Forward-time simulations using simuPOP, selected topics

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

How population sizes are changed

- Some simuPOP functions and operators can forcefully change population sizes
  - operators: `splitSubPop`, `mergeSubPops`
  - population member functions: `resize`, `splitSubPop`, `splitSubPopByProportion`, `mergeSubPops`, `mergePopulation`, `mergePopulationByLoci`
  - global functions: `MergePopulations`, `MergePopulationsByLoci`
- Mating schemes can generate offspring generation with different subpopulation sizes
  - parameter: `newSubPopSize`, `newSubPopSizeExpr`, `newSubPopSizeFunc` (recommended)

Population manipulation

```
>>> pop = population(subPop=[100, 200], loci=[1])
>>> pop.splitSubPop(0, [20, 80])
>>> # Note that subpop 1 is intact
>>> print pop.subPopSizes()
(20, 200, 80)
>>> pop.splitSubPopByProportion(1, [0.4, 0.6])
>>> print pop.subPopSizes()
(20, 80, 80, 120)
>>> # merge
>>> pop.mergeSubPops([1,2])
>>> # Note that subpopulation 2 is not removed
>>> print pop.subPopSizes()
(20, 160, 0, 120)
>>> pop.removeEmptySubPops()
>>> print pop.subPopSizes()
(20, 160, 120)
>>> pop1 = pop.clone()
>>> pop.mergePopulation(pop1)
>>> print pop.subPopSizes()
(20, 160, 120, 20, 160, 120)
>>>
```

parameter newSubPopSizeFunc

```
>>> def demo(gen, oldsize):
...     return [x+10 for x in oldsize]
...
>>> simu = simulator(
...     population(subPop=[100, 200]),
...     randomMating(newSubPopSizeFunc=demo)
... )
>>> simu.evolve(
...     ops = [
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     end=5
... )
[110, 210]
[120, 220]
[130, 230]
[140, 240]
[150, 250]
[160, 260]
True
>>>
```

Split and grow?

```
>>> def demo(gen, oldsize):
...     if gen < 4:
...         return [100]
...     else:
...         return [50+gen]*3
...
>>> simu = simulator(
...     population(size=100),
...     randomMating(newSubPopSizeFunc=demo)
... )
>>> simu.evolve(
...     ops = [
...         splitSubPop(which=0, proportions=[0.2, 0.4, 0.4], at=[4]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     end=10
... )
[100]
[100]
[100]
[100]
[54, 54, 54]
[55, 55, 55]
[56, 56, 56]
[57, 57, 57]
[58, 58, 58]
[59, 59, 59]
[60, 60, 60]
True
>>>
```

Outline

2 **Genetic models**

- Mutation
- Recombination
- Selection
- Migration
- Penetrance
- Quantitative traits

Mutation

mutation in simuPOP refers to the change of allele at a locus from one state to another.

- mutation happens at a given rate μ
- a mutator determines how allele state changes

Note that:

- No direct support for infinite allele or infinite site models
- No direct support for state-dependent mutation rates (μ_S and μ_N)

K-allele mutation model: `kmmMutator`

With a `kmmMutator`, the allele will be mutated to any other allele state with equal probability $\frac{\mu}{k-1}$.

```
>>> simu = simulator(
...     population(size=1000, loci=[1]),
...     randomMating()
... )
>>> simu.step(ops = [
...     kamMutator(rate=0.5, atLoci=[0], maxAllele = 5),
...     stat(alleleFreq=[0]),
...     pyEval(r'", ".join(["%.2f" % x for x in alleleFreq[0]])
...     ]
... )
0.49, 0.09, 0.10, 0.10, 0.11, 0.11
True
>>>
```

Stepwise mutation model: `smmMutator`

With a `smmMutator`, the allele is increased or decreased by one. This is a classical model for the mutation of microsatellite markers.

```
>>> simu = simulator(
...     population(size=1000, loci=[1]),
...     randomMating()
... )
>>> simu.evolve(
...     preOps = [initByValue([5])],
...     ops = [
...         smmMutator(rate=0.1, atLoci=[0], incProb=0.4),
...         stat(alleleFreq=[0]),
...         pyEval(r'", ".join(["%.2f" % x for x in alleleFreq[0]]) + "\n"')
...     ],
...     end=5
... )
0.00, 0.00, 0.00, 0.00, 0.05, 0.92, 0.03
0.00, 0.00, 0.00, 0.00, 0.11, 0.83, 0.06, 0.00
0.00, 0.00, 0.00, 0.01, 0.15, 0.75, 0.09, 0.00
0.00, 0.00, 0.00, 0.01, 0.17, 0.70, 0.10, 0.01
0.00, 0.00, 0.00, 0.02, 0.19, 0.65, 0.12, 0.01
0.00, 0.00, 0.01, 0.03, 0.20, 0.62, 0.13, 0.01
True
>>>
```

Recombination

simuPOP allows

- uniform recombination rate between all adjacent loci
- varying recombination rate between adjacent loci
- a recombination intensity with recombination rate between adjacent loci determined by loci distances
- sex-specific recombination rate

Varying recombination rates

```
>>> simu = simulator(
...     population(size=1000, loci=[1000]),
...     randomMating()
... )
>>> rates = [0.00001]*400 + [0.0001]*200 + [0.000001]*399
>>> expr = r'"%.3f %.3f %.3f\n" % (LD[100][101], LD[500][501], LD[900][901])'
>>> simu.evolve(
...     preOps = [initByValue([1]*1000+[2]*1000)],
...     ops = [
...         recombinator(rate=rates, afterLoci=range(999)),
...         stat(LD=[[100,101], [500, 501], [900, 901]]),
...         pyEval(expr, step=100)
...     ],
...     end = 1000
... )
0.250 0.249 0.250
0.249 0.249 0.248
0.242 0.226 0.249
0.208 0.210 0.241
0.160 0.174 0.199
0.129 0.134 0.243
0.183 0.169 0.210
0.202 0.153 0.237
0.173 0.184 0.242
0.191 0.148 0.235
0.153 0.136 0.250
True
>>>
```

Recombination intensity

```
>>> dist = [0.01]*400+[0.1]*200+[0.01]*400
>>> simu = simulator(
...     population(size=1000, loci=[1000],
...         lociPos=[sum(dist[:x]) for x in range(1000)]),
...     randomMating()
... )
>>> expr = r'"%.3f %.3f %.3f\n" % (LD[100][101], LD[500][501], LD[900][901])'
>>> simu.evolve(
...     preOps = [initByValue([1]*1000+[2]*1000)],
...     ops = [
...         recombinator(intensity=0.001),
...         stat(LD=[[100,101], [500, 501], [900, 901]]),
...         pyEval(expr, step=100)
...     ],
...     end = 1000
... )
0.250 0.250 0.250
0.249 0.202 0.208
0.247 0.246 0.158
0.200 0.249 0.088
0.147 0.245 0.000
0.221 0.225 0.000
0.248 0.233 0.000
0.242 0.180 0.000
0.235 0.209 0.000
0.244 0.192 0.000
0.250 0.183 0.000
True
>>>
```

How selection is implemented

Selection in simuPOP is implemented as ***probability (ability) to mate***, achieved by two components.

1. A selector sets information field `fitness` of individuals in given subpopulations and mark these subpopulations as under selection.
2. A mating scheme chooses parents with probabilities in proportional to their `fitness` values. Assuming that f_i is the fitness value of individual i,

$$Pr\left(\text{individual } i \text{ is selected}\right) = \frac{f_i}{\sum_{j=0}^{N} f_j}$$

Note: offspring is not subject to selection.

Map selector: `mapSelector`

User provide a `map` of fitness values of genotype at given loci. For example

```
mapSelector(locus=1,
    fitness={'0-0':1, '0-1':0.9, '1-1':0.9}
```

It can also handle multi-locus cases, with genotypes coded as

```
mapSelector(loci=[1, 3],
    fitness={'0-0|0-0': 1, '0-0|0-1': 1.01,
             '0-0|1-1': 1.02, '0-1|0-0': 0.99,
             '0-1|0-1': 0.99, '0-1|1-1': 0.99,
             '1-1|0-0': 1, '1-1|0-1': 1,
             '1-1|1-1': 1}
)
```

You can see that this is not particularly easy.

Multi-allele selector: `maSelector`

Assuming two classes of alleles: wildtype (`W`) and disease susceptibility alleles (`S`), a multi-allele selector uses three values for genotype `WW`, `WS` and `SS` at each locus.

```
maSelector(locus=0, wildtype=[0],
    fitness=[1, 1.0001, 1.0002])
```

Multi-allele selector: `maSelector` (cont.)

In the n-locus cases, an array of length 3^n is expected, e.g.

```
s = 0.0001
maSelector(loci=[1,3], wildtype=[0],
    fitness=[1, 1+s, 1+2*s,
                 1, 1-s, 1-s,
                 1, 1-s, 1-s])
```

selects individuals with a fitness model

	BB	Bb	bb
AA	1	1.0001	1.0002
Aa	1	0.9999	0.9999
aa	1	0.9999	0.9999

Multi-locus selector: `mlSelector`

Multi-locus selector obtains fitness values at each locus, calculated by other selectors, and combine them to a single `fitness` value. The compounding method is determined by the `mode` parameter

- `mode=SEL_Additive`: $f = 1 - \sum_i (1 - f_i)$
- `mode=SEL_Multiplicative`: $f = \prod_i f_i$
- `mode=SEL_Heterogeneity`: $f = 1 - \prod_i (1 - f_i)$

E.g.

```
mlSelector( [
  maSelector(locus=0, fitness=(1, 1.01, 1.02)),
  maSelector(locus=1, fitness=(1, 0.99, 0.98)),
  ], mode=SEL_Additive)
```

Migration operator: `migrator`

The simuPOP `migrator` accepts a migration rate matrix, that is interpreted differently depending on the `mode` parameter, which can be

MigrByProbability This is the default migration mode where migration rate is treated as $r_{ij} = Pr$ (migrate from subpop i to j). $r_{ii} = 1 - \sum_{j \neq i} r_{ij}$ is automatically calculated.

MigrByProportion A given proportion of individuals in a subpopulation will migrate to other subpopulations.

MigrByCount A given number of individuals in a subpopulation will migrate to other subpopulations.

Migration

```
>>> simu = simulator(
...     population(subPop=[1000]*5),
...     randomMating()
... )
>>> simu.evolve(
...     ops = [
...         migrator(rate=[
...             [0, 0.2, 0.1],
...             [0, 0, 0.1],
...             [0.2, 0.2, 0]],
...             fromSubPop=[1,2,3], toSubPop=[1,2,3]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     end = 3
... )
[1000, 883, 1330, 787, 1000]
[1000, 790, 1525, 685, 1000]
[1000, 701, 1669, 630, 1000]
[1000, 610, 1762, 628, 1000]
True
>>>
```

Penetrance models

Penetrance is the probability that an individual gets affected, condition on his/her genotype. Several penetrance operators and functions are provided.

- mapPenetrance (MapPenetrance)
- maPenetrance (MaPenetrance)
- pyPenetrance (PyPenetrance)

These penetrance operators (functions) calculate penetrance values for each individual, and assign affection status randomly.

Operator PyPenetrance

```
>>> def myPene(geno):
...     return (0.01, 0.1, 0.3)[sum(geno)]
...
>>> simu = simulator(
...     population(size=20000, loci=[1]),
...     randomMating()
... )
>>> expr = r'"%s (%.3f)\n" % (numOfAffected, '
>>>         '1.*numOfAffected/popSize)'
  File "topics.py", line 1
    '1.*numOfAffected/popSize)'
   ^
IndentationError: unexpected indent
>>> simu.evolve(
...     preOps = [initByFreq([0.9, 0.1])],
...     ops = [
...         pyPenetrance(locus=0, func=myPene),
...         stat(numOfAffected=True, popSize=True),
...         pyEval(expr, step=10),
...     ],
...     end=20
... )
Traceback (most recent call last):
```

Quantitative traits

There are many quantitative trait models so `pyQuanTrait` and its function form `PyQuanTrait` are most frequently used.

```
>>> pop = population(100, loci=[1, 1], infoFields=['qtrait'])
>>> InitByFreq(pop, [0.4, 0.6])
>>> def qtrait(geno):
...     return sum(geno)
...
>>> PyQuanTrait(pop, loci=[0, 1], func=qtrait)
>>> for i in range(5):
...     ind = pop.individual(i)
...     print '%d %d %d %d: %.2f' % (ind.allele(0, 0),
...         ind.allele(1, 0), ind.allele(0, 1),
...         ind.allele(1, 1), ind.info('qtrait'))
...
1 0 0 0: 1.00
0 0 0 1: 1.00
1 0 0 0: 1.00
1 1 1 0: 3.00
1 0 0 0: 1.00
>>>
```

Outline

3 **Misc topics**

- Pause a simulation
- User interface
- Use of wxPython
- Integration with R

Pause and time a simulation

Use `pause()` operator to pause the evolution by

- `pause(...)` pauses the simulation at given generations
- `pause(stopOnKeyStroke)` pauses the simulation at key stroke.

Use `ticToc()` operator to display elapsed time

Operators: `pause` and `ticToc`

```
>>> simu = simulator(
...     population(size=50000, loci=[100]*10),
...     randomMating()
... )
>>> simu.evolve(
...     ops = [
...         pause(stopOnKeyStroke=True),
...         ticToc(step=10),
...     ],
...     end = 50
... )
Elapsed Time: 0s  Overall Time: 00:00:00
Elapsed Time: 7s  Overall Time: 00:00:07
Elapsed Time: 6s  Overall Time: 00:00:13
Elapsed Time: 6s  Overall Time: 00:00:19
Elapsed Time: 5s  Overall Time: 00:00:24
Elapsed Time: 6s  Overall Time: 00:00:30
True
>>>
```

User Interface

See, for example, `simuLDDecay.py`

Use of wxPython

If `wxPython` is installed, it will be used

- simuOpt will use wxPython to build parameter dialogs, instead of the outdated tk dialogs
- ListVars(pop.dvars()) will use wxPython to display population variables
- simuViewPop can be used to view population information
  - run simuViewPop.py file.bin' to view a saved population
  - In python, import simuViewPop and call viewPop(pop)

Use of wxPython (cont.)

```
>>> pop = population(1000, loci=[3,5])
>>> InitByFreq(pop, [.2, .8])
>>> Stat(pop, alleleFreq=range(8), LD=[1,2])
>>> from simuUtil import ListVars
>>> ListVars(pop.vars())
>>> import sys
>>> sys.path.append('../scripts')
>>> from simuViewPop import *
>>> viewPop(pop)
>>>
```

Integration with R

```
>>> from rpy import *
Unable to determine R version from the registry. Trying another
RHOME= c:\PROGRA~1\R\R-24~1.1
RVERSION= 2.4.1
RVER= 2041
RUSER= C:\
Loading the R DLL c:\PROGRA~1\R\R-24~1.1\bin\R.dll .. Done.
Loading Rpy version 2041 .. Done.
Creating the R object 'r' ..  Done
>>> def association(pop, loci):
...     Stat(pop, alleleFreq=loci)
...     a1 = pop.dvars().alleleNum[loci[0]][0]
...     a2 = 2*pop.popSize() - a1
...     b1 = pop.dvars().alleleNum[loci[1]][0]
...     b2 = 2*pop.popSize() - b1
...     print '%4d %4d %4d %4d: %.3f' % (a1, a2, b1, b2,
...         r.chisq_test(with_mode(NO_CONVERSION, r.matrix)(
...         (a1, a2, b1, b2), ncol=2, byrow=True))['p.value'])
...     return True
...
```

Integration with R (cont.)

```
>>> simu = simulator(
...     population(10000, loci=[2]),
...     randomMating()
... )
>>> simu.evolve(
...     preOps=[initByValue([0,1,1,0])],
...     ops = [
...         recombinator(rate=0.0001),
...         pyOperator(func=association, param=[0,1], step=20)
...     ],
...     end=100
... )
9929 10071 10070 9930: 0.162
9597 10403 10397 9603: 0.000
9242 10758 10748 9252: 0.000
9285 10715 10741 9259: 0.000
9531 10469 10461 9539: 0.000
10101 9899 9867 10133: 0.020
True
>>>
```