In-depth course

Bo Peng, Ph.D.

Loading simuPOP

Population

Individual

Operator

Mating scheme

Simulator

Forward-time simulations using simuPOP, an in-depth course

Bo Peng, Ph.D.

Department of Epidemiology
UT MD Anderson Cancer Center
Houston, TX

June 15th, 2007
simuPOP workshop
School of Public Health, Department of Biostatistics
University of Alabama Birmingham

outline

Outline

1. **Loading simuPOP**
   - simuPOP modules
   - Random Number Generator
   - Debug information
   - Getting help

simuPOP modules

simuPOP provides six types of modules

1. Possible allele states:
   - **short** $0 \sim 2^8 - 1$
   - **long** $0 \sim 2^{16} - 1$
   - **binary** 0 and 1
2. Optimization
   - **standard** with debug information and runtime validation
   - **optimized** without debug information and runtime validation

Note: A Message Passing Interface (parallel) version of simuPOP is under development.

Loading appropriate module

1. Use `simuOpt.setOptions`

```
>>> from simuOpt import setOptions
>>> setOptions(alleleType='long', optimized=False, quiet=False)
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2006 Bo Peng
Developmental Version (Jun 12 2007) for Python 2.3.4
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)]
Random Number Generator is set to mt19937 with random seed 0x34679d9c2f218200
This is the standard long allele version with 65536 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>>
```

2. Set environment variables (system dependent)
   - `SIMUALLELETYPE = short/long/binary`
   - `SIMUOPTIMIZED` for optimized version
3. Command line argument of scripts using the `simuOpt` module (`--optimized`)

Standard modules

Perform strict runtime check. Produce proper debug information if anything goes wrong.

```
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "course.py", line 1, in ?
    #!/usr/bin/env python
IndexError: src/genoStru.h:428 absolute locus index (10) out of range of 0 - 1
>>> pop.individual(20).setAllele(1, 0)
Traceback (most recent call last):
  File "course.py", line 1, in ?
    #!/usr/bin/env python
IndexError: src/population.h:452 individual index (20) is out of range of 0 ~ 9
>>>
```

Optimized modules

No runtime check. Improper usages may crash simuPOP.

```
% setenv SIMUOPTIMIZED
% python
Python 2.3.4 (#1, Jan  9 2007, 16:40:09)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2006 Bo Peng
Developmental Version (May 21 2007) for Python 2.3.4
[GCC 3.4.6 20060404 (Red Hat 3.4.6-3)]
Random Number Generator is set to mt19937 with random seed 0x2f04b9dc5ca0fc00
This is the optimied short allele version with 256 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
1.2731974748756028e-313
>>> pop.individual(20).setAllele(1, 0)
Segmentation fault
```

Random Number Generator

simuPOP uses RNG from the GNU Scientific Library

```
>>> rng().name()
'mt19937'
>>> rng().seed()
3776160106704634368
>>> r = ListAllRNG()
>>> print r[:5]
('gfsr4', 'mt19937', 'mt19937_1999', 'mt19937_1998', 'r250')
>>> SetRNG('taus2', 1234)
>>> rng().name()
'taus2'
>>> rng().seed()
1234
>>> rng().randUniform01()
0.8298944395501166
>>>
```

Note: simuPOP uses system clock to set random seeds under windows.

Debug information

Several ways to turn on/off debug information

- Set environment variable `SIMUDEBUG`
- Use function `TurnOnDebug, TurnOffDebug`
- Use operator `turnOnDebug, turnOffDebug` to turn on/off debug at specific generations

Debug information (cont.)

```
>>> TurnOnDebug(DBG_POPULATION)
>>> ind = population(10, loci=[5]).individual(1)
Constructor of population is called
Destructor of population is called
>>> # This line may crash simuPOP
>>> print ind.allele(2)
0
>>> # Show all debug code
>>> ListDebugCode()
Debug code                    On/Off
DBG_ALL                         0
DBG_GENERAL                     1
DBG_UTILITY                     0
DBG_OPERATOR                    0
DBG_SIMULATOR                   0
DBG_INDIVIDUAL                  0
DBG_OUTPUTER                    0
DBG_MUTATOR                     0
DBG_RECOMBINATOR                0
DBG_INITIALIZER                 0
DBG_POPULATION                  1
```

Getting help


```
>>> help(population.addInfoFields)
Help on method population_addInfoFields:

population_addInfoFields(...) unbound simuPOP_la.population method
    Description:

        add one or more information fields to a population

    Usage:

        x.addInfoFields(fields, init=0)

    Arguments:

        fields:         new information fields. If one or more of the
                        fields alreay exist, they will simply be re-
                        initialized.
        init:           initial value for the new fields.

>>>
```

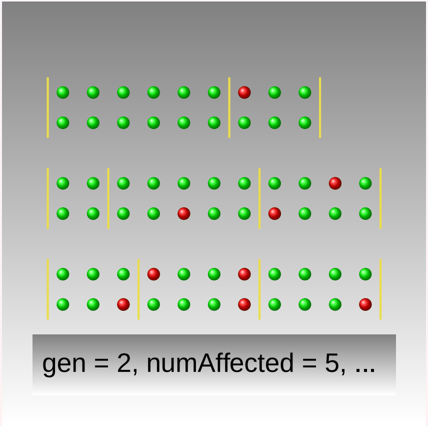
Outline

2 Population

- Structure of population
- Genotypic structure
- Population structure
- Population variables
- Manipulate population

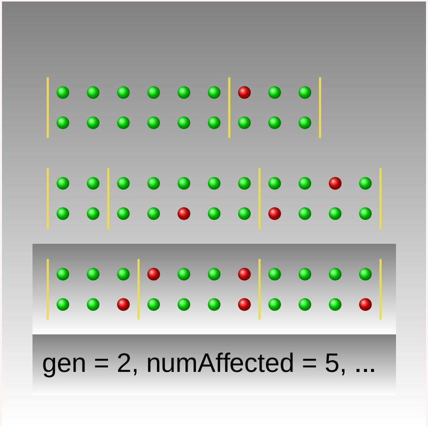
Structure of a population

- Unaffected
- Affected



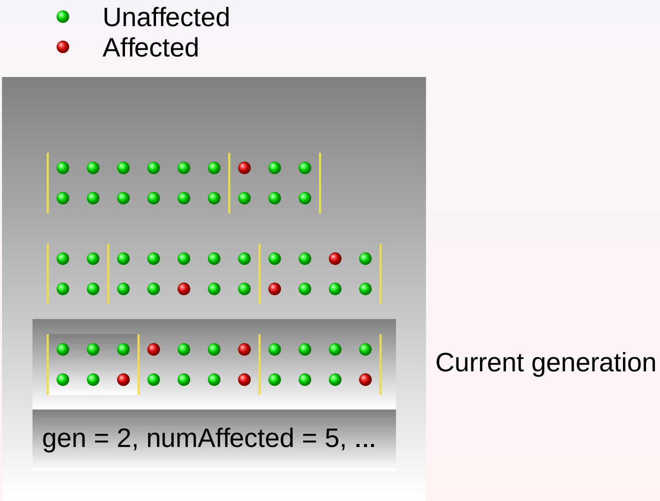
Structure of a population

- Unaffected
- Affected



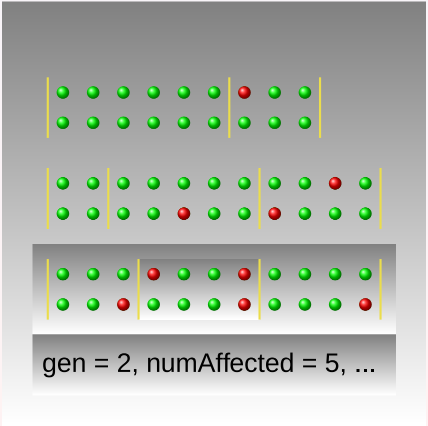
Current generation

Structure of a population



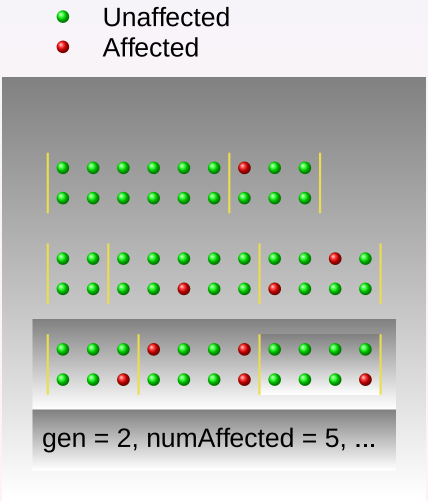
Structure of a population

- Unaffected
- Affected



Current generation

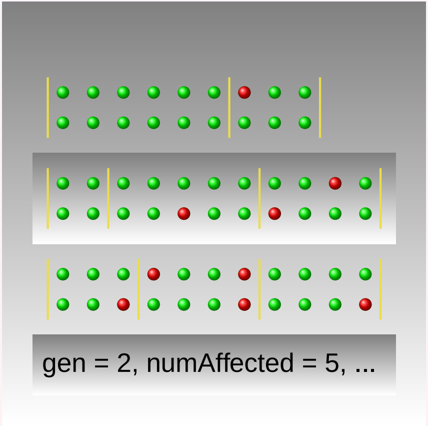
Structure of a population



Current generation

Structure of a population

- Unaffected
- Affected

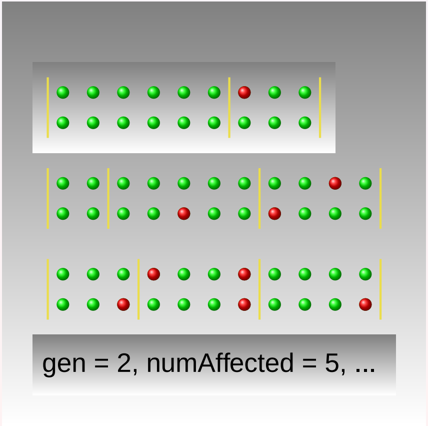


Ancestral generation 1

Current generation

Structure of a population

- Unaffected
- Affected

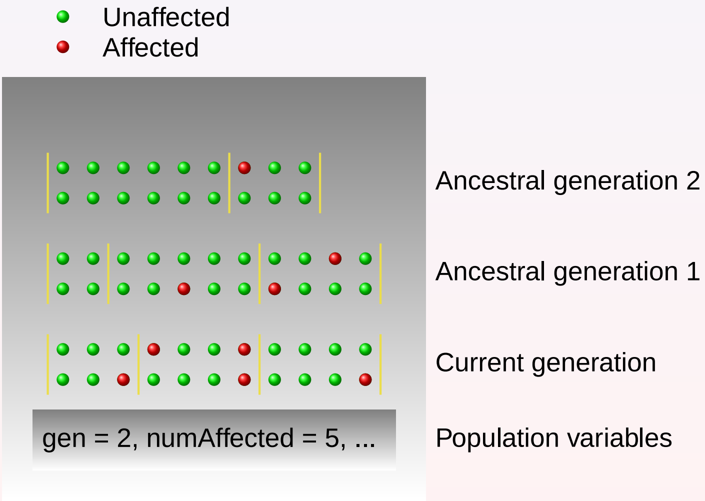


Ancestral generation 2

Ancestral generation 1

Current generation

Structure of a population



Genotypic Structure

All individuals have the same genotypic structure, which refers to

- Ploidy (diploid, haploid, triploid, ...)
- Number of chromosomes
- Number of loci on each chromosome
- Names and positions of loci
- Names of information fields
- Allele names
- Existence of sex chromosome

Create a population


```
>>> pop = population(size=10, loci=[2, 3])
>>> Dump(pop)
Ploidy:                 2
Number of chrom:        2
Number of loci:         2 3
Maximum allele state:   65535
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=4,
...       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()
'tetraploid'
>>> pop.individual(1).allele(1, 2)
0
>>>
```

Create a population with subpopulations


```
>>> pop = population(subPop=[2, 5, 6], loci=[2])
>>> print pop.popSize()
13
>>> print pop.subPopSizes()
(2, 5, 6)
>>> print pop.subPopSize(1)
5
>>> Dump(pop, infoOnly=True)
Ploidy:                 2
Number of chrom:        1
Number of loci:         2
Maximum allele state:   65535
Loci positions:
                1 2
Loci names:
                loc1-1 loc1-2
population size:        13
Number of subPop:       3
Subpop sizes:           2 5 6
Number of ancestral populations:        0
>>>
```

Mating happens within subpopulation


```
>>> pop = population(subPop=[5, 6], loci=[2])
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = [
...         initByFreq(alleleFreq=[0.2, 0.8], subPop=[0]),
...         initByFreq([0, 0, 0, 0.5, 0.5], subPop=[1])
...         ],
...     ops = [
...         dumper(alleleOnly=True, indRange=[[0, 3], [5, 7]]),
...         recombinator(rate=0.1) ],
...     end = 1
... )
```

Mating happens within subpopulation (cont.)


```
individual info:
sub population 0:
   0: MU   1  1 |   1  1
   1: FU   0  1 |   1  1
   2: FU   1  1 |   1  1
sub population 1:
   5: FU   4  4 |   4  3
   6: MU   4  4 |   4  4
End of individual info.


No ancenstral population recorded.
individual info:
sub population 0:
   0: FU   1  1 |   0  1
   1: FU   1  1 |   1  1
   2: MU   1  1 |   0  1
sub population 1:
   5: MU   4  4 |   4  3
   6: FU   4  4 |   4  3
End of individual info.
```

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]
0.7
>>> from simuUtil import ListVars
>>> ListVars(pop.dvars(), useWxPython=False)
 grp :  -1
 rep :  -1
 alleleNum :
   [1]
     [0]          21
     [1]          8
     [2]          1
 genoFreq :
   [2]
     [0]
       0 :        0.266666666667
       1 :        0.4
       2 :        0.266666666667
     [1]
       1 :        0.0666666666667
 genoNum :
   [2]
     [0]
       0 :        4.0
       1 :        6.0
       2 :        4.0
     [1]
       1 :        1.0
 alleleFreq :
```

Population variables (cont.)


```
subPop
  [0]
    alleleNum :
      [1]
        [0]    6
        [1]    3
        [2]    1
    genoNum :
      [2]
        [0]
          1 :  3.0
          2 :  2.0
    genoFreq :
      [2]
        [0]
          1 :  0.6
          2 :  0.4
    alleleFreq :
      [1]
        [0]    0.6
        [1]    0.3
        [2]    0.1
  [1]
    alleleNum :
      [1]
        [0]    15
```

Population manipulation


```
>>> # make a copy of pop
>>> pop = population(1000, loci=[2,3])
>>> 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])
>>> # 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  2.000000
2       loc1-1_1        1.000000
2       loc1-2_1        2.000000
3       loc2-1  1.000000
3       loc2-2  2.000000
3       loc2-3  3.000000

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

Population manipulation (cont.)

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

>>>
```

Outline

3 Individual

- Structure of individual
- Individual object
- Information fields
- Iterate through a population

Structure of individual

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_idx	···

Structure of individual

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

Male

Affected

fitness	father_idx	...

Structure of individual

Assume ploidy = 2, maxAllele = 1

Chromosome 0

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

Chromosome 1

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

Male

Affected

fitness	father_idx	···

Structure of individual

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

fitness	father_idx	···

Structure of individual

In-depth course
Bo Peng, Ph.D.
Loading simuPOP
Population
Individual
Structure of individual
Individual object
Information fields
Iterate through a population
Operator
Mating scheme
Simulator

Assume ploidy = 2, maxAllele = 1

Chromosome 0

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

Chromosome 1

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

Sex: Male

Affection status: Affected

fitness	father_idx	…

Structure of individual

Assume ploidy = 2, maxAllele = 1

Chromosome 0

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

Chromosome 1

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

Sex: Male

Affection status: Affected

Information fields:

fitness	father_idx	...

Individual


```
>>> pop =  population(subPop=[100, 200], loci=[2, 3])
>>> # the first individual
>>> ind1 = pop.individual(0)
>>> # the second individual in the second subpop
>>> ind2 = pop.individual(1, 1)
>>> # genotypic strcuture
>>> print ind1.numLoci(1)
3
>>> print ind1.numChrom()
2
>>> # an editable allele list
>>> alleles = ind1.arrGenotype(0)
>>> alleles[:] = range(ind1.totNumLoci())
>>> print ind1.arrGenotype(0)
[0, 1, 2, 3, 4]
>>> # ploidy 1, index 4
>>> ind1.setAllele(3, 4, 1)
>>> print ind1.allele(4, 1)
3
>>>
```

Information fields

Pieces of information that can be attached to each individual, e.g.

- `fitness:` fitness of each individual, calculated by selectors
- `father_idx, mother_idx`: index of parents in the parental generation
- `old_index:` index of an individual in the population where it is sampled

Or, self-defined

- birthday
- geographic location
- ...

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')
48.0 40.0
>>> print ind1.info('father_idx'), ind1.info('mother_idx')
48.0 40.0
>>>
>>>
```

Iterate through a population

```
>>> 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 invividuals 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: 1 Fit: 0.200 Geno: 0 1
Aff: 0 Fit: 0.200 Geno: 0 2
Aff: 0 Fit: 0.200 Geno: 1 0
Aff: 0 Fit: 0.200 Geno: 1 0
Aff: 0 Fit: 0.200 Geno: 2 0
Aff: 0 Fit: 0.050 Geno: 0 0
Aff: 0 Fit: 0.050 Geno: 0 0
Aff: 0 Fit: 0.500 Geno: 1 1
>>>
```

Outline

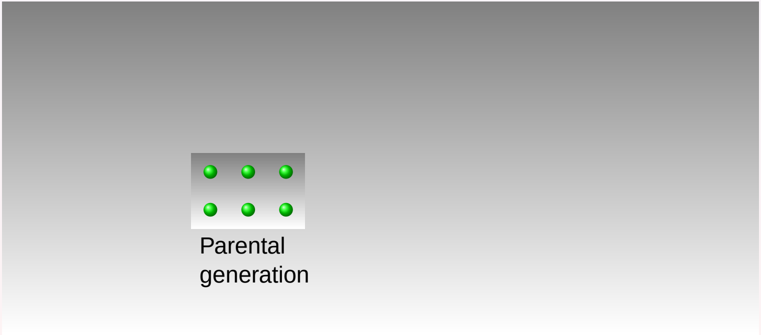
4. **Operator**
   - Function form of an operator
   - Stage of an operator
   - Applicable generations
   - Replicates and replicate groups
   - Output and output expression
   - Python Operators

Function form of an operator

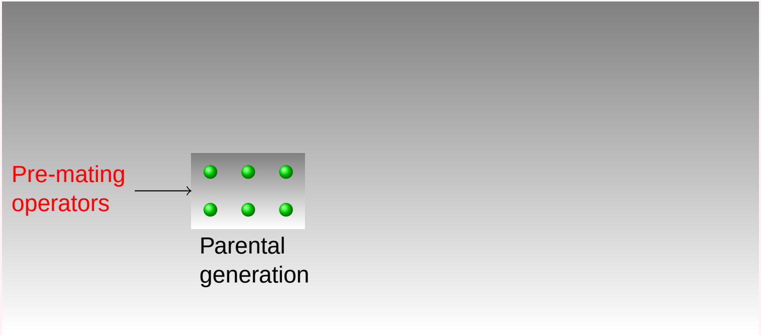
Many operators can be applied to a population in its function form. For example `Dump(pop, args)` is equivalent to `dumper(args).apply(pop)`. For example

operator	Function	operator	Function
dumper	Dump	mergeSubPops	MergeSubPops
initByFreq	InitByFreq	splitSubPop	SplitSubPop
initByValue	InitByValue	mapSelector	MapSelect
stat	Stat	maSelector	MaSelect
kamMutator	KamMutate	mapPenetrance	MapPenetrance
smmMutator	SmmMutate	caseControlSample	CaseControlSample
pyMutator	PyMutate	largePedigreeSample	LargePedigreeSample

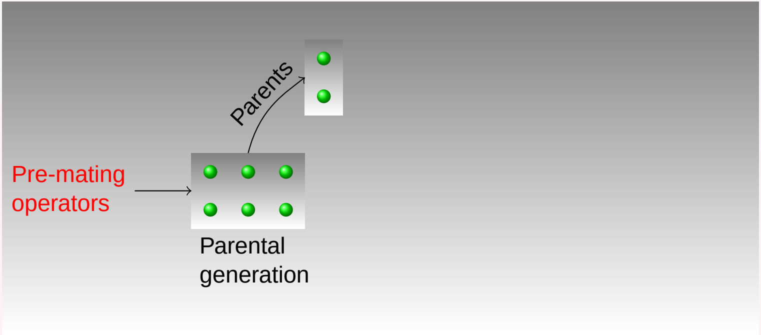
Stage of an operator



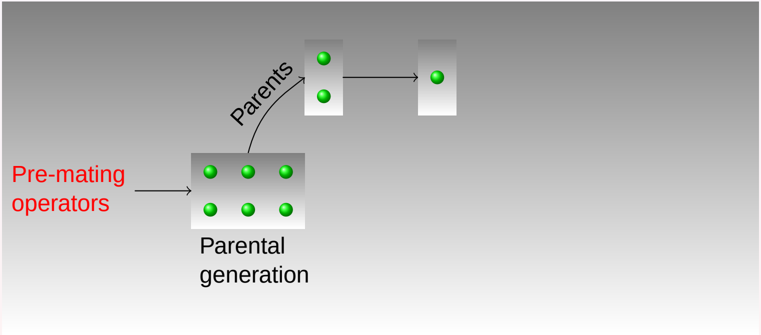
Stage of an operator



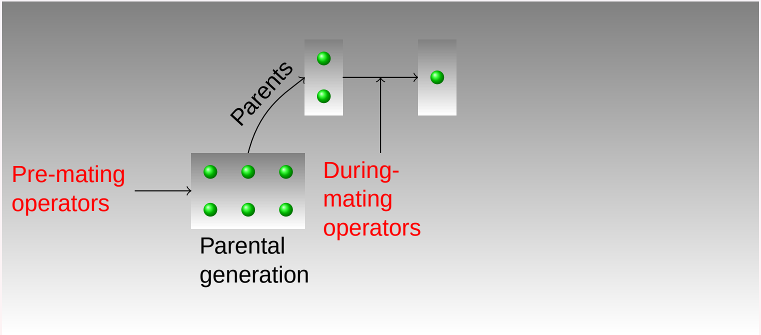
Stage of an operator



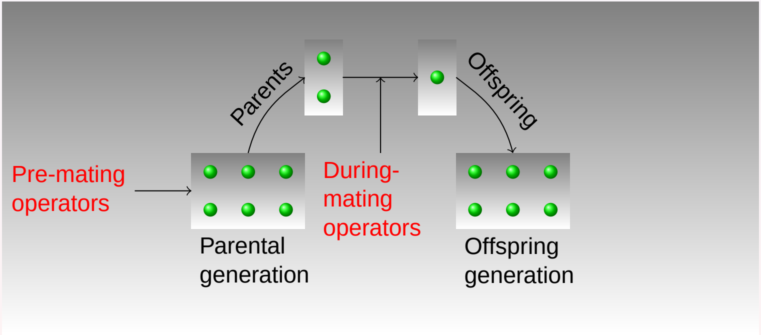
Stage of an operator



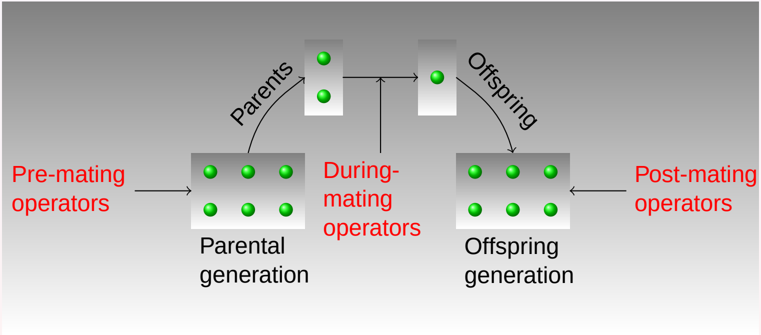
Stage of an operator



Stage of an operator



Stage of an operator



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: 0.304200
Gen:  10  Freq: 0.290700
Gen:  20  Freq: 0.285300
Gen:  30  Freq: 0.288750
Gen:  40  Freq: 0.283750
Gen:  50  Freq: 0.284100
Last Gen:  50  Freq: [0.28410000000000002, 0.71589999999999998]
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
... )
0.47 0.52
0.49 0.56
0.51 0.60
0.52 0.62
0.56 0.60
0.52 0.62
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()
0.56 0.55 0.46 0.47 0.54
0.56 0.55 0.42 0.55 0.57
0.58 0.56 0.40 0.57 0.56

>>> print open('out1').read()
0.56 0.55 0.56 0.55 0.58 0.56
>>> print open('out2').read()
0.46 0.47 0.54 0.42 0.55 0.57 0.40 0.57 0.56
>>>
```

Python operator

A Python operator is an operator that calls a user-provided Python function when it is applied to a population. A hybrid operator performs its main function at the C++ level, and a pure Python operator depends on this user-provided function for its functionality.

A hybrid operator

A (weird) selector with fitness

	BB	Bb	bb
AA	1.	1.01	1.02
Aa	1.	0.99	0.98
aa	1.	1.01	1.02

```
>>> def mySelector(geno, gen):
...     if geno[0] + geno[1] == 0:
...         return (1, 1.01, 1.02)[geno[2] + geno[3]]
...     elif geno[0] + geno[1] == 1:
...         return (1, 0.99, 0.98)[geno[2] + geno[3]]
...     else:
...         return (1, 1.01, 1.02)[geno[2] + geno[3]]
...
```

Note: This operator can be more efficiently implemented using other non-Python operators.

A hybrid operator (cont.).


```
>>> expr = r'"%.3f %.3f\n" % (alleleFreq[0][0], alleleFreq[1][0
>>> simu = simulator(
...     population(10000, loci=[1,1],
...         infoFields=['fitness']),
...     randomMating(),
... )
>>> simu.evolve(
...     preOps = [initByFreq([0.3, 0.7])],
...     ops = [
...         pySelector(loci=[0, 1], func=mySelector),
...         stat(alleleFreq=[0, 1], step=20),
...         pyEval(expr, step=20)
...         ],
...     end = 100
... )
0.294 0.298
0.252 0.278
0.184 0.246
0.134 0.232
0.078 0.215
0.047 0.209
True
>>>
```

A pure Python operator


```
>>> from random import normalvariate
>>> def trait(ind):
...     return [ind.info('trait') + normalvariate(0, 1)]
...
>>> def avgTrait(pop):
...     t = sum(pop.indInfo('trait', False))/pop.popSize()
...     pop.dvars().trait = t
...     print 'Average trait at gen %4d : %.4f' % (pop.gen(), t)
...     return True
...
>>> simu = simulator(
...     population(100, infoFields=['trait']),
...     randomMating()
... )
>>> simu.evolve(
...     ops = [
...         pyIndOperator(func=trait, infoFields=['trait']),
...         pyOperator(func=avgTrait, step=100),
...     ],
...     end = 500
... )
Average trait at gen    0 : 0.0218
Average trait at gen  100 : 1.2040
Average trait at gen  200 : 0.8297
Average trait at gen  300 : 0.9698
Average trait at gen  400 : 0.3187
Average trait at gen  500 : -0.2373
True
>>>
```

Outline

5. **Mating scheme**
   - Demographic model
   - Number of offspring

Mating schemes

Mating schemes

- Populate offspring subpopulation from corresponding parental subpopulation
- Can not change number of subpopulations
- 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>]
>>>
```

Outline

6 **Simulator**
- What a simulator does
- Simulator operations
- Retrieve populations

Simulator

A simulator manages

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

and evolve the populations.

simulator operations


```
>>> simu = simulator(
...     population(size=10000, loci=[3]),
...     randomMating(),
... )
>>> # genotypic structure can be accessed at the simulator level
>>> print simu.lociPos()
(1.0, 2.0, 3.0)
>>> simu.step(ops = [])
True
>>> print simu.gen()
1
>>> # add information fields to all populations
>>> simu.addInfoFields(['father_idx', 'mother_idx'])
>>> simu.setMatingScheme(randomMating(numOffspring=2))
>>>
```

Get populations from a simulator


```
>>> # get a reference to the first replicate
>>> pop = simu.population(0)
>>> pop.individual(0).setAllele(1, 0)
>>> print simu.population(0).individual(0).allele(0)
1
>>> # get a real copy
>>> pop = simu.getPopulation(0)
>>> pop.individual(0).setAllele(1, 1)
>>> print simu.population(0).individual(0).allele(1)
0
>>>
```