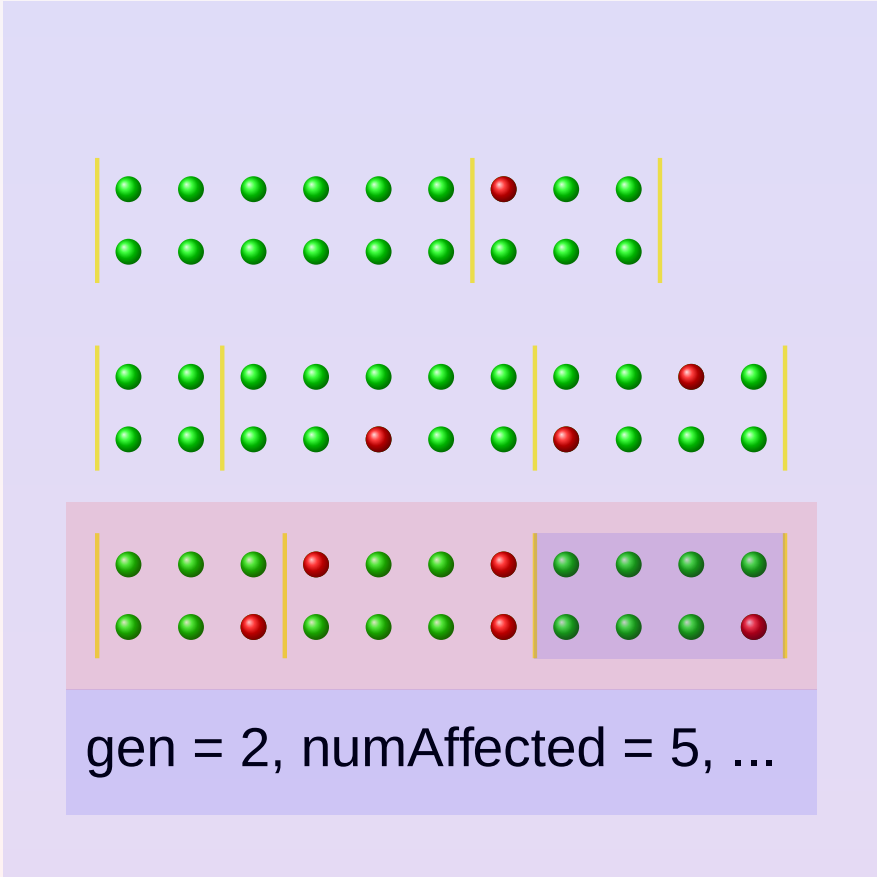
- Unaffected
- Affected



Current generation

Structure of a population

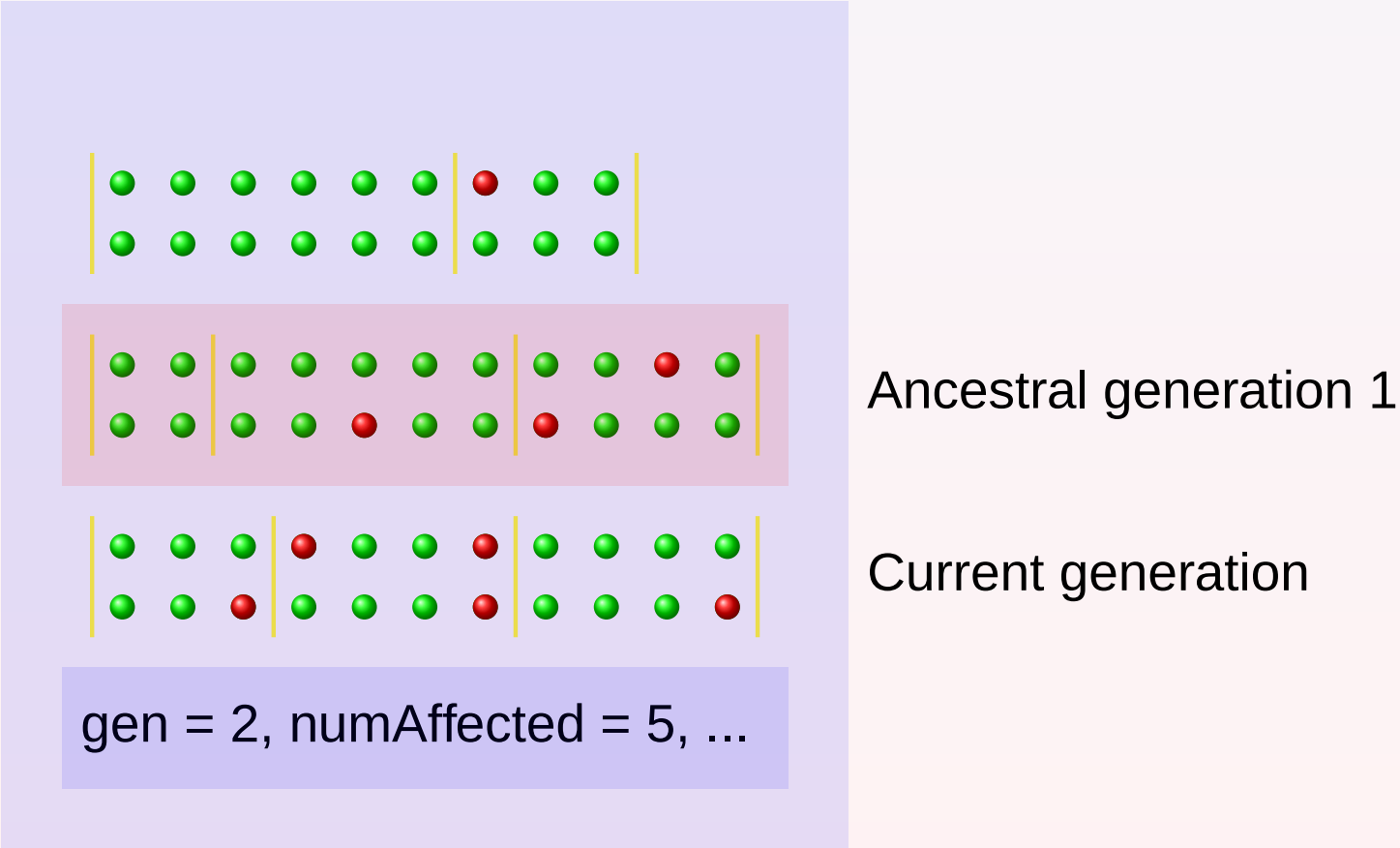
- Unaffected
- Affected



Current generation

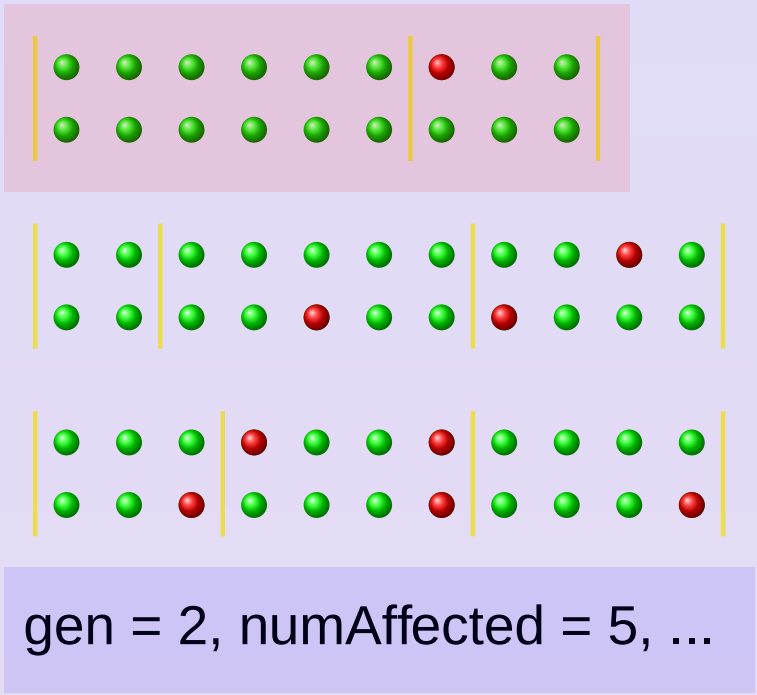
Structure of a population

- Unaffected
- Affected



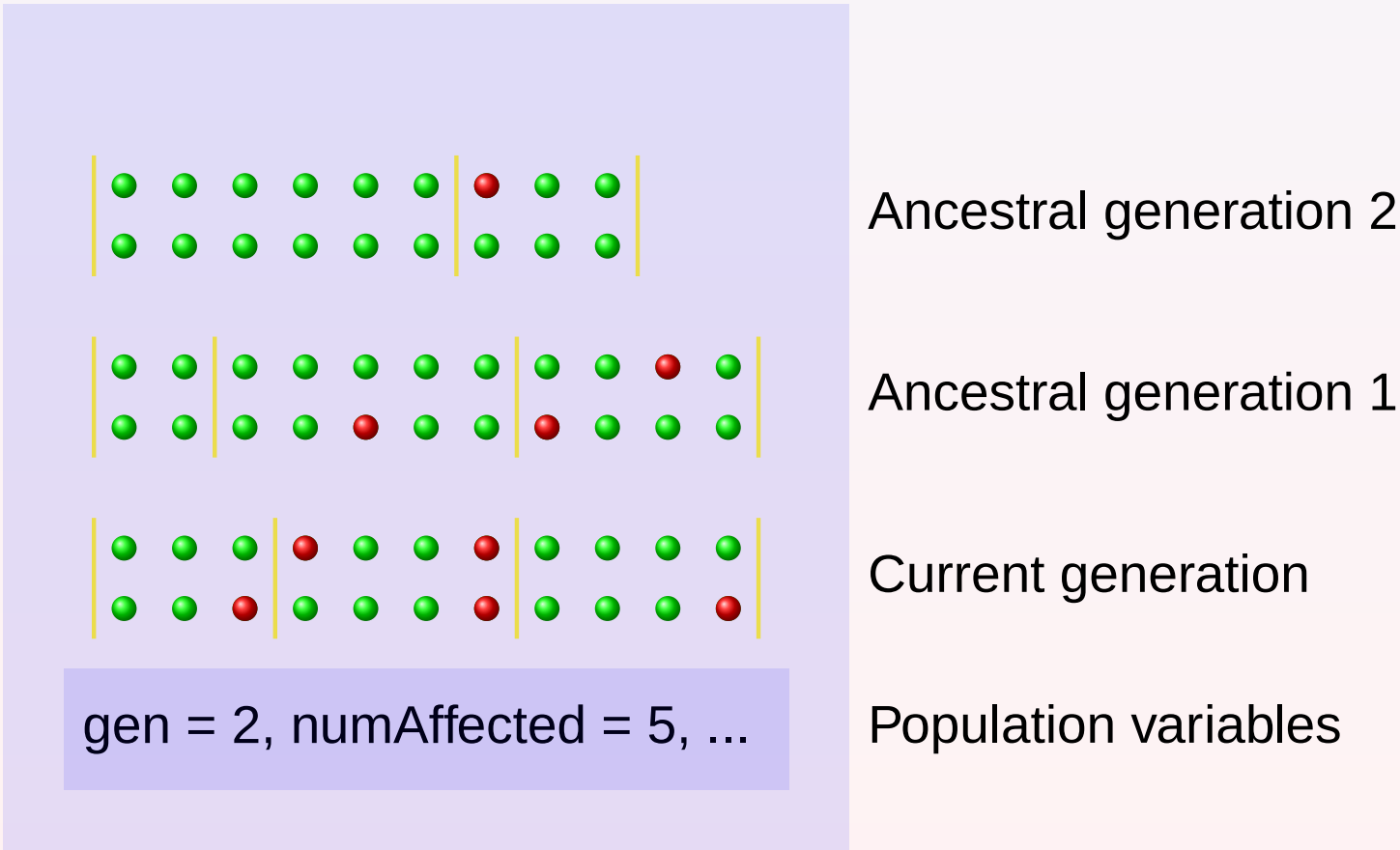
Structure of a population

- Unaffected
- Affected



Structure of a population

- Unaffected
- Affected



Create and manipulate populations


```
>>> pop = population(size=10, loci=[2, 3])
>>> Dump(pop)
Ploidy:                    2
Number of chrom:           2
Number of loci:            2 3
Maximum allele state:      255
Loci positions:
                  1 2
                  1 2 3
Loci names:
                  loc1-1 loc1-2
                  loc2-1 loc2-2 loc2-3
population size:           10
Number of subPop:          1
Subpop sizes:              10
Number of ancestral populations:          0
individual info:
sub population 0:
   0: MU   0  0   0  0  0 |   0  0   0  0  0
   1: MU   0  0   0  0  0 |   0  0   0  0  0
   2: MU   0  0   0  0  0 |   0  0   0  0  0
   3: MU   0  0   0  0  0 |   0  0   0  0  0
```

Genotypic structure


```
>>> pop = population(subPop=[200, 300], loci=[3, 2],
...     maxAllele=3, ploidy=3,
...     lociPos=[[1, 3, 5], [2.5, 4]],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> pop.numLoci(0)
3
>>> pop.totNumLoci()
5
>>> pop.locusPos(4)
4.0
>>> pop.subPopSize(1)
300
>>> pop.popSize()
500
>>> pop.ploidyName()
'triploid'
>>> pop.individual(1).allele(1, 2)
0
>>>
```

Population manipulation


```
>>> # make a copy of pop
>>> pop1 = pop.clone()
>>> # remove loci 2, 3, 4
>>> pop.removeLoci(keep=[0, 1])
>>> # pop2 will have 3 chromosomes, with loci 2, 3, 2
>>> pop2 = MergePopulationsByLoci(pops=[pop, pop1])
>>> # randomly assign alleles using given allele frequencies
>>> InitByFreq(pop2, [0.8, .2])
>>> # calculate population allele frequency
>>> Stat(pop2, alleleFreq=range(pop2.totNumLoci()))
>>> # print allele frequency
>>> print pop2.dvars().alleleFreq
[[0.7946666666666663, 0.2053333333333334], [0.8033333333333333
>>> # assign affection status using a penetrance model
>>> MapPenetrance(pop2, locus=1,
...     penetrance={'0-0': 0.05, '0-1': 0.2, '1-1': 0.8})
>>> # draw case control sample
>>> (sample,) = CaseControlSample(pop2, cases=5, controls=5)
>>> # save sample in Merlin QTDT format
>>> from simuUtil import SaveQTDT
>>> SaveQTDT(sample, output='sample', affectionCode=['U', 'A'],
...     fields=['affection'])
```

Population manipulation (cont.)


```
>>> # have a look at the sample in Merlin-QTDT Format
>>> print open('sample.map').read()
CHROMOSOME MARKER POSITION
1        loc1-1  1.000000
1        loc1-2  3.000000
2        loc1-1_1        1.000000
2        loc1-2_1        3.000000
2        loc1-3  5.000000
3        loc2-1  2.500000
3        loc2-2  4.000000

>>> print open('sample.dat').read()
A        affection
M        loc1-1
M        loc1-2
M        loc1-1_1
M        loc1-2_1
M        loc1-3
M        loc2-1
M        loc2-2
```

Population manipulation (cont.)


```
>>> print open('sample.ped').read()
1 1 0 0 2 A 2 1 2 1 1 1 2 1 1 1 1 1 1 1
2 1 0 0 2 A 1 1 1 1 1 2 1 1 1 2 1 1 1 1
3 1 0 0 2 A 2 1 2 1 1 1 1 1 1 1 1 1 1 1
4 1 0 0 1 A 2 2 2 2 1 2 1 1 1 2 2 2 1 2
5 1 0 0 1 A 1 1 2 1 1 1 1 1 1 1 2 1 2 1
6 1 0 0 2 U 1 1 1 2 1 2 1 2 2 1 1 1 1 1
7 1 0 0 1 U 2 1 1 1 1 2 1 1 1 1 1 1 1 1
8 1 0 0 2 U 1 1 1 1 1 1 1 1 1 2 1 1 1 1
9 1 0 0 1 U 1 1 2 1 2 1 2 1 1 2 1 1 1 1
10 1 0 0 2 U 1 1 1 1 1 1 1 1 1 2 1 1 2 2

>>>
```

Population variables


```
>>> pop = population(subPop=[5, 10], loci=[5])
>>> InitByFreq(pop, [.6, .3, .1])
>>> Stat(pop, alleleFreq=[1], genoFreq=[2])
>>> print pop.dvars().alleleFreq[1][0]
0.533333333333
>>> from simuUtil import ListVars
>>> ListVars(pop.dvars(), useWxPython=False)
 grp :  -1
 rep :  -1
 alleleNum :
   [1]
     [0]        16
     [1]        12
     [2]        2
 genoFreq :
   [2]
     [0]
       0 :      0.266666666667
       1 :      0.533333333333
       2 :      0.0666666666667
     [1]
       1 :      0.0666666666667
       2 :      0.0666666666667
 genoNum :
   [2]
     [0]
       0 :      4.0
       1 :      8.0
       2 :      1.0
     [1]
       1 :      1.0
```

Population variables (cont.)


```
    [1]         0.4
    [2]         0.0666666666667
subPop
  [0]
    alleleNum :
      [1]
        [0]     5
        [1]     5
    genoNum :
      [2]
        [0]
          0 :   3.0
          1 :   1.0
        [1]
          2 :   1.0
    genoFreq :
      [2]
        [0]
          0 :   0.6
          1 :   0.2
        [1]
          2 :   0.2
    alleleFreq :
      [1]
        [0]     0.5
        [1]     0.5
```

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2		3	4			5	6
0	1	1		1	0			0	1
0	0	1		1	1			0	1

0			1	2	3	4	5
0			1	0	0	0	1
1			0	1	1	0	0

Male

- Affected

fitness	father_id	···

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2		3	4			5	6
0	1	1		1	0			0	1
0	0	1		1	1			0	1

0			1	2	3	4	5
0			1	0	0	0	1
1			0	1	1	0	0

Chromosome 0

Male

- Affected

fitness	father_id	···

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2		3	4			5	6
0	1	1		1	0			0	1
0	0	1		1	1			0	1

0			1	2	3	4	5
0			1	0	0	0	1
1			0	1	1	0	0

Chromosome 0

Chromosome 1

Male

- Affected

fitness	father_id	...

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2		3	4			5	6
0	1	1		1	0			0	1
0	0	1		1	1			0	1

0			1	2	3	4	5
0			1	0	0	0	1
1			0	1	1	0	0

Chromosome 0

Chromosome 1

Male

Sex

- Affected

fitness	father_id	···

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2		3	4			5	6
0	1	1		1	0			0	1
0	0	1		1	1			0	1

0			1	2	3	4	5
0			1	0	0	0	1
1			0	1	1	0	0

Chromosome 0

Chromosome 1

Male

Sex

- Affected

Affection status

fitness	father_id	···

Structure of Individuals

Assume ploidy = 2, maxAllele = 1

0	1	2	3	4	5	6
0	1	1	1	0	0	1
0	0	1	1	1	0	1

Chromosome 0

0	1	2	3	4	5
0	1	0	0	0	1
1	0	1	1	0	0

Chromosome 1

Male

Sex

- Affected

Affection status

fitness	father_id	...

Information fields

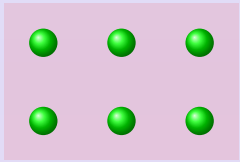
Individuals


```
>>> pop = population(subPop=[5, 8], loci=[5],
...     infoFields=['penetrance'])
>>> InitByFreq(pop, [.6, .3, .1])
>>> MaPenetrance(pop, locus=2, penetrance=[0.05, 0.2, 0.5],
...     wildtype=[0], infoFields=['penetrance'])
>>> # iterate through all inviduals in subPop 1
>>> for ind in pop.individuals(1):
...     print 'Aff: %d Fit: %.3f Geno: %d %d' % \
...         (ind.affected(), ind.info('penetrance'), \
...         ind.allele(2, 0), ind.allele(2, 1))
...
Aff: 0 Fit: 0.050 Geno: 0 0
Aff: 0 Fit: 0.200 Geno: 1 0
Aff: 0 Fit: 0.050 Geno: 0 0
Aff: 0 Fit: 0.050 Geno: 0 0
Aff: 0 Fit: 0.200 Geno: 1 0
Aff: 1 Fit: 0.200 Geno: 0 2
Aff: 0 Fit: 0.200 Geno: 0 2
Aff: 0 Fit: 0.050 Geno: 0 0
>>>
```

Information fields


```
>>> pop = population(100, loci=[5, 8],
...     infoFields=['father_idx', 'mother_idx'])
>>> simu = simulator(pop, randomMating(numOffspring=2))
>>> simu.evolve(ops=[parentsTagger()], end=5)
True
>>> ind = simu.population(0).individual(0)
>>> ind1 = simu.population(0).individual(1)
>>> print ind.info('father_idx'), ind.info('mother_idx')
89.0 0.0
>>> print ind1.info('father_idx'), ind1.info('mother_idx')
89.0 0.0
>>>
>>>
```

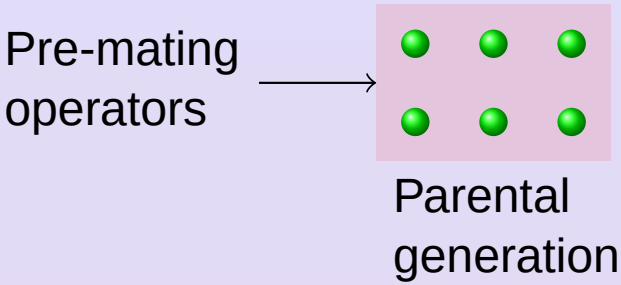
Life cycle of a generation



Parental generation

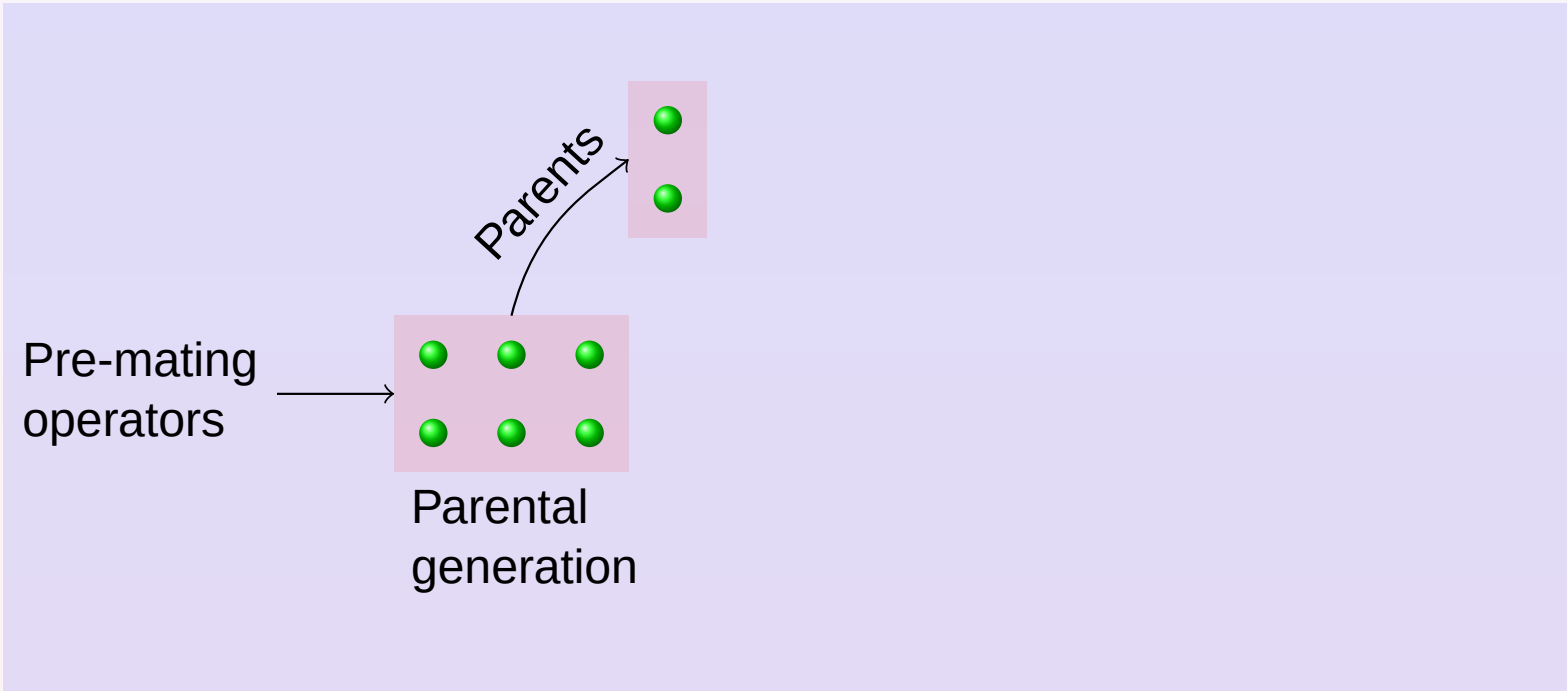
Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



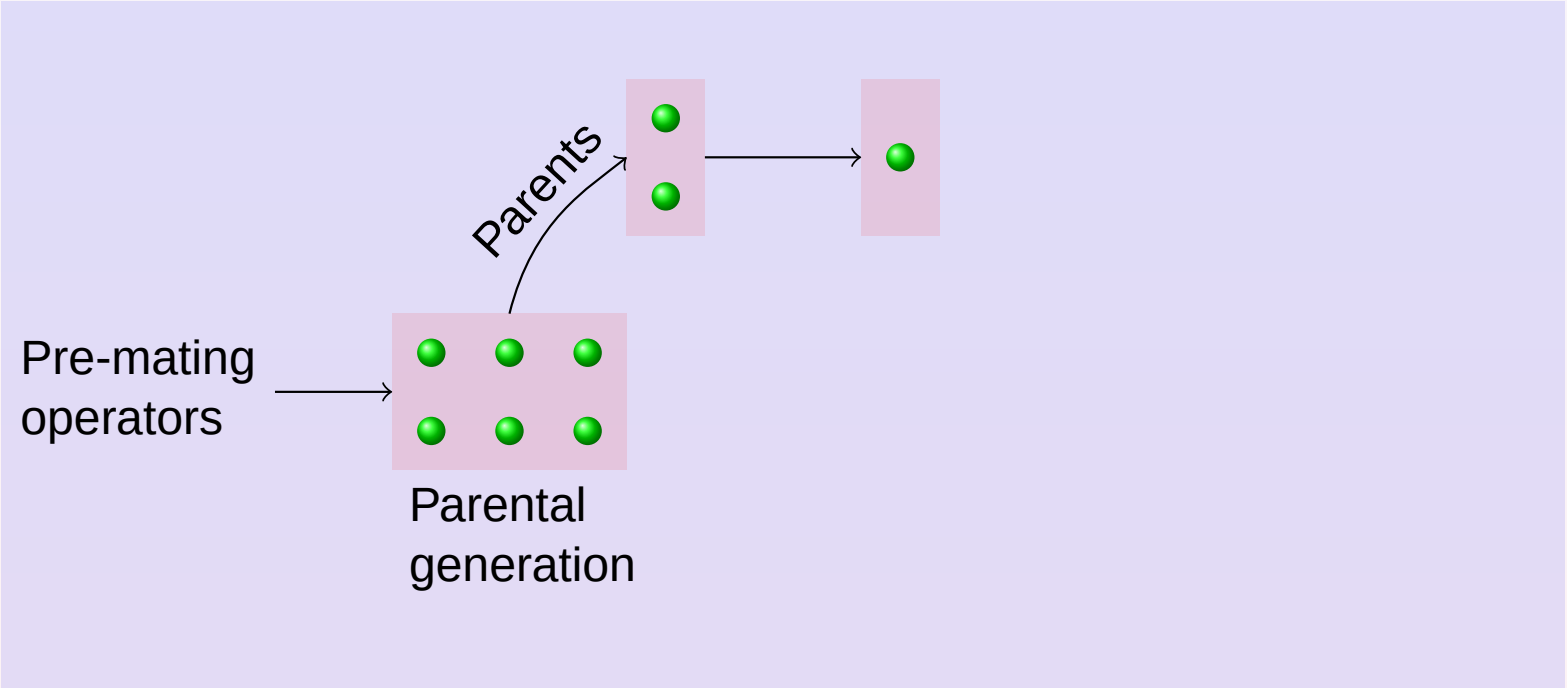
Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



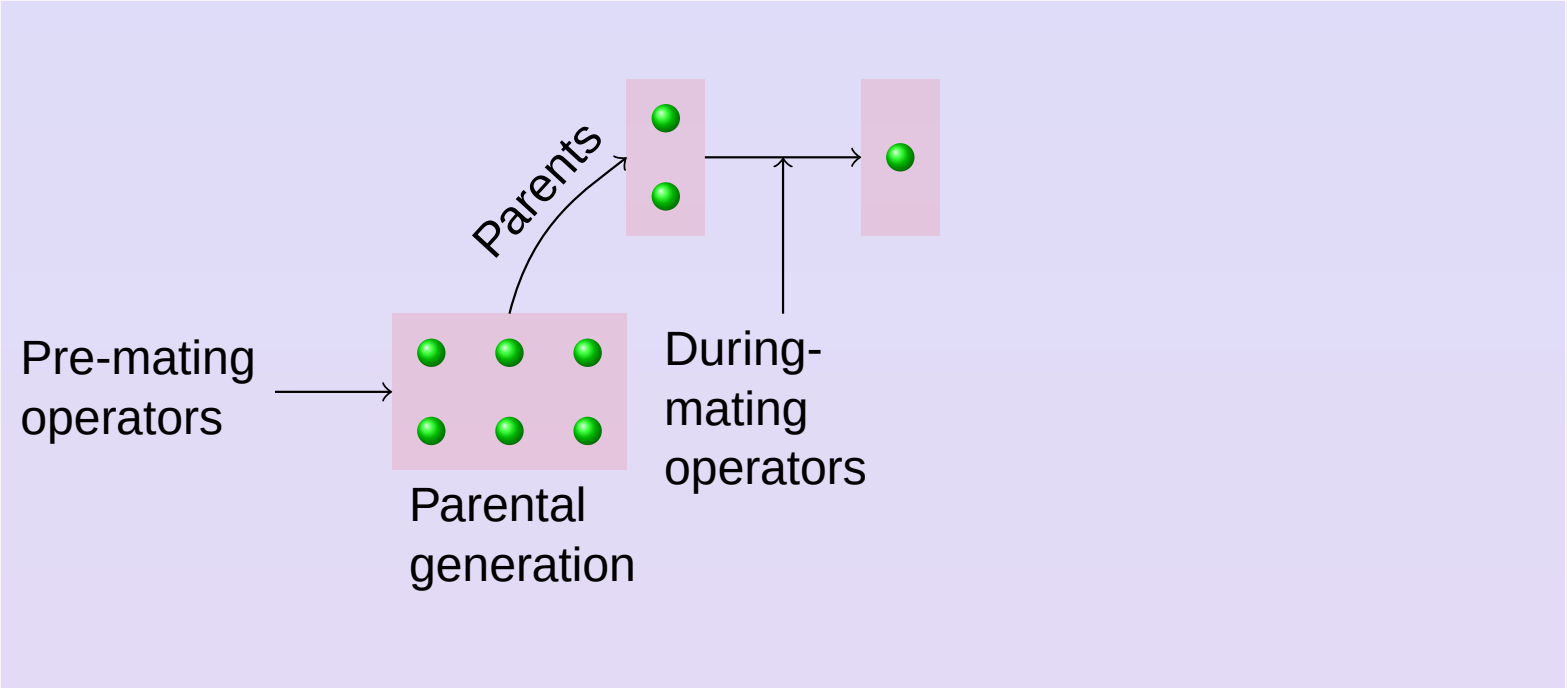
Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



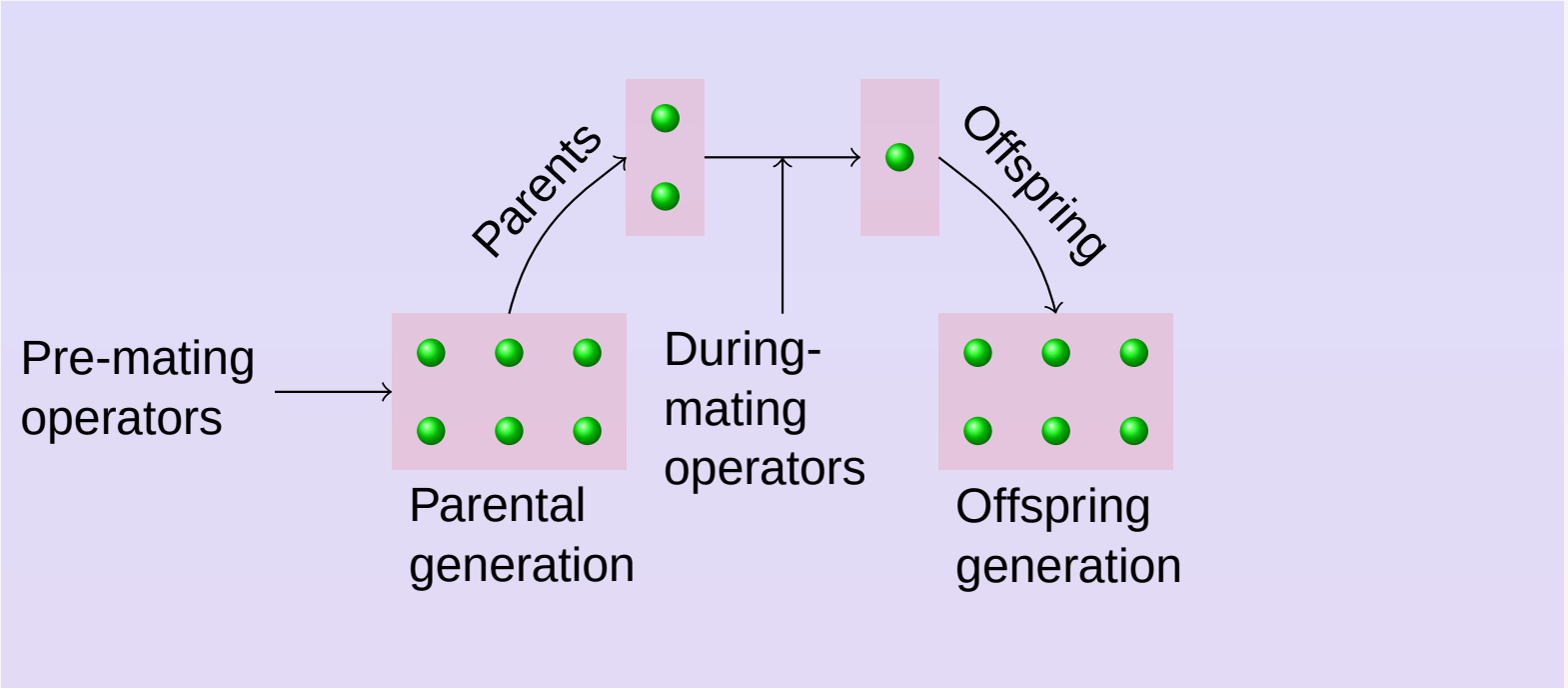
Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



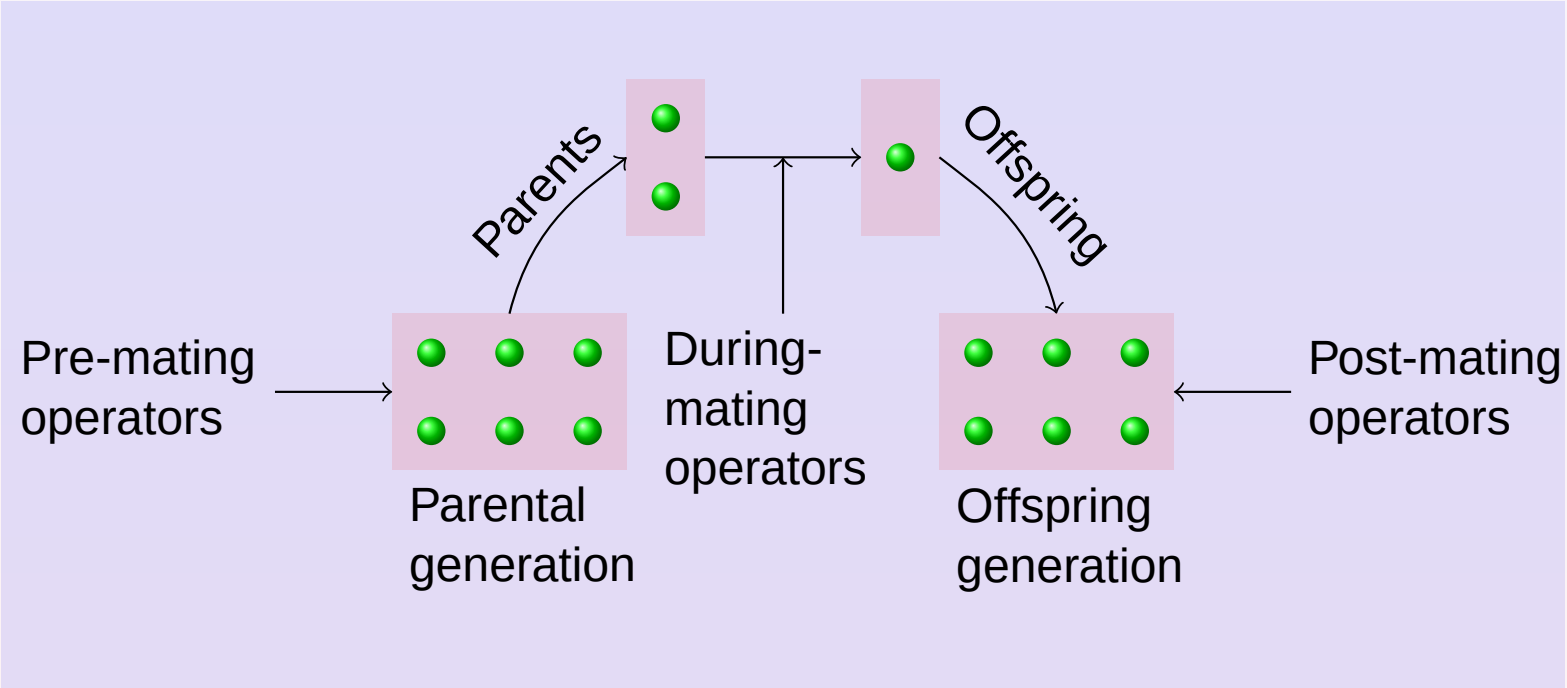
Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



Every operator has a default stage, and a `stage` parameter to change it.

Life cycle of a generation



Every operator has a default stage, and a `stage` parameter to change it.

Pre-, During- and PostMating operators


```
>>> simu = simulator(
...     population(subPop=[20, 80], loci=[3]),
...     randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         kamMutator(maxAllele=10, rate=0.00005, atLoci=[0,2]),
...         recombinator(rate=0.001),
...         dumper(stage=PrePostMating),
...         stat(alleleFreq=[1]),
...     ],
...     dryrun=True
... )
Dryrun mode: display calling sequence
Apply pre-evolution operators
  Replicate 0
      - <simuPOP::initByFreq> end at 1
Start evolution
  Replicate 0
    Pre-mating operators
      - <simuPOP::dumper> at all generations
    Start mating
      - <simuPOP::recombination> at all generations
    Apply post-mating operators
      - <simuPOP::k-allele model mutator K=10> at all generations
      - <simuPOP::dumper> at all generations
      - <simuPOP::statistics> at all generations
True
>>>
```

Applicable generations


```
>>> simu = simulator(
...     population(10000, loci=[3]),
...     randomMating())
>>> eval1 = r"'Gen: %3d  Freq: %f\n' % (gen, alleleFreq[1][0])"
>>> eval2 = r"'Last Gen: %3d  Freq: %s\n' % (gen, alleleFreq[1])"
>>> simu.evolve(
...     preOps = [initByFreq([0.3, 0.7])],
...     ops = [
...         recombinator(rate=0.01, begin=10, end=30),
...         stat(alleleFreq=[1], step=10),
...         pyEval(eval1, step=10),
...         pyEval(eval2, at=[-1])
...     ],
...     end = 50
... )
Gen:   0  Freq: 0.297000
Gen:  10  Freq: 0.303700
Gen:  20  Freq: 0.322550
Gen:  30  Freq: 0.317650
Gen:  40  Freq: 0.313800
Gen:  50  Freq: 0.319350
Last Gen:  50  Freq: [0.31935000000000002, 0.68064999999999998]
True
>>>
```

Applicable replicates


```
>>> simu = simulator(
...     population(100, loci=[3]),
...     randomMating(),
...     rep=5, grp=[1,1,2,2,2])
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         stat(alleleFreq=[1]),
...         recombinator(rate=0.01, grp=1),
...         recombinator(rate=0.01, grp=2),
...         pyEval(r"'%.2f ' % alleleFreq[1][0]", grp=1),
...         pyEval(r"'\n'", rep=REP_LAST),
...     ],
...     end=5
... )
0.45 0.40
0.45 0.47
0.42 0.49
0.41 0.44
0.34 0.48
0.35 0.45
True
>>>
```

Output


```
>>> simu = simulator(
...     population(100, loci=[3]),
...     randomMating(),
...     rep=5, grp=[1,1,2,2,2])
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         stat(alleleFreq=[1]),
...         pyEval(r"'%.2f ' % alleleFreq[1][0]",
...             output='>>out'),
...         pyEval(r"'\n'", rep=REP_LAST, output='>>out'),
...         pyEval(r"'%.2f ' % alleleFreq[1][0]",
...             outputExpr="'>>out%d' % grp"),
...     ],
...     end=2
... )
True
>>> print open('out').read()
0.44 0.53 0.40 0.47 0.49
0.49 0.52 0.39 0.48 0.45
0.48 0.49 0.38 0.53 0.44

>>> print open('out1').read()
0.44 0.53 0.49 0.52 0.48 0.49
>>> print open('out2').read()
0.40 0.47 0.49 0.39 0.48 0.45 0.38 0.53 0.44
>>>
```

Mating schemes

Mating schemes

- Population offspring subpopulation from corresponding parental subpopulation
- Can change subpopulation size
- Select parents according to their `fitness` value (information field)
- Can produce more than one offspring

Demographic model


```
>>> def lin_inc(gen, oldsize=[]):
...     return [10+gen]*5
...
>>> simu = simulator(
...     population(subPop=lin_inc(1), loci=[1]),
...     randomMating(newSubPopSizeFunc=lin_inc)
... )
>>> simu.evolve(
...     ops = [
...         stat(popSize=True),
...         pyEval(r'"%d %d\n"%(gen, subPop[0]["popSize"])'),
...     ],
...     end=5
... )
0 10
1 11
2 12
3 13
4 14
5 15
True
>>>
```

Number of offspring


```
>>> simu = simulator(
...     population(size=10000, loci=[1]),
...     randomMating(),
... )
>>> simu.evolve(
...     preOps = [initByFreq([0.1, 0.9])],
...     ops = [ ], end=100
... )
True
>>> simu.setMatingScheme(randomMating(numOffspring=2))
>>> simu.addInfoFields(['father_idx', 'mother_idx'])
>>> simu.setAncestralDepth(1)
>>> simu.step(ops=[parentsTagger()])
True
>>> pop = simu.getPopulation(0)
>>> MaPenetrance(pop, locus=0, penetrance=[0.05, 0.1, 0.5])
>>> AffectedSibpairSample(pop, size=100)
[<simuPOP::population of size 200>]
>>>
```

Simulator

A simulator manages

- Replicates of a population
- A mating scheme
- Many operators

and evolve the populations.

Utility modules and scripts

simuOpt.py provides an easy way to handle parameters.

simuUtil.py provides functions to save/load in many formats, gene mapping functions, list variables etc

simuCluster.py a control script to send jobs to cluster systems

simuLDDecay.py a simple script to demonstrate the decay of linkage disequilibrium under recombination

simuForward.py implements a traditional forward-time simulation scenario

simuComplexDisease.py implements a new forward-time simulation method (PLoS Genetics, 2007)

simuCDCV.py demonstrate the evolution of allelic spectrum

...

Outline

4 A real-world application

scripts/loadHapMap.py

Using optimized binary version of simuPOP

```
from simuOpt import setOptions
setOptions(optimized=True, alleleType='binary')
from simuPOP import *
```

scripts/loadHapMap.py

Load genotype from hapmap data file

```
def load_population(pop, ch, type):
    '''Load population from file, with type (subpopulation type)'''
    subPop = {'CEU':0, 'YRI':1, 'JPT+CHB':2}[type]
    file = genotype_file % (ch, type, rev)
    print 'from %s...' % file
    for line_no,line in enumerate(open(file).readlines()):
        genotype = [int(x) for x in line.split()]
        ind = line_no / 2
        ploidy = line_no % 2
        ind = pop.individual(ind, subPop)
        for i,g in enumerate(genotype):
            # always chromosome 0, because each population has only one chromosome
            ind.setAllele(g, i, ploidy)
```

scripts/loadHapMap.py

Save HapMap chromosome files in simuPOP format

```
if __name__ == '__main__':
    ps = [0,0,0]
    for ch in range(1, 23):
        popFile = "hapmap_%d.bin" % ch
        (lociPos, lociName) = getLoci(ch)
        popSize = getPopSize(len(lociPos), ch)
        if ps[0] == 0:
            ps = popSize
        else:
            if ps[0] != popSize[0] or ps[1] != popSize[1] or ps[2] != popSize[2]:
                print "Population size does not match across chromosomes"
                sys.exit(1)
        pop = population(subPop=popSize, ploidy=2, loci=[len(lociPos)],
            lociPos=lociPos, lociNames=lociName)
        load_population(pop, ch, type='CEU')
        load_population(pop, ch, type='YRI')
        load_population(pop, ch, type='JPT+CHB')
        Stat(pop, alleleFreq=range(pop.totNumLoci()))
        SavePopulation(pop, popFile)
```

Pick markers from HapMap data


```
>>> genes = [
...     ("p53exon4", "rs1042522"),
...     ("p53_6", "rs1625895"),
...     ("xpdex23", "rs1799793"),
...     ("xpdex10", "rs13181"),
...     ("xpa", "rs1800975"),
...     ("xpg1104", "rs17655"),
...     ("xpf662", "rs2020955"),
...     ("ercc61097", "rs2228526"),
...     ("ercc61230", "rs4253211"),
...     ("xpc_939", "rs2228001"),
...     ("ccnh", "rs2266690"),
...     ("rad23", "rs1805329"),
...     ("ercc1", "rs3212986"),
...     ("xpc_499", "rs2228000"),
... ]
>>>
>>> names = [x[1] for x in genes]
```

Pick markers from HapMap data (cont.)


```
>>> names = [x[1] for x in genes]
>>> pops = []
>>> for i in range(1, 23):
...     print "Loading hapmap chromosome %d..." % i
...     pop = LoadPopulation('hapmap_%d.bin' % i)
...     markers = []
...     for name in names:
...         try:
...             idx = pop.locusByName(name)
...             markers.append(idx)
...         except:
...             pass
...     if len(markers) > 0:
...         markers.sort()
...         pop.removeLoci(keep=markers)
...         pops.append(pop)
...
>>> all = MergePopulationsByLoci(pops)
>>>
```

Acknowledgments

- W.M. Keck Foundation to the Gulf Coast Consortia through the Keck Center for Computational and Structural Biology
- Rice Terascale Cluster funded by NSF under Grant EIA-0216467, Intel, and HP
- M.D. Anderson Cancer Center High Performance Cluster
- BP was supported in part by a grant CA75432 from NCI
- Yaji Xu helped with all the figures in this talk

For further reading

 **Bo Peng** and Marek Kimmel (2005). simuPOP: a forward-time population genetics simulation environment. *Bioinformatics,* 21:3686–3687

 **Bo Peng** and Marek Kimmel (2007) Simulations provide support for the common disease common variant hypothesis. *Genetics.* 175:763-776.

 **Bo Peng**, Christopher I. Amos and Marek Kimmel (2007) Forward-time simulations of complex human diseases. *PLoS Genetics,* 3(3):e47.