Introduction to NewGarden

Stephan Pelikan
pelikan@math.uc.edu
Department of Mathematical Sciences
and
Steven Rogstad
Department of Biology
University of Cincinnati
Cincinnati OHIO 45221

August 2, 2011

Contents

1 Introduction **2**

2 Running the program **2**
2.1 Running a java program 2
2.1.1 shell scripts 3

3 A sample run described **4**

4 The output **5**

5 The XML file **5**
5.1 The Document Type Definition 6
5.2 LOCI 7
5.3 Dioecious 8
5.4 Reproduction 8
5.5 Offspring distribution 9
5.6 Pollen Rate 10
5.7 Mortality 10
5.8 Selfing 11
5.9 Random mating 11
5.10 Dispersal of recruits 11
5.11 Pollen dispersal 13
5.12 Regions 14
5.13 Summary regions 16
5.14 Number Generations 17

5.15 Number Runs 17
5.16 Initial Population 17
5.17 Founders F 18

6 Sources, copyrights etc. 18

1 Introduction

This document is intended as a quick outline for how you can work with the NewGarden program. It describes what the program is intended to do, how it does it, and how you can modify its behavior to use it in modeling a variety of situations.

There are a number of other sources of help and explanation too. The example `SimData.xml` file has comments edited into it that, day to day, we find offer sufficient reminders of how the options are to be used. There are additional sample `Simulation_data` files that model different scenarios described in different chapters of the text. Finally, you will also find that the distribution includes snippets of text useful for pasting into `Simulation_data` parameter specification files as well as some sample scripts for running the program.

Additional supporting scripts and documents are available at
`http://math.uc.edu/~pelikan/NEWGARDEN`
where we intend to also distribute updated versions of the NewGarden documents and programs.

2 Running the program

Here we trace a sample run of the computer model, describing what it does. After that we'll trace it again describing how it does it.

As mentioned earlier, time is measured in terms of rounds of breeding and distances are expressed in terms of the mean distance between adjacent viable locations. In the version of the model implemented in the main NewGarden program these viable sites are assumed to form a square lattice of points in some rectangular region in the plane.

The program has many options and most will be explained later; here we consider a rather simple simulation that exercises the basic elements of the program.

The user specifies the parameters for the model by providing a data structure called *SumulationData* in an XML file; this procedure, too, will be explained later and for the time being we will work with a small simulation as specified in the file *SimData.xml.* Additionally, the user controls many options of the program by adding text to the command line that starts the program.

2.1 Running a java program

The NewGarden program is written in the Java programming language and so can be run on any computer for which there is a Java runtime environment available. These include all modern versions of Windows, Mac OS, and Linux.

The program is packaged into a single file called `NewGarden.jar` and there are, in fact, several programs included in this "Java ARchive". The program that runs the simulations is called `Model` and can be executed from a command line by entering

```
java -jar NewGarden.jar Model
```

This will generate an error message since the command doesn't provide all the information needed for correct execution of the Model program; usually one must select some command line options and also specify the name of a parameter file. The simplest way to execute the Model program correctly is to include a command line option asking that a help message be printed:

```
java -jar NewGarden.jar Model --help
```

The resulting output is a summary of the command line options for the program:

```
You asked for help
The command line options are
-h or --help: show this help message
-x or --xmlfile <filename>: specify the XML parameter file
-d or --dumpfile <filename>: put all the population data into the filename
                 or to stdout if <filename> is not given
-l or --lifetable <filename>: write Leslie lifetable to the file filename
                 or to stdout is <filename> is not given.
-p or --progressbar : show a progress bar on screen
-v to label all bits of output with date-time and program version
-r or --resultsfile <filename> to send the main results to the file filename
-s to include a copy of the XML specified parameters in the results file
-n or --showNe to have the program report estimates of Ne
```

To run the sample simulation we put the NewGarden.jar file and parameter file SimData.xml in the same directory (folder), make that the active directory (working directory) and give the command to run the Model program using the parameter file SimData.xml as in

```
java -jar NewGarden.jar Model -x SimData.xml
```

2.1.1 shell scripts

This direct command line approach works fine but most people end up creating a shell script to allow them to select a number of command line options easily. Sample scripts `run.bat` and `run.sh` (for Windows command line use and Linux/Mac command line with the `bash` shell use respectively) are provided on the CD.

For example, using windows one could then use the command line

```
run SimData.xml
```

And in Linux one might type a command like:

```
./run.sh SimData.xml
```

3 A sample run described

The program performs a user-specified number of repetitions of the simulation and accumulates the data needed to generate statistical summaries of the results at the end of all the trials.

In each run of the simulation the program begins with a specific list of plants (the founders of the population) and tracks the population through several rounds of breeding and mortality. It keeps track of the location, genotype, age, and sex of each of the plants in the virtual population.

The user specifies whether the population is mono- or dioecious and describes the nature of genes to simulate by giving the number of loci to simulate and the number of alleles (along with their frequencies) at each locus.

In each run of the simulation the program establishes a number of founders. This is done by introducing plants into the list of all population members according to the `Initial_Population` specified by the user. For each founder the user specifies the *age* at the start of the simulation, the *location* of the founder and the *sex* of the founder. The program selects the genotype of each founder by choosing at random with replacement from a population with user specified F value (FoundersF) and allele frequencies as specified by the user. Note that the frequencies of the alleles among the founders may be different due to sample size and sampling error; the same is true of the departure (F) from Hardy-Weinberg equilibrium.

The program then loops to simulate a number of rounds of breeding and mortality. Starting with a list of plants present at the start of the round of breeding (and therefore with the list of founders if it is the first round of breeding in the simulation) the program proceeds as follows:

Considering each member of the population (in a monoecious population) or each female member of the population (in a dioecious population) the program selects the number of potential recruits that this population member will generate during this round of breeding. This number depends on the *age* of the parent in a user-specified manner (ReproductionRate) and is determined by one of three user selectable rules (bracket,poisson,round).

The program then organizes data about the potential pollen donors for this parent; all populations members (if the population is monoecious) or all males (if dioecious) are assigned an age-dependent relative probability of donating pollen and are assigned into (user specified) distance classes according to how far their location is from the (female) parent. The chance that a pollen donor is selected from each of these distance classes is user specified.

Each potential recruit is then determined by selecting a male parent according to the age-dependent relative probabilities of donating pollen and according to user-specified probabilities that the donor belongs to each of the distance classes. The genotype of the potential recruit is determined at each locus by selecting at random from the parent's genes at the locus and the location of the potential recruit is determined by selecting a dispersal distance at random according to a user-specified distribution of dispersal distances (`Dispersal_Distance`) and then selecting at random from among the locations that lie at this distance from the (female) parent.

The potential recruit is the added to a list of all potential recruits generated by all parents during this round of breeding.

When all potential recruits from all population members have been generated, mortality is applied to the current population with each member having a user-specified, age-specific probability of death following each round of breeding.

After dead members have been removed from the population, selection is made from among the potential recruits to obtain new members for the population. Each unoccupied location is considered in turn and one potential recruit is chosen at random from all the potential recruits with that location to actually join the population.

Note that parents occupy locations and recruits are excluded from these locations unless the parent dies in the round of mortality before recruits are selected. Annuals work in this way with parent generation being removed entirely before offspring are established.

At the end of a single run of the simulation a list of all populations members is available including breeding cycle in which the plant was established and in which it died, its sex, location and genotype. Intermediate statistics are obtained and saved to be aggregated over all the runs of the simulation.

4 The output

When all the replications of the simulation are complete, summary data is printed (or, optionally, saved to a file). Data is summarized both for each cohort (produced during a given round of breeding) and for the total population standing at the end of each breeding-mortality-recruitment cycle.

The results calculated include, for each time period, the mean and standard deviation of the population size, of the number of distinct alleles in the population, of the expected heterozygosity H_e and of the observed, or actual, heterozygosity H_o, and of the the overall inbreeding coefficient F (technically F_{IT} of the population. These summary statistics are reported for the entire population and also (optionally) as a summary of that part of the population with locations in arbitrary user-specified subregions.

The output is comma separated and can generally be put in a file that can then be opened/read by spreadsheet programs and statistical packages for further processing, generation of statistical plots, and the like. There is a command line option that automatically directs the output to a file.

5 The XML file

The parameters for a simulation are specified in an XML file (e.g., see the file `SimData.xml`). These files are fairly easy to create and modify by hand while designing simulations, and are in a format that is intended to be parsed easily by computer programs.

The first line of any of the parameter files is

`<?xml version="1.0" standalone="yes"?>`

which simply announces that what follows is intended to be an XML file.

5.1 The Document Type Definition

The contents and format of any XML file are restricted and defined by its Document Type Definition (DTD). These restrictions make it easier to write computer programs that can read the data encoded in the file. In an XML file data is typically either enclosed in tagged elements that are marked off with labels enclosed in angle brackets, as in

`<program_name>NewGarden</program_name>`

or it is included as so called attributes in the element's tag itself, as in this example with two attributes:

```
<program_name version="2.2.1" date="Oct 2009">
NewGarden
</program_name>
```

Here the names of the attributes are `version` and `date` and the corresponding values are `2.2.1` and `Oct 2009` respectively. Attributes for an element are always specified in the element's starting tag using this name = value convention. You must always enclose the values in double quotation marks are illustrated here.

It is also possible to have a tagged element with attributes but no content:

`<point x="12" y="12"></point>`

and empty elements are often written with shorthand notation:

`<point x="12" y="12"/>`

In an XML file the order in which tagged elements appear and the ways in which they can be nested (contained one in another) is completely determined by the Document Type Definition for the file. The XML files that specify the parameters for a NewGarden simulation are of the `Simulation_Data` document type and this format is specified by the first lines of the sample file:

```
<!DOCTYPE Simulation_Data[
...
]>
```

The inclusion of these lines is not essential for the NewGarden program to parse the file and obtain the parameter values. The advantage of including the

`DOCTYPE` statement is that then it is possible to use a facility in the NewGarden program to ensure that a file is in the correct format and that it can be parsed unambiguously. This process of ensuring that the file meets the format requirements is referred to as *validation* and is accomplished with the command

```
java -jar NewGarden.jar Validator SimData.xml
```

If the parameter file named `SimData.xml` doesn't meet the format requirements, this command will display a list of the ways in which it is deficient. If no message results, the file is syntactically correct.

In a `Simulation_Data` type file, all the other tagged elements are contained in an enveloping element

```
<Simulation_data>
...
</Simulation_data>
```

and, after the optional DOCTYPE preface, this must be the first tagged element in the XML file.

Here we describe what other elements are allowed and how the parameters they specify are used by NewGarden's Model program to run a simulation. The order in which they are discussed is the order in which they appear in a valid `Simulation_data` element.

In reading or writing a `Simulation_data` it is useful to know that comments, which are ignored by programs are set off with the markers `<!--` at the start of the comment and `-->` at the end: so here is an XML comment:

```
<!-- THIS IS A COMMENT -->
```

5.2 LOCI

The `LOCI` element describes the number of independent loci to be used in the simulation and specifies the number of alleles that occur at each of these loci together with their frequencies in the virtual population from which founders (initial population members) will be selected.

```
<LOCI number_loci="2">
        <locus>
           <dpdpoint x="0.5" y="0"/>
           <dpdpoint x="0.5" y="1"/>
   </locus>
        <locus>
           <dpdpoint x="0.8" y="0"/>
           <dpdpoint x="0.1" y="1"/>
           <dpdpoint x="0.1" y="2"/>
        </locus>
</LOCI>
```

The attribute `number_loci` specifies the number of different loci to be used in the simulation and each `locus` element nested inside the `LOCI` element describes one of the loci by giving the labels for and frequencies of the alleles at the locus. This is accomplished with the `dpdpoint` elements nested within the `locus` element.

The elements `dpdpoint` in a single `locus` describe a *discrete probability distribution* by giving the values `y` and the probability of those values `x`. Technically the sum of all the probability values `x` specified in the `dpdpoint` elements of a single `locus` should be non negative and sum to 1 but the program checks that this is the case and renormalizes the `x`'s if necessary.

If you want to run a simulation that tracks a large number of similar loci you can specify that the program should generate these automatically rather than specifying them explicitly with `<locus>` elements. The following `<LOCI>` element calls for a simulation using 5 loci — two that are specified explicitly and 3 more that are generated automatically each with 4 equally probably alleles.

```
<LOCI number_loci="2"
      auto_alleles_per_locus="4"
      number_automatic_loci="3">
        <locus>
           <dpdpoint x="0.5" y="0"/>
           <dpdpoint x="0.5" y="1"/>
</locus>
        <locus>
           <dpdpoint x="0.8" y="0"/>
           <dpdpoint x="0.1" y="1"/>
           <dpdpoint x="0.1" y="2"/>
        </locus>
</LOCI>
```

Using the `auto_alleles_per_locus` and `number_automatic_loci` it is possible to generate loci that have the same number of equally likely alleles.

An alternative for specifying a large number of loci in a `Simulation_data` file is to use the copy and paste functions of a word processor to quickly generate many duplicate `<locus>...<\locus>` statements and paste these into the XML file. The CD includes some plain text files with plain text that can be used in this way.

5.3 Dioecious

The `value` attribute of this element determines whether the population that is simulated is dioecious or monoecious as in:

```
<Dioecious value="false"/>
```

In a simulation using a monoecious population the sexes of the populations members are ignored by the program.

5.4 Reproduction

The overall, raw, reproduction rate in the population and they way it depends on age is specified with the `Reproduction_Rate` element. This is accomplished by providing a list of points that lie on the graph of reproduction rate vs. time. An example is:

```
<Reproduction_Rate>
       <functionpoint x="0" y="4"/>
       <functionpoint x="1" y="4"/>
       <functionpoint x="5" y="0"/>
</Reproduction_Rate>
```

The `functionpoint` elements nested in the `Reproduction_Rate` specify the horizontal (`x`) and vertical (`y`) coordinates of points of the graph of the reproduction rate vs. age function. For ages that are not listed explicitly among the user-provided points the program interpolates linearly between points and extrapolates beyond all specified points using the value at the nearest point. That is, the reproductive rate function $y(x)$ specified by the above element is

$$y(x) = \begin{cases} 4 & \text{if } x \leq 1 \\ y = 4 - x & \text{if } 1 \leq x \leq 5 \\ 0 & \text{if } x \geq 5 \end{cases}$$

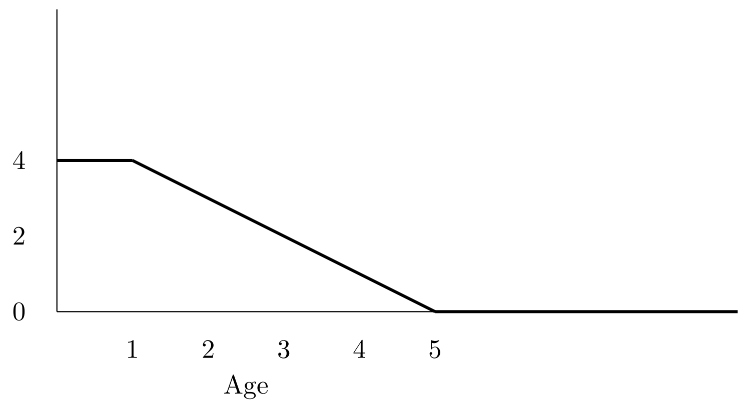


Figure 1: This shows a plot of the reproduction rate vs. age function defined by the preceding `Reproduction Rate` element.

Note that this “reproduction rate” is the number of offspring that are generated as potential recruits from a maternal parent but the actual number that enter the population can be smaller because potential recruits may land at a grid point that is already occupied or be dispersed out of the designated “preserve”.

5.5 Offspring distribution

The process used by the program to determine the number of potential offspring to generate for a given parent — based on the data given in the `Reproduction_Rate` as described above — can result in fractional numbers. Even if points with integer coordinates are supplied by the user, interpolation may result in fractional numbers. The `Offspring_Distribution` element specifies how these potentially fractional numbers can be converted into an integer that is the actual number of offspring to generate.

```
<Offspring_Distribution method="bracket"/>
```

The element has only one attribute called `method` and it can be assigned equal to one of `round`, `bracket` or `poisson`.

If the method specified is `round` then the actual number of offspring is obtained by rounding down to the nearest whole number.

If the method specified is `bracket` then the actual number of offspring is obtained from a fractional number x by rounding down or rounding up to the nearest whole number in such a way that the average value obtained remains x.

If the method specified is `poisson` then the actual number of offspring is obtained from a fractional number x by generating a random number from a Poisson distribution with mean x.

5.6 Pollen Rate

The parameters given in the `Pollen_Rate` element describe the relative chances of a plant (or male plant in a simulation of a dioecious population) acting as a pollen donor by specifying how these relative chances depend on the age of the plant.

```
<Pollen_Rate>
        <functionpoint x="0" y="0"/>
        <functionpoint x="1" y="0"/>
        <functionpoint x="2" y="1"/>
        <functionpoint x="7" y="1"/>
        <functionpoint x="8" y="2"/>
        <functionpoint x="10" y="2"/>
        <functionpoint x="11" y="2"/>
        <functionpoint x="20" y="1"/>
        <functionpoint x="40" y="0"/>
</Pollen_Rate>
```

The `<functionpoint>` elements specify points on the graph of the relative probability vs. age function. As usual, linear interpolation is used to obtain function values at intermediate x values and constant extrapolation from the nearest specified value is used for x values outside the range of x's specified by the user.

5.7 Mortality

The dependence of death rates on age is described in a similar way, but using the `Mortality_Rate` element:

```
 <Mortality_Rate>
        <functionpoint x="0" y="0"/>
        <functionpoint x="1" y="0"/>
        <functionpoint x="4" y="0.5"/>
        <functionpoint x="5" y="1"/>
</Mortaility_Rate>
```

5.8 Selfing

In monoecious populations the overall probability of self fertilization is set with the `value` attribute of the `Selfing_Rate` element:

```
<Selfing_Rate value="0.0"/>
```

This parameter has no effect if the population has been declared dioecious.

5.9 Random mating

For small populations in which the selfing rate has been specified at 0 the exclusion of self crosses can perturb Hardy-Weinberg equilibrium. By setting the `value` attribute of the `RandomMating` element to true, this perturbation is eliminated by allowing the simulation to include self crosses in such a way that every mating (including self matings) is equally likely.

```
<RandomMating value="false"/>
```

5.10 Dispersal of recruits

Potential recruits are assigned a location by picking a dispersal distance d at random from a user-specified distribution of distances and then selecting at random from among all the locations at distance d from the parent.

The distribution of dispersal distances is specified by giving points on the graph of the *cumulative distribution* function of the distances. As usual, this function is specified by using `functionpoint` elements to give points on its graph.

The first (left most) point must be
`<functionpoint x="0" y="0"/>`
so that only non-negative distances are selected.

Since the distribution of dispersal distances is given as a cumulative distribution, adding the point $(12, 0.5)$ to the graph of the distribution with the element
`<functionpoint x="12" y="0.5"/>`

says that the probability that the randomly chosen dispersal distance is ≤ 12 is 50%. This means that the chance is 50% that one of the values $0, 1, \ldots, 11$ will be selected. While 12 is a possible, its probability is 0 as is the probability of any specific fractional number x. We round down to the nearest whole number distance.

To make the chance of dispersing exactly 12 units equal to 0.1, the next entry should be

```
<functionpoint x="13" y="0.6"/>
```

since then the chance that the distance is between 12 and 13 is $0.6 - 0.5 = 0.1$ and when working with distances, which are always whole numbers, the program always *rounds down.*

If the maximum distance is to be D the last functionpoint should be

```
<functionpoint x="M" y="1.0"/>
```

where $M = D + 1$ since then some number between 0 and D is always selected when a random distance is generated.

```
<Dispersal_Distribution>
        <functionpoint x="0" y="0"/>
        <functionpoint x="1" y="0.1"/>
        <functionpoint x="2" y="0.3"/>
        <functionpoint x="3" y="0.7"/>
        <functionpoint x="6" y="1.0"/>
</Dispersal_Distribution>
```

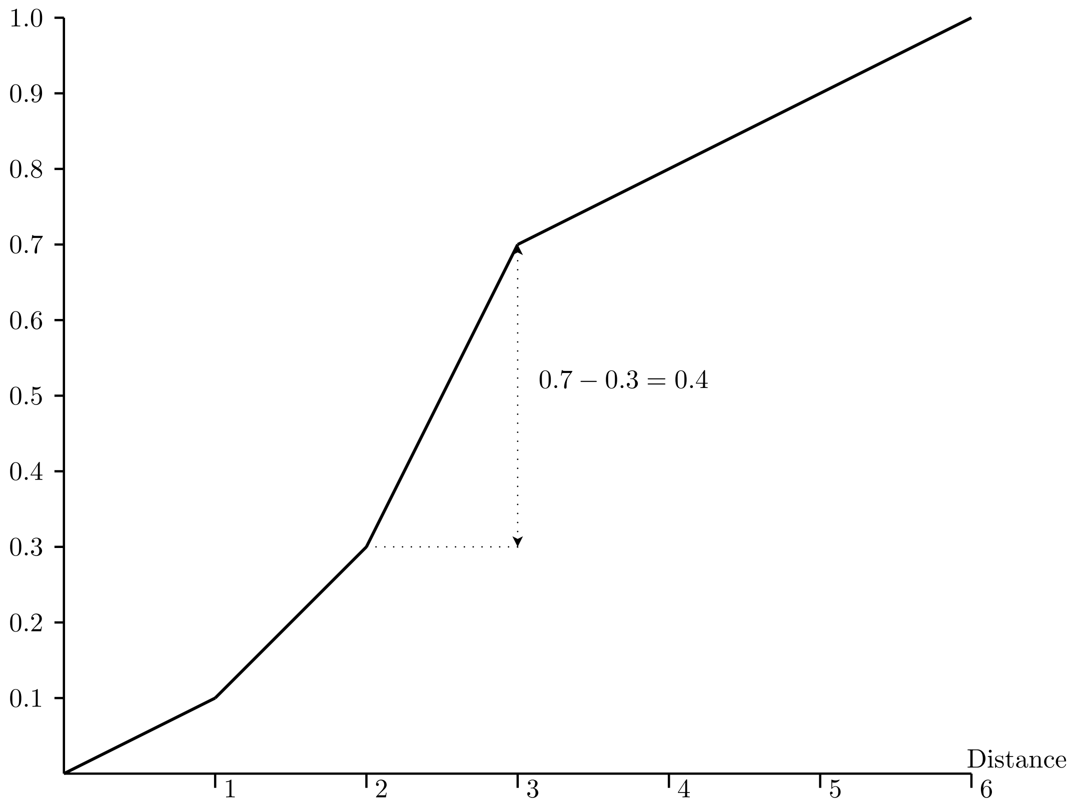


Figure 2: The cumulative distribution function for dispersal distances specified by the the lines. Since the cumulative probability increases 40% as the distance changes from 2 to 3 and since distances are rounded down to the next lowest whole number, this curve describes a probability distribution that makes the dispersal distance 2 with probability 0.4.

5.11 Pollen dispersal

The program selects pollen sources by first picking a range of distances at random according to a use-specified distribution and then selecting a pollen source at random from among those whose distance from the parent is in this range of distances.

```
<EasyPollen>
<pollenframe low="0" high ="2" prob="0.5"/>
    <pollenframe low="3" high ="6" prob="0.45"/>
    <pollenframe low="7" high ="Inf" prob="0.05"/>
</EasyPollen>
```

The parameters describing these distance classes and the probabilities with which they occur are adjusted with the `EasyPollen` element and the `pollenframe` elements included in it. Each `pollenframe` element describes one of the distance classes by giving the lowest and highest distances in the class and the probability with which this class is to be selected. These values are given as the `low`, `high`, and `prob` attributes of the `pollenframe` element.

The `EasyPollen` element above describes a simulation in which 50% of the time the pollen source is selected from those at a distance of $0, 1, 2$ from the female parent; 45% of the time from among the sources at distance between 3 and 6 and 5% of the time pollen comes from a source that is at distance 7 or more form the female parent.

The `pollenframe` elements for the distance classes must be mentioned in *increasing* distance and should describe ranges of distance that do not overlap.

While the purpose of these parameters is to let the user control the distribution of distances to pollen sources and thus simulate different strategies and mechanisms for pollination it should be noted that this mechanism does not always enforce the specified distribution of distances. As an extreme example consider the case in which an isolated female grows at distance 20 from all other plants and thus all possible pollen sources. If the user has specified that almost pollen must come from nearby sources, perhaps by stating

```
<EasyPollen>
<pollenframe low="0" high="5" prob="0.99"/>
<pollenframe low="6" high="Inf" prob="0.01"/>
</EasyPollen>
```

Then there is only a 1% chance of successful pollination for the isolated female.

If, in the process of selecting a pollen source for a mating it happens that there are no population members (or no males) in the randomly generated distance range pollination fails and the number of potential recruits generated by the female in that round of breeding is reduced below the "raw" value specified.

5.12 Regions

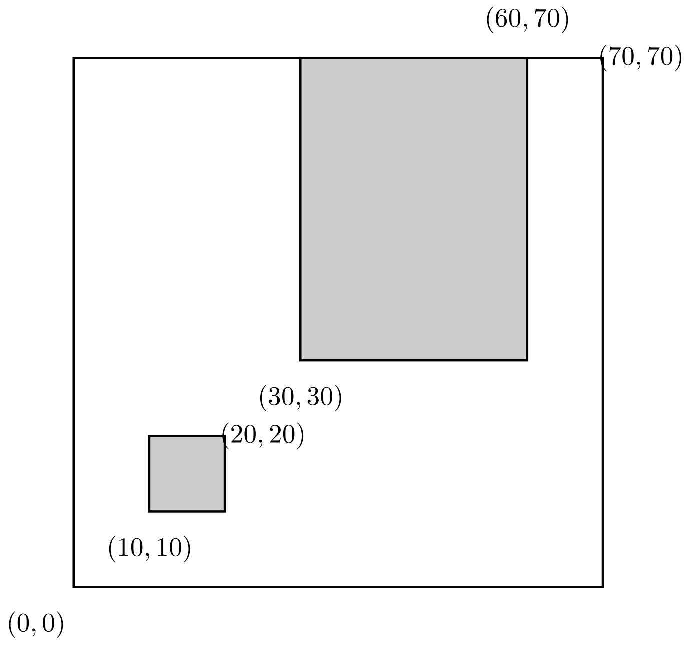
The entire simulated population resides in a user-specified region and this region is assumed to live inside a rectangular array of lattice points. If no other restriction on the viable locations are required for the simulation, a `Region` element with attributes mentioning the coordinates of the lower left (XL,YL) and upper right (XH,YH) corner points is all that is needed:

```
<Region XL="0" XH="70" YH="70" YL="0">
 </Region>
```

More general shapes for the region (the grid points that are allowed to be occupied by a population member) used in a simulation can be defined by specifying just the subset of the lattice points in the containing rectangular array that are in the region. This subset may be represented as the collection of all points belonging to one or more rectangles with sides parallel to the sides of the containing region or as belonging to one or more convex polygons that have their vertices at lattice points.

The rectangles are specified by giving the coordinates of their lower left and upper right corners and the convex polygons are specified by giving the coordinates of their corners, reported in (say) clockwise or counterclockwise order. A more general region than the rectangle described above and having, for example two (disconnected) parts might be

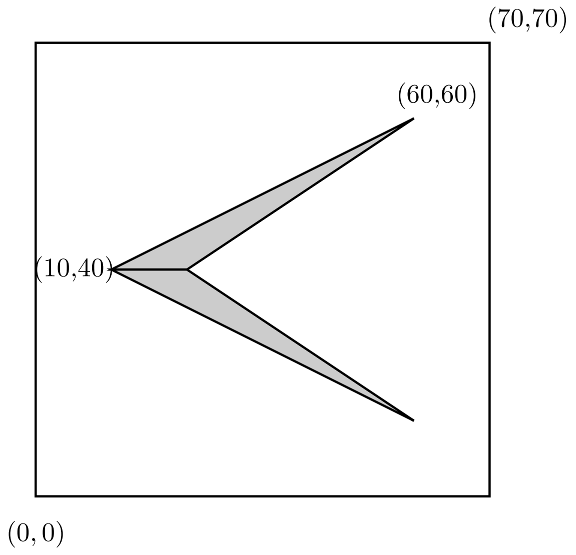
```
<Region XL="0" XH="70" YH="70" YL="0">
       <Rectangle XL="10" YL="10" XH="20" YH="20"/>
       <Rectangle XL="30" YL="30" XH="60" YH="70"/>
   </Region>
```



And an even more complicated region is described by the elements:

```
<Region XL="0" XH="70" YH="70" YL="0">
       <ConvexPolygon>
           <Vertex x="20" y="40"/>
           <Vertex x="30" y="40"/>
           <Vertex x="60" y="60"/>
       </ConvexPolygon>
        <ConvexPolygon>
           <Vertex x="20" y="40"/>
           <Vertex x="30" y="40"/>
```

```
            <Vertex x="60" y="20"/>
        </ConvexPolygon>
    </Region>
```



A point is considered to lie in a region if it is both inside the large rectangle specified by the attributes of the `Region` element and in one or more of the `Rectangle`s or `ConvexPolygon`s listed inside the `Region` element.

Again, if no rectangles of convex polygons are mentioned in the description of a region points are either in or out of the region according to whether they belong to the bounding rectangle specified in the `Region` element itself.

5.13 Summary regions

As mentioned already, the program tabulates information about the population after each episode of reproduction and mortality, saving enough data to print summaries of the number of alleles, heterozygosity, etc. for the entire population and for the age class generated by the most recent round of reproduction. By default these summaries include all members of the population and of the cohort. By specifying `SummaryRegions`'s you can cause the program to also calculate these summaries for that population located in a particular region or regions.

The following example shows how the regions are specified: `Rectangle` and `ConvexPolygon` elements are placed in `Region` elements, just as is done to define the total region for the simulation; and these `Region` elements are listed consecutively in a `SummaryRegions` element. Data tabulated for the different summary regions is presented in the output in the same order that the regions are defined in the `SummaryRegions` element.

```
<SummaryRegions>
    <Region XL="0" XH="70" YH="70" YL="0">
       <Rectangle XL="30" YL="30" XH="60" YH="70"/>
    </Region>
     <Region XL="0" XH="70" YH="70" YL="0">
      <ConvexPolygon>
            <Vertex x="10" y="10"/>
            <Vertex x="10" y="20"/>
            <Vertex x="15" y="20"/>
        </ConvexPolygon>
        <Rectangle XL="4" XH="55" YL="0" YH="8"/>
    </Region>
</SummaryRegions>
```

5.14 Number Generations

The number of generations — rounds of reproduction and mortality — through which the simulations are carried is controlled with the `value` attribute of the `Number_Generations` element, as in:

```
<Number_Generations value="4"/>
```

5.15 Number Runs

the `value` attribute of the `Number_Runs` element controls the number of independent simulations the program runs before calculating and presenting summary statistics.

```
 <Number_Runs value="4"/>
```

5.16 Initial Population

At the start of each run of the simulation the population is established by introducing a number of founders as specified by the user. The members of this founding population are specified in the `Initial_Population` element; each individual is specified with a `Plant` element that has attributes giving the `age` of the individual at the start of the simulation, the location of the individual (the `X` and `Y` attributes are the coordinates on the grid), and the sex of the individual, specified as a `true` or `false` value for the attribute `femaleP`. The sex attribute is ignored unless the population has been specified as dioecious.

An initial population element specifying two founding members is the something like:

```
 <Initial_Population>
        <Plant age="0" X="12" Y="12" femaleP ="true"/>
        <Plant age="0" X="15" Y="15" femaleP ="false"/>
 </Initial_Population>
```

For simulations that start with a large number of founders the number of `Plant` elements can be quite large. We suggest using a small script or program (`founders.py` is an example of a Python script that is distributed with NEWGARDEN) to generate the text for these elements and/or saving "libraries" of `Plant` elements that can be edited and pasted in to other data files. A variety of these snippets of plain text are also provided on the CD.

5.17 Founders F

As the founders are selected at the start of a simulation run their genotypes are selected at random as if they were drawn from a population with gene frequencies as specified in the `<LOCI>` element. Further, it is assumed that the inbreeding coefficient, F, for the population from which the founders are drawn is $F = 0$ unless a different value is specified using the `value` attribute in the `<FoundersF>` element as in this example:

```
<FoundersF value="0.5"/>
```

6 Sources, copyrights etc.

All of the source code for the programs in the NEWGARDEN distribution are actually included with the distribution.

1. Most of the source code for these programs was written by Pelikan and Rogstad. These files are labelled as such and released under the GNU Public Licence (i.e. they're "GPL'd "). A copy of this licence is included in the distribution.

2. One program uses code by `wolfgang.hoschek@cern.ch` for the evaluation of special functions. This code lives in the source file Gamma.java. The permission and copyright notice for this source code is:

   As requested, the above notice appears here and is included in the file Gamma.java as well.

3. The portion of the code that computes lifetables a la Leslie uses code for manipulation of matrices and for finding eigenvectors that is based on java sources originally created by Bruce R Miller at NIST. He shared his code with me before it was ever developed as a package and distributed, I believe. It is now part of the *Java Matrix Package* freely available at `http://math.nist.gov/javanumerics/jama/`

   I modified Bruce's code a little so what's included here may not be recognizable as originating with him except there's a comment at the start of some files mentioning him.

4. The getopt package that is used to parse command line options is available under the GNU Library General Public License. Here are the details from the file Getopt.java:

```
/**************************************************************************
/* Getopt.java -- Java port of GNU getopt from glibc 2.0.6
/*
/* Copyright (c) 1987-1997 Free Software Foundation, Inc.
/* Java Port Copyright (c) 1998 by Aaron M. Renn (arenn@urbanophile.com)
/*
/* This program is free software; you can redistribute it and/or modify
/* it under the terms of the GNU Library General Public License as published
/* by  the Free Software Foundation; either version 2 of the License or
/* (at your option) any later version.
/*
/* This program is distributed in the hope that it will be useful, but
/* WITHOUT ANY WARRANTY; without even the implied warranty of
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
/* GNU Library General Public License for more details.
/*
/* You should have received a copy of the GNU Library General Public License
/* along with this program; see the file COPYING.LIB.  If not, write to
/* the Free Software Foundation Inc., 59 Temple Place - Suite 330,
/* Boston, MA  02111-1307 USA
/**************************************************************************/
```