Developing tests for verification of the *NAADSM* modeling framework

Anthony “Drew” Schwickerath
Animal Population Health Institute
College of Veterinary Medicine & Biomedical Sciences
Colorado State University
Fort Collins, Colorado 80523

For correspondence, please contact:
Neil Harvey <neilharvey@gmail.com>

NAADSM Technical Paper #3
Version 2009/08/14
http://www.naadsm.org/techpapers/

Document revision history

- 2009/08/14 –Initial public version

Introduction

One of the key characteristics of NAADSM is that it has been verified against the model specification. Part of this verification is the included unit and system test suite. It is crucial that this test suite is kept up to date. This document should help you, the developer, add new automated tests.

Running tests is as simple as typing `make -i check1` in either the `sim/` directory (to run all of the tests) or in one of the subdirectories (to run the tests for that module).

Writing new tests is somewhat less simple. It involves having at least a passing familiarity with DejaGnu (which means having a working knowledge of Expect), lex, and yacc. It also means understanding the way that each module is tested within this framework.

DejaGnu

DejaGnu is an open source automated test system. It is written in the Expect scripting language, which is discussed in a later section. For NAADSM, a DejaGnu test system consists of the test harness and a set of test cases. The test harness is often a C program (possibly generated using lex and yacc) that allows us to exercise the module under test. The test cases are generally written in Expect.

For more detail than is provided below, see Rob Savoye, *DejaGnu:The Gnu Testing Framework*.

Test Output

When DejaGnu runs, it generates files with different degrees of information. The highest level is the `.sum` file (eg, `models/models.sum`). This contains the PASS/FAIL status of each test, as well as any informative messages that the test case outputs. This is what you will normally look at, when you run the tests.

Kicking the detail up a notch is the `.log` file (eg, `models/models.log`). This contains all of the text sent to the test harness, as well as the text that the test harness output back to your Expect script. Usually this is too much detail, but when tracking down a defect in either the module under test or the test suite itself, an inspection of this file can quickly show the cause of a failure.

If this still is not enough detail, DejaGnu can be run with the `--debug` option to produce the `dbg.log` file. This file contains detailed information on the execution of the Expect code. Since the NAADSM test suite is relatively mature, this level of information will probably not be necessary.

Input and Output

At the beginning of a set of tests, the test program is executed. Rather than the user typing commands into it and reading the output, though, the DejaGnu/Expect script handles both. The test program's standard input and output (`stdin`, `stdout` in C) are redirected to the script.

The Expect scripts implement the test cases. For each test case, it sends commands to the test shell. In general, after sending each command it reads, parses, and compares the response with the expected

---

1 The -i option keeps make from terminating when a test fails, as there are always some of the statistical tests that fail and whose results will need to be evaluated by hand.

result. For some commands, there is no reply, so there is nothing to read. If the command does produce a reply, though, the script must read it.

Expect

DejaGnu is written in Expect, an extension of Tcl. Expect is designed to interact with other programs, providing input to the program under control and receiving (and processing/interpreting) the output that is generated in response.

To get started modifying existing test cases or even writing your own, you do not need to be a Tcl/Expect expert. Almost everything that you should need to know is obvious from looking through a few of the existing test files. Because regular expressions are central to the way Expect is used, you will want to familiarize yourself with those, as well.

For more detail on Expect and regular expressions, see Don Libes, *Exploring Expect*.

Test Shells

Some of the tests make use of a shell to drive the library under test. These shells are written using lex and yacc. A definitive treatment of these tools is beyond the scope of this document. Most of the work that you will need to do will involve extending existing files rather than starting from scratch. As a consequence, you will have examples within the working code to guide you. Below we provide a little background information on what these tools do and references to find more information, should you need more detail, though, to get you started.

lex

Lex produces a lexical analyzer based upon a specification file (`.l`). A lexical analyzer reads a string of characters and produces a sequence of tokens. For example, a lexical analyzer might read "`3 + 4`" and produce "`NUM PLUS NUM`".

For information on lex, see either Tony Mason and Doug Brown, *Lex and Yacc* or John Levine, *Flex & Bison*.

yacc

Yacc produces a parser based upon a specification file (`.y`). The parser yacc generates takes the string of tokens produced by a lexical analyzer, groups them according to rules and performs operations on them. For the purpose of writing tests for NAADSM, the parser interprets the commands that DejaGnu/Expect sends in and makes calls into the NAADSM unit under test with the appropriate parameters. The code block within a rule is written in C.

For information on yacc, see either Tony Mason and Doug Brown, *Lex and Yacc* or John Levine, *Flex & Bison*.

NAADSM Test Suite

NAADSM is constructed of a number of modules. Each of these modules is contained in a subdirectory in the source tree and has its own test harness and test cases. A module's tests are contained in its `test/` directory. If there are any convenience functions or regular expressions, as is

the case for `prob_dist/`, these are contained in the module's `test/config/unix.exp` file.

gis/

The `gis/` tests verify the GIS distance and polygon operations. The tests are written in Expect and the test shell is constructed using lex and yacc. The Expect files in `gis/test/libgis.all/` send commands to the test shell and check the results against the expected values. The test shell commands are

`great circle distance (`***lat1***`, `***lon1***`, `***lat2***`, `***lon2***`)`
: Returns the great-circle distance in km between two points. The arguments are given in degrees.

`distance (`***x1***`, `***y1***`, `***x2***`, `***y2***`)`
: Returns the distance in km between two points. The arguments are given on a km grid.

`great circle heading (`***lat1***`, `***lon1***`, `***lat2***`, `***lon2***`)`
: Returns the initial heading in degrees from point 1 to point 2. The arguments are given in degrees.

`heading (`***x1***`, `***y1***`, `***x2***`, `***y2***`)`
: Returns the heading in degrees from point 1 to point 2. The arguments are given on a km grid.

`area (`***x1***`,`***y1***`,`***x2***`,`***y2***`,...)`
: Returns the area of a polygon. The initial point does not have to be repeated as the final point. Inputs with zero, one, or two points are allowed, in which case the area should be 0.

`perimeter (`***x1***`,`***y1***`,`***x2***`,`***y2***`,...)`
: Returns the perimeter of a polygon. The initial point does not have to be repeated as the final point. Inputs with zero, one, or two points are allowed, in which case the perimeter should be 0.

herd/

The `herd/` tests verify the basic behavior (disease progression, etc) of herds. More detailed tests are found in `models/`. The tests are written in Expect and the test shell is constructed using lex and yacc. The Expect files in herd/test/libherd.all/ send commands to the test shell and check the results against the expected values. The test shell commands are

`herd (`***type***`,`***size***`,`***lat***`,`***long***`)`
: Adds a herd to the test set. The first argument may be any string, e.g., `"Beef Cattle"`, `"Swine"`. The herds are numbered starting from 0. They start as `Susceptible`.

`infect (`***herd***`,`***latent***`,`***infectious_subclinical***`,`***infectious_clinical***`,`***immunity***`)`
: Starts the progression of the disease in the given herd. The next four arguments (all integers) give the number of days the herd spends in each state.

`vaccinate (`***herd***`,`***delay***`,`***immunity***`)`
: Vaccinates the given herd. The argument ***delay*** gives the number of days the vaccine requires to take effect. The

	animals are immune to the disease for the number of days given by the argument ***immunity***.
`destroy (`***herd***`)`	Destroys the given herd.
`step`	Step forward one day. The output of this command is a space-separated list of the herd statuses.
`reset`	Erase all currently entered herds.

models/

Unlike the other tests, the `models/` tests consist of set of XML files rather than of Expect files. The tests for each model or group of models are located in a separate directory, `models/test/model.*/`.

Each of the `models/test/model.*/` directories contains an `all.xml` file, which describes all of the tests for this model. When `make` is run, `models/test/model.*/all.expect` is generated. Whenever the description of a test is added, removed, or changed in `all.xml`, `make` should be rerun.

Each test within `all.xml` can be one of four types: deterministic, variable, stochastic, or stochastic variable. Deterministic tests always have the same outcome and their output is the state of each herd at each day. This is accomplished by forcing all random draws to return 0.5. Stochastic tests are run repeatedly with varying rather than fixed random draws. The possible runs (state of each herd for each day) are specified, along with the expected proportion of the tests runs that will have that output. Variable and stochastic variable tests allow different output reporting variables to be monitored rather than the herd states.

Before digging into the specifics of these tests, let us look at their commonalities. First, their specification in `all.xml` contain the same fields.

`category`	This matches the model directory and will be the same for all of the tests in this `all.xml` file. This is a string. Generally this does not contain whitespace or other special characters.
`short-name`	A short description of this test. This is what follows PASS or FAIL in DejaGnu's output (ie, `model.sum`). This is a string. Any special characters should be given by reference name (ie, `<` rather than <).
`description`	A description of the test's purpose, generally in a few sentences. This is a string. Any special characters should be given by reference name. This is used in the documentation generated by doxygen.
`author`	The name and email address of the test's author. This is a string. Any special characters should be given by reference name.
`creation-date`	The date that the test was originally written on.
`model-version`	The NAADSM software version that this test was

introduced in, not the model specification version number.

`herd-file` — This is the name of the herd file that will be used for this test, without the `.xml` extension. The herd files are located in `models/test/`, not in `models/test/model.*/`.

`parameter-file` — This is the name of the scenario file that will be used for this test, without the `.xml` extension. The scenario files are located in `models/test/model.*/`.

`parameter-description` — This is a detailed description of the parameters. Usually a description of the progression of the disease or control policy through the herds is given on a day-by-day basis, as it applies to this test. This is used in the documentation generated by doxygen.

`output` — This is the output that is being matched against. The exact contents depend upon the type of test that is being specified. In general, this is a table made up of rows (`<tr>`) and data elements (`<td>`).

```
<herds>
  <herd>
    <id>0</id>
    <production-type>Beef Cattle</production-type>
    <size>25</size>
    <location>
      <latitude>0</latitude>
      <longitude>0</longitude>
    </location>
    <status>Latent</status>
  </herd>
  <herd>
    <id>1</id>
    <production-type>Beef Cattle</production-type>
    <size>25</size>
    <location>
      <latitude>0.0898315</latitude>
      <longitude>0</longitude>
    </location>
    <status>Susceptible</status>
  </herd>
</herds>
```

Table 1: An example herd file. (`models/test/2herd.xml`)

The herd file is also consistent among all of the test types. The sample herd file in Table 1 shows two premises. Both premises have a production type of ``Beef Cattle"[2] and each contains 25 animals. The first one (id 0) is Latent (already infected but not yet shedding). The second one (id 1) is Susceptible (not yet infected). Coordinates are given in latitude and longitude.[3]

2 New versions of the software also allow the production types to be specified in the herd file.

3 In a future version, a different spacial reference and projection system can be specified.

```
<?xml version="1.0" encoding="UTF-8"?>
<naadsm:disease-simulation
  xmlns:naadsm="http://www.naadsm.org/schema"
  xmlns:xdf="http://xml.gsfc.nasa.gov/XDF">
  <description>Specs for a sample simulation run.</description>
  <num-days>10</num-days>
  <num-runs>1</num-runs>

  <models>

    <disease-model production-type="Beef Cattle">
      <latent-period>
        <point>1</point>
        <units><xdf:unit>day</xdf:unit></units>
      </latent-period>
      <infectious-subclinical-period>
        <point>0</point>
        <units><xdf:unit>day</xdf:unit></units>
      </infectious-subclinical-period>
      <infectious-clinical-period>
        <point>2</point>
        <units><xdf:unit>day</xdf:unit></units>
      </infectious-clinical-period>
      <immunity-period>
        <point>1</point>
        <units><xdf:unit>day</xdf:unit></units>
      </immunity-period>
    </disease-model>

    <conflict-resolver></conflict-resolver>

  </models>

  <output>
    <variable-name>all-units-states</variable-name>
    <frequency>daily</frequency>
  </output>

</naadsm:disease-simulation>
```

Table 2: An example scenario file. (`models/test/model.disease/disease_1.xml`)

The scenario file is much more complex than the herd file. The scenario file is made up of parameter definitions for each of the models/modules and the exact order of these model blocks is important. For this reason, it is often best to either generate one using NAADSM PC or to use an existing one as a starting point.

The scenario file also specifies the outputs that the test run will generate. These must match the test description in `all.xml`. The frequency should be "daily". The output parameters can either be specified in the output section of generating model or monitor module or in the global output section. The effect is the same.

Table 2 has a simple scenario. It consists of only two modules: a single instance of the disease-model and the conflict-resolver. The conflict-resolver should always be the last module specified in the models block.

This scenario would be appropriate for use in a deterministic or a stochastic test, since the only output specified is all-units-states on a daily basis. This is the only output that should be specified for deterministic or stochastic (not variable or stochastic-variable) tests.

```
<deterministic-test>
  <category>disease</category>
  <short-name>fixed # days for disease stages</short-name>

  <description>
    Test a scenario with fixed durations for each stage of the disease.
  </description>

  <author>Neil Harvey &lt;neilharvey@gmail.com&gt;</author>
  <creation-date>25 November 2003</creation-date>
  <model-version>3.0</model-version>

  <herd-file>1herd</herd-file>

  <parameter-file>disease_1</parameter-file>

  <parameter-description>
    The latent period lasts 1 day, infectiousness lasts 2 days, clinical signs
    accompany infectiousness immediately, and immunity lasts 1 day.

    There is no spread, detection, vaccination, or destruction.
  </parameter-description>

  <output>
    <tr><td>Latent</td></tr>
    <tr><td>InfectiousClinical</td></tr>
    <tr><td>InfectiousClinical</td></tr>
    <tr><td>NaturallyImmune</td></tr>
<!--
    No infectious units, simulation stops.
-->
  </output>
</deterministic-test>
```

Table 3: An example deterministic test definition. (from `models/test/model.disease/all.xml`)

Table 3 shows one of a few possible test definitions using the scenario file in Table 2. `models/test/model.disease/all.xml` pairs this same scenario file with different herd files to test different conditions. This particular test definition expects to take a single herd through the Latent (day 1), InfectiousClinical (days 2 and 3), and NaturallyImmune (day 4) states, as can be seen in the output section.

Notice that the output section is a table. Each row (`<tr>`) represents one day and each column (`<td>`)contains the state of a single herd.

```
<stochastic-test>
  <category>direct-contact-spread</category>
  <short-name>2 possible targets, same sizes</short-name>

  <description>
    Test the choice of a target for infection when two units are the same
    distance from the source unit.  Here the targets are the same size, so they
    should be chosen with equal probability.

    NB: This test case does not care about secondary spread, so we end the test
    before that can happen.
  </description>

  <author>Neil Harvey &lt;neilharvey@gmail.com&gt;</author>
  <creation-date>17 February 2004</creation-date>
  <model-version>3.0</model-version>

  <herd-file>2targets_same_dist_same_size</herd-file>

  <parameter-file>spread_2</parameter-file>

  <parameter-description>
    The latent period lasts 1 day, infectiousness lasts 2 days, clinical signs
    accompany infectiousness immediately, and immunity lasts 3 days.

    The movement rate is fixed at 1 shipment per day.  The distance of
    shipments is fixed at 10 km.

    Latent units can be the source of an exposure.

    The probability of infection given exposure is 1.

    There is no detection, vaccination, or destruction.
  </parameter-description>

  <output probability="0.5">
    <tr><td>Latent</td>              <td>Susceptible</td>  <td>Susceptible</td></tr>
    <tr><td>InfectiousClinical</td>  <td>Latent</td>       <td>Susceptible</td></tr>
  </output>

  <output probability="0.5">
    <tr><td>Latent</td>              <td>Susceptible</td>  <td>Susceptible</td></tr>
    <tr><td>InfectiousClinical</td>  <td>Susceptible</td>  <td>Latent</td></tr>
  </output>
</stochastic-test>
```

Table 4: An example stochastic test definition. (from models/test/model.direct-contact-spread/all.xml)

Stochastic tests are slightly more complicated than deterministic tests. The parts of NAADSM that are specified with probability density functions are randomly determined rather than being locked to 0.5 as they are in the deterministic tests. As a consequence, different sequences of herd states and herd interactions are possible for a given scenario. When running stochastic tests, rather than being run a single time as is done with the deterministic case, they are run a number of times. The frequency of each output combination is computed from these test runs and compared against the output frequencies specified in `all.xml`.

A simple example of a stochastic test is shown in Table 4. This test starts with a single Latent and two Susceptible herds. On the second day, one of the two uninfected herds will become infected by the first herd, and there is an equal probability that any given one of those will be selected. The two `<output`

`probability="0.5">` blocks reflect this and enumerate the two possible sequences of states.

```
<?xml version="1.0" encoding="UTF-8"?>
<naadsm:disease-simulation
  xmlns:naadsm="http://www.naadsm.org/schema"
  xmlns:xdf="http://xml.gsfc.nasa.gov/XDF">
  <description>Specs for a sample simulation run.</description>
  <num-days>10</num-days>
  <num-runs>1</num-runs>

  <models>

    <disease-model production-type="Beef Cattle,Pigs">
      <latent-period>
        <point>1</point>
        <units><xdf:unit>day</xdf:unit></units>
      </latent-period>
      <infectious-subclinical-period>
        <point>0</point>
        <units><xdf:unit>day</xdf:unit></units>
      </infectious-subclinical-period>
      <infectious-clinical-period>
        <point>2</point>
        <units><xdf:unit>day</xdf:unit></units>
      </infectious-clinical-period>
      <immunity-period>
        <point>1</point>
        <units><xdf:unit>day</xdf:unit></units>
      </immunity-period>
    </disease-model>

    <vaccine-model production-type="Beef Cattle,Pigs">
      <delay>
        <value>0</value>
        <units><xdf:unit>day</xdf:unit></units>
      </delay>
      <immunity-period>
        <point>3</point>
        <units><xdf:unit>day</xdf:unit></units>
      </immunity-period>
    </vaccine-model>

    <destruction-monitor>
      <output>
        <variable-name>desnU</variable-name>
        <broken-down>yes</broken-down>
        <frequency>daily</frequency>
      </output>
    </destruction-monitor>

    <conflict-resolver></conflict-resolver>

  </models>

</naadsm:disease-simulation>
```

Table 5: An example variable scenario file. (models/test/model.direct-contact-spread/disease_2species_same_params_w_vars.xml)

```
<variable-test>
  <category>disease</category>
  <short-name>override initial state, destroyed, check reporting variables</short-name>

  <description>
    Test a scenario where the starting disease state of a herd is given as
    Destroyed in the herd file.  This test also includes one infected herd so
    that the simulation doesn't early-exit right away.
  </description>

  <author>Neil Harvey &lt;neilharvey@gmail.com&gt;</author>
  <creation-date>29 June 2009</creation-date>
  <model-version>3.0</model-version>

  <herd-file>2herds_1_start_destroyed</herd-file>

  <parameter-file>disease_2species_same_params_w_vars</parameter-file>

  <parameter-description>
    The parameters are identical to "override initial state, destroyed" above.
  </parameter-description>

  <output>
    <tr>
      <td>desnUAll</td>
      <td>desnUIni</td>
      <td>desnUBeefCattle</td>
      <td>desnUPigs</td>
      <td>desnUIniBeefCattle</td>
      <td>desnUIniPigs</td>
    </tr>
    <!--                                                  Initially/  Initially/  -->
    <!--  Total     Initially     Cattle       Pigs       Cattle       Pigs     -->
    <tr><td>1</td>  <td>1</td>  <td>0</td>  <td>1</td>  <td>0</td>  <td>1</td></tr>
    <tr><td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td></tr>
    <tr><td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td></tr>
    <tr><td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td>  <td>0</td></tr>
  </output>
</variable-test>
```

Table 6: An example variable test definition. (from `models/test/model.disease/all.xml`)

There are situations where we want to make sure that other aspects of the model, not just the disease states of the herds. Some of these variables give us a window into the inner working of control methods. Others are useful in determining the amount of damage caused by an outbreak. The (deterministic) variable and stochastic-variable tests verify that these outputs are computed and reported correctly.

Table 5 shows a simple scenario. Notice that the output no longer contains all-units-states. Instead, the destruction-monitor outputs the desnU variables. broken-down is yes, so all of the related desnU variables are produced.

The other half of the test, the (deterministic) variable test description from `all.xml`, can be found in Table 6. For a variable test, the rows are still days, but now the columns are variables rather than herds. An extra row at the top of the output table lists the variable names. Notice that, while we only listed `desnU` in the scenario file, we specified that all of the pieces would be broken out. As a consequence, we need to list `desnUAll`, `desnUIni`, `desnUBeefCattle` and `desnUPigs` (the two production types specified in this scenario), and `desnUIniBeefCattle` and `desnUIniPigs`.

```
<stochastic-variable-test>
  <category>ring-vaccination</category>
  <short-name>
    infection vs. vaccination on same day, check reporting variables
  </short-name>

  <description>
    Test the reporting of the number of new infections when infection and
    vaccination (with immediate effect) happen on the same day.  If the
    vaccination happens "first", the infection should not be recorded.
  </description>

  <author>Neil Harvey &lt;neilharvey@gmail.com&gt;</author>
  <creation-date>10 September 2007</creation-date>
  <model-version>3.0</model-version>

  <herd-file>2herds</herd-file>

  <parameter-file>spread_radius_10_w_vars</parameter-file>

  <parameter-description>
    What should happen:
     - On day 1 unit 0 will be Latent.
     - On day 2 unit 0 will be Infectious Clinical.  It will ship to unit 1.
       Unit 0 will be detected, and unit 1 will be marked for vaccination.
     - On day 3 the shipment will arrive at unit 1 and unit 1 will be
       vaccinated.  Half of the time the vaccination will happen "first" and
       the other half of the time the infection will happen "first".
     - On day 4 unit 1 will be either Latent or Vaccine Immune.
  </parameter-description>

  <!-- probability that unit becomes Vaccine Immune = probability that the
    vaccination happens "before" the infection (0.5). -->
  <output probability="0.5">
    <tr>
      <td>vacnUAll</td>
      <td>expnUAll</td>
      <td>infnUAll</td>
    </tr>
    <tr><td>0</td>  <td>0</td>  <td>1</td></tr>
    <tr><td>0</td>  <td>0</td>  <td>0</td></tr>
    <tr><td>1</td>  <td>1</td>  <td>0</td></tr>
  </output>

  <!-- probability that unit becomes Latent = probability that the infection
    happens "before" the vaccination (0.5). -->
  <output probability="0.5">
    <tr>
      <td>vacnUAll</td>
      <td>expnUAll</td>
      <td>infnUAll</td>
    </tr>
    <tr><td>0</td>  <td>0</td>  <td>1</td></tr>
    <tr><td>0</td>  <td>0</td>  <td>0</td></tr>
    <tr><td>1</td>  <td>1</td>  <td>1</td></tr>
  </output>
</stochastic-variable-test>
```

Table 7: An example stochastic variable test definition. (from `models/test/model.ring-vaccination/all.xml`)

Stochastic variable tests are the obvious marriage of the stochastic tests and variable tests described above. Multiple output sections, each with a probability of occurring, are given. These each contain a

table with a row for each day (plus one for variable headings) and a column for each variable specified in the scenario file. A simple example is provided in Table 7.

prob_dist/

NAADSM relies heavily on a number of probability distributions. The probability distribution tests check the distribution of random draws, range, mean, probability density function (PDF), and cumulative density function (CDF) for each distribution.

Each distribution has a corresponding `prob_dist/test/libprob_dist.all/*.exp` file containing all of the test cases. These files are written in TCL/Expect. They make use of functions defined in `prob_dist/test/config/unix.exp` for performing the standard tests.

`pdf_test` ***xvalues pvalues*** — Test `PDF_pdf()` for the distribution under test by passing in each element of ***xvalues*** and comparing it with the corresponding element of ***pvalues***.

`cdf_test` ***xvalues areavalues*** — Test `PDF_cdf()` for the distribution under test by passing each element of ***xvalues*** and comparing it with the corresponding element of ***areavalues***.

`random_number_test` ***true_answer histogram_low_bound*** — Test `PDF_random()` by randomly drawing 1000000 samples (by default, though this can be changed with the optimal ***iterations*** parameter) and generating a histogram. The first bin is has an x value of ***histogram_low_bound*** and the number of bins is the number of elements in ***true_answer***, which is the expected frequency for that bin. The bin width is 1.

`statistics_test` ***min max mean variance*** — Test `PDF_min()`, `PDF_max()`, `PDF_mean()`, and `PDF_variance()` for this distribution by comparing the results with ***min***, ***max***, ***mean***, and ***variance***, respectively. Because `fcmp` does not deal well with comparing zero (0) to near-zero values, these tests may fail for some distributions. If a given value is undefined for this distribution (`PDF_has_min()`, for instance, returns `FALSE`), then that statistic is not tested.

`sample_statistics_test` — Test `PDF_min()`, `PDF_max()`, `PDF_mean()`, and `PDF_variance()` for this distribution by drawing 1000000 samples (by default, though this can be changed with the optional ***iterations*** parameter) and calculating the min, ma, mean, and variance for the data. This verifies that the statistical functions match the distribution of random numbers generated. If a given value is undefined for this distribution (`PDF_has_min()`, for instance, returns `FALSE`), then that statistic is not tested.

To test a new probability distribution, you first need to update the test shell. Add the distribution's

name to `prob_dist/test/scanner.l` using a form similar to that given for `gaussian`. After that, add the probability distribution's name (matching what you added to `scanner.l`), syntax, and creation function to `prob_dist/test/shell.y`. To do this, you will need to add the name to one of the `%token` lines. Then add a syntax line with a `NUM` entry for each numeric parameter that must be given to specify the new distribution. Search for `GAUSSIAN` for an example of all of the pieces that are necessary.

Once you have updated the test shell, you need to add a new Expect file to `prob_dist/test/libprob_dist.all` for your distribution. `prob_dist/test/libprob_dist.all/point.exp` is a good file to base a new test on. You will need to rename and modify the `create_point_pdf` to match the name and parameters of your new distribution. Also update the call to the `create_*_pdf` function to pass the appropriate parameters for your test case distributions. Update the `xvalues` and `answers` to reflect the the PDF and CDF.

rel_chart/

The `rel_chart/` tests verify the relationship chart data structure. The tests are written in Expect and the test shell is constructed using lex and yacc. The Expect files in `rel_chart/test/librel_chart.all/` send commands to the test shell and check the results against the expected values. The test shell commands are

`chart (`***x1***`,`***y1***`,`***x2***`,`***y2***`,...)`	Creates a relationship chart.
`lookup (`***x***`)`	Returns the ***y***-value for a given ***x***-value from the most recently created relationship chart.
`range`	Returns the lowest and highest ***y***-values in the most recently created relationship chart.

reporting/

The `reporting/` tests verify the output variable reporting mechanism. The tests are written in Expect and the test shell is constructed using lex and yacc. The Expect files in `reporting/test/libreporting.all/` send commands to the test shell and check the results against the expected values. The test shell commands are

`variable (`***name***`,`***frequency***`)`	Creates a new output variable. The first argument may be any string, e.g., `"x"`, `"variable_1"`. The frequency must be one of `never`, `once`, or `daily`.
`set (`***value***`,`***category***`,`***sub-category***`,...)`	Sets the value of the most recently created variable. There can be an arbitrary number of sub-categories, or, the category and sub-categories can be omitted altogether.
`add (`***value***`,`***category***`,`***sub-category***`,...)`	Adds the value to the most recently created variable. Same comments apply as for `set`.
`subtract (`***value***`,`***category***`,`***sub-category***`,...)`	

	Subtracts the value from the most recently created variable. Not available for text variables. Same comments apply as for `set`.
`get (`***category*****`,`*****sub-category*****`,...`**`)`	Retrieves the value of the most recently created variable. Same comments apply as for `set`.

wml/

The wml/ tests verify the point-set functionality from the WML (Wild Magic Library). The tests are written in Expect and the test shell is constructed using lex and yacc. The Expect files in wml/`test/libwml.*/` send commands to the test shell and check the results against the expected values. The test shell commands are

`pointset (`***x1*****`,`*****y1*****`,`*****x2*****`,`*****y2*****`,...`**`)`	Creates a set of points.
`hull`	Finds the convex hull around the most recently created point set. The output from this command is a list of point indices, separated by spaces.
`point_in_poly (`***x*****`,`*****y***`)`	Finds whether the point ***x,y*** is inside the arbitrary polygon whose vertices are defined by the most recently created point set. The output is 't' or 'f'.