Unit Tests: Automated JSL Testing

Joseph Morgan and Xan Gregg, JMP

Software development practitioners use the term unit to refer to the smallest testable component of a software product and the phrase unit testing to refer to the process of validating that a unit operates as expected. Over the past twenty years, these practitioners have become increasingly interested in automated unit testing and the phrase unit testing framework has come to refer to the mechanism used to facilitate automated unit testing.

Purpose

Software developers, including JSL developers, have a vested interest in ensuring that the software they develop operates as expected. Apart from this obvious motivation, there are two additional reasons:

1. **Software evolution**: Software evolves over time. Typically, this is because features are added or changed but sometimes software evolves due to refactoring. A properly maintained suite of unit test cases, routinely used to validate the product, can ensure that the software product retains its expected behavior as it evolves. Units that regress (break or stop working) as the product evolves are quickly identified.
2. **Executable documentation**: Unit test cases document the expected behavior of software products. One of the benefits of automated unit testing is that the test cases act as executable documentation. As a result, as long as the test cases are routinely used to validate the product as it evolves, these test cases are likely to reflect the correct expected behavior of the software product, unlike static written documentation.

In recent years, the chief advocates of unit testing in general, and automated unit testing in particular, have been the eXtreme Programming (XP) community. Partly due to the enthusiasm of this community, unit testing frameworks are now available for every major programming language.

Features You Expect in a Framework

Frameworks should facilitate automated unit testing and so, at a minimum, should have the following features:

1. **Easy to use**: Not only should the framework make it easy to execute unit tests, it should also make the writing of unit tests simple and straightforward. There are several aspects to this:
    a. **Expected behavior specification capability**: A unit test specifies expected behavior for a given input. A framework should allow for this by providing some way to make an assertion about the output of the unit, or its state, given a particular input.
    b. **Expected error support**: A suite of unit test cases should include cases with invalid inputs as well as those with valid inputs. For invalid inputs, a framework should provide some way to trap the expected error behavior and treat it as a success.
    c. **Fuzzy comparison capability**: When the output from a unit is numeric, a value that is close enough to the expected value should be considered a success. The framework should therefore provide some way to specify a close enough level of accuracy.
    d. **Setup and cleanup support**: Often, a unit test requires that some setup operation be done prior to the test and the test should be followed by some cleanup operation. For example, the unit test may require that the input data for the test be obtained from an external data source, such as a relational database. This is an example of a setup operation. Closing the database connection would be a cleanup operation. The framework should provide an easy mechanism to do these operations.

    e. **Unit test management**: For anything but trivial software products, the unit test suite will likely consist of thousands of test cases. A framework should provide some way to easily retrieve test cases for execution or for maintenance.

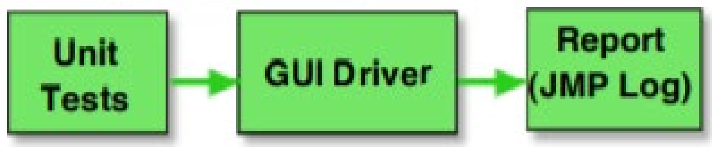
2. **Abstract away complexity of automation**: A framework should provide the superstructure needed to write automated unit tests but, at the same time, abstract away the complexity of this superstructure from the user. Furthermore, the framework should make it easy to execute a single unit test or a group of unit tests. It should also make it easy for the user/tester to query the repository of unit tests to determine which subset of unit tests, if any, provide coverage for a particular feature.
3. **Reporting**: A framework should, at a minimum, provide enough details to allow the user to easily identify the test cases that fail and the cause of failure. Also, the user needs to see a summary report with counts of successes and failures, as well as build and version information (if appropriate), plus the date of execution.

Why Unit Testing in JMP?

JSL has evolved into a full-featured scripting language that is increasingly being used to build non-trivial applications. We believe that as this phenomenon continues, JSL programmers will need more sophisticated development tools. The JMP JSL editor is one step towards that end, and we believe that JSL-Unit is another. JSL-Unit refers to a unit testing framework that may be used to test JSL applications as they evolve. JSL-Unit is completely written in JSL and so, for the JSL programmer, it is readily accessible. In addition, since it is written in JSL, it can easily be extended by JSL programmers. Furthermore, one of its main advantages when compared to third party unit testing frameworks is that it is guaranteed to work on any host for which JMP is available. JSL-Unit is particularly suited to a JSL development philosophy that views JSL applications as a collection of loosely coupled functions. Such functions may be thought of as units, and so fit naturally into the JSL unit testing framework. Notice, however, that the framework is not limited to validating simple functions that return a simple value. As long as the unit to be tested has a well-defined expected output (e.g., a JMP report or a JMP data table) then the framework can be used to validate it.

JSL-Unit Implementation

The architecture is as follows:



Unit Tests

Unit tests may be specified as either JSL scripts or JMP data tables. Expected behavior is specified by way of a JSL function defined by the GUI driver. The prototype of this function is:

```
ut assert ( expression, expected value )
```

This is the mechanism that test scripts use to make assertions about the behavior of a JSL application (i.e., to define test cases). An individual test case is an invocation of `ut assert` where the `expression` argument specifies the actual result and the `expected value` argument the expected result. A test case is considered a success (or a pass) if the actual and expected results are equal. The test case also passes if the expected and actual values are both missing. Consider the following example:

```
ut assert( Expr(Sqrt(1.21)), 1.1 );
```

For this example, the unit is the JSL internal **`sqrt`** function. The expression **`Sqrt(1.21)`** is evaluated and its result compared to **`1.1`**. Since both values are numbers, a fuzzy comparison is done.
The **`ut assert`** function understands JMP data types and so knows when to do fuzzy comparisons and when to do exact comparisons. For fuzzy comparisons, it allows the user to override the default level of accuracy. Furthermore, it allows the user to mark selected test cases as stress test cases. These are often edge test cases that are expected to fail for a variety of reasons (e.g., floating point round off error).
Now consider the following example:

```
ut assert( Expr(Sqrt()), ut_error );
```

The expected value **`ut_error`** is a special expected value that allows JSL-Unit to provide expected error support.

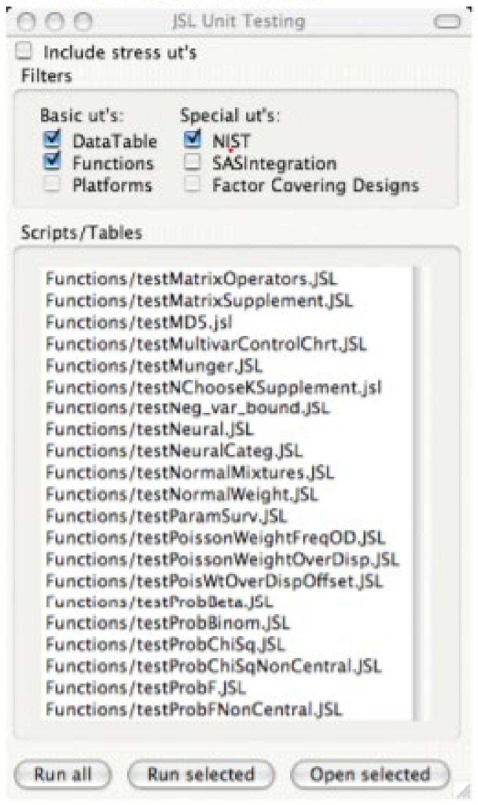
GUI Driver

The GUI driver

- provides the user interface
- defines the superstructure to automate unit tests
- provides the reporting capability
- provides the unit test management mechanism
- defines the **`ut assert`** assertion function.

As a result, the driver is the core of the framework. It makes the framework easy to use and abstracts away the complexity of automation (see Figure 1).

Figure 1 Unit Test Framework GUI



JSL Unit Testing

[ ] Include stress ut's

Filters

Basic ut's:	Special ut's:
[✓] DataTable	[✓] NIST
[✓] Functions	[ ] SASIntegration
[ ] Platforms	[ ] Factor Covering Designs

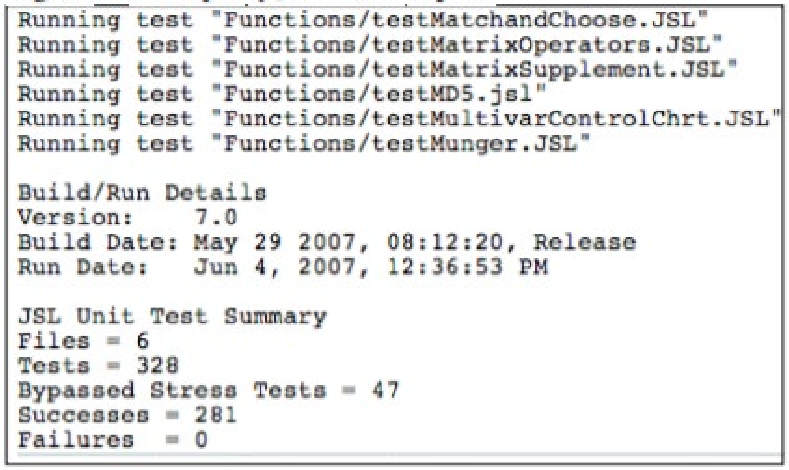
Scripts/Tables

```
Functions/testMatrixOperators.JSL
Functions/testMatrixSupplement.JSL
Functions/testMD5.jsl
Functions/testMultivarControlChrt.JSL
Functions/testMunger.JSL
Functions/testNChooseKSupplement.jsl
Functions/testNeg_var_bound.JSL
Functions/testNeural.JSL
Functions/testNeuralCateg.JSL
Functions/testNormalMixtures.JSL
Functions/testNormalWeight.JSL
Functions/testParamSurv.JSL
Functions/testPoissonWeightFreqOD.JSL
Functions/testPoissonWeightOverDisp.JSL
Functions/testPoisWtOverDispOffset.JSL
Functions/testProbBeta.JSL
Functions/testProbBinom.JSL
Functions/testProbChiSq.JSL
Functions/testProbChiSqNonCentral.JSL
Functions/testProbF.JSL
Functions/testProbFNonCentral.JSL
```

Figure 1 Unit Test Framework GUI

Report
JSL-Unit uses the JMP log for reporting. Figure 2 shows an example of the type of report produced:

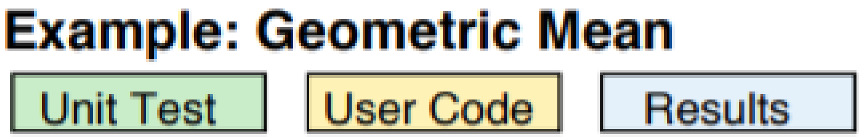
Figure 2 Example of JSL-Unit Report



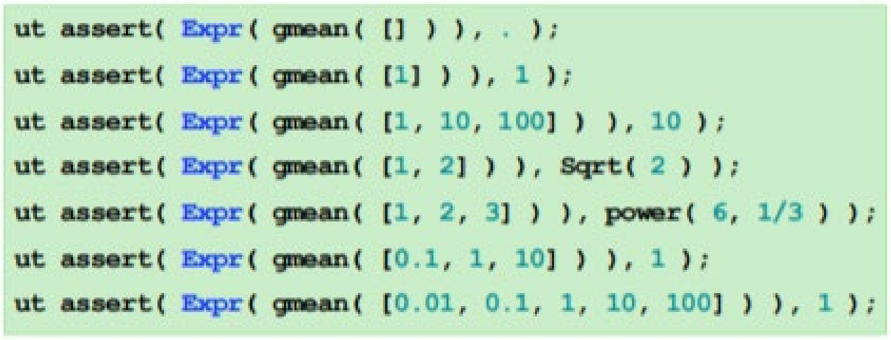
```
Running test "Functions/testMatchandChoose.JSL"
Running test "Functions/testMatrixOperators.JSL"
Running test "Functions/testMatrixSupplement.JSL"
Running test "Functions/testMD5.jsl"
Running test "Functions/testMultivarControlChrt.JSL"
Running test "Functions/testMunger.JSL"

Build/Run Details
Version:    7.0
Build Date: May 29 2007, 08:12:20, Release
Run Date:   Jun 4, 2007, 12:36:53 PM

JSL Unit Test Summary
Files = 6
Tests = 328
Bypassed Stress Tests = 47
Successes = 281
Failures  = 0
```

[Text Wrapping Break]

JSL-Unit Usage at SAS
JSL-Unit was developed to automate the testing of JMP itself and has been heavily used by JMP development over the past two years. The repository of unit tests currently contains several hundred thousand test cases organized into approximately hundreds of unit test suites. These test cases validate JMP internal functions and JMP platforms. The entire suite of test cases is run on each host (MacOS and Windows) for each new build of JMP.



The *geometric mean* of a data set a_1, a_2, ... a_n is the nth root of the product of all the values. The geometric mean is useful for finding the average multiplier.
For example, if an investment returns 7%, –5% and 28% in consecutive years, the average return is the geometric mean of the multipliers, 1.07, 0.95, and 1.28, which is 1.09, or a 9% return. Using the arithmetic mean would show an 11% average return.
Our first step is to write some tests to define how our new function should behave, taking the geometric mean of numbers that are elements of a matrix. The initial tests are simple cases where we can compute the answer directly.



```
ut assert( Expr( gmean( [] ) ), . );
ut assert( Expr( gmean( [1] ) ), 1 );
ut assert( Expr( gmean( [1, 10, 100] ) ), 10 );
ut assert( Expr( gmean( [1, 2] ) ), Sqrt( 2 ) );
ut assert( Expr( gmean( [1, 2, 3] ) ), power( 6, 1/3 ) );
ut assert( Expr( gmean( [0.1, 1, 10] ) ), 1 );
ut assert( Expr( gmean( [0.01, 0.1, 1, 10, 100] ) ), 1 );
```

Next we write our simple JSL function to compute the geometric mean using the Product operator and the Power (^) operator.

```
gmean = Function( {m},
    Local( {p, i},
    p = Product( i = 1, N Row( m ), m[i] );
    p ^ (1 / N Row( m ));
    )
  );
```

```
JSL Unit Test Summary
Successes = 7
Failures = 0
```

When we run the tests with the test driver, the output report shows that all tests pass. ☺
We extend the last test to cover all powers of 10 from 10^{-7} to 10^{7} with the Index (::) operator, and add a similar test with numbers centered around 2.

```
ut assert( Expr( gmean( 10 ^ (-7 :: 7) ) ), 1 );
ut assert( Expr( gmean( 2 * 10 ^ (-7 :: 7) ) ), 2 );
```

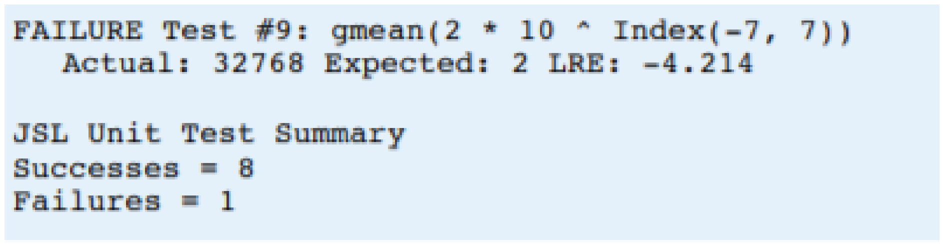
Running all tests again shows that all new tests fail. ☹

```
FAILURE Test #8: gmean(10 ^ Index(-7, 7))
   Actual: 0.0000001 Expected: 1 LRE: 0.00000004
FAILURE Test #9: gmean(2 * 10 ^ Index(-7, 7))
   Actual: 0.0000002 Expected: 2 LRE: 0.00000004

JSL Unit Test Summary
Successes = 7
Failures = 2
```

The failures are because the :: operator produces a row vector, but our **gmean** function only handles column vectors. We update **gmean** to cycle over rows *and* columns by using nested **Product** operators.

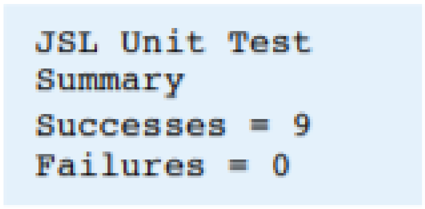
```
gmean = Function( {m},
     Local( {p, i, j},
         p = Product( i = 1, N Row( m ),
             Product( j = 1, N Col( m ), m[i, j]
)
         );
         p ^ (1 / N Row( m ));
     )
```

This test also fails, though with a different result. ☹

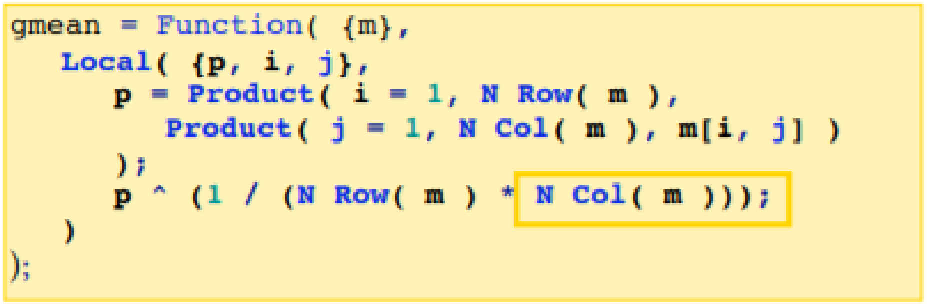


```
FAILURE Test #9: gmean(2 * 10 ^ Index(-7, 7))
   Actual: 32768 Expected: 2 LRE: -4.214

JSL Unit Test Summary
Successes = 8
Failures = 1
```

It looks like we forgot to consider the number of columns when taking the root of the product. A new version of our function fixes that.

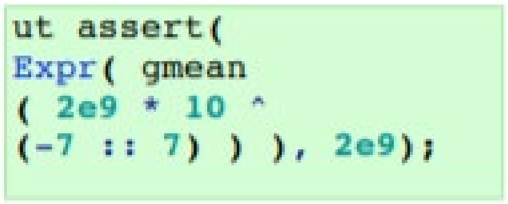


```
JSL Unit Test
Summary
Successes = 9
Failures = 0
```

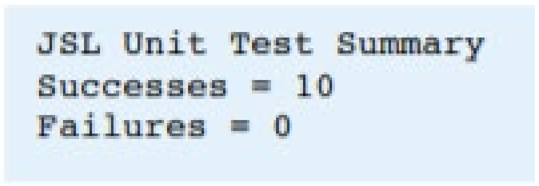


```
gmean = Function( {m},
   Local( {p, i, j},
      p = Product( i = 1, N Row( m ),
         Product( j = 1, N Col( m ), m[i, j] )
      );
      p ^ (1 / (N Row( m ) * N Col( m )));
   )
);
```

Now all tests pass again.☺
Next we add a test for really large values. Thanks to the framework's consideration of relative errors, expected rounding errors don't cause problems and the test passes.

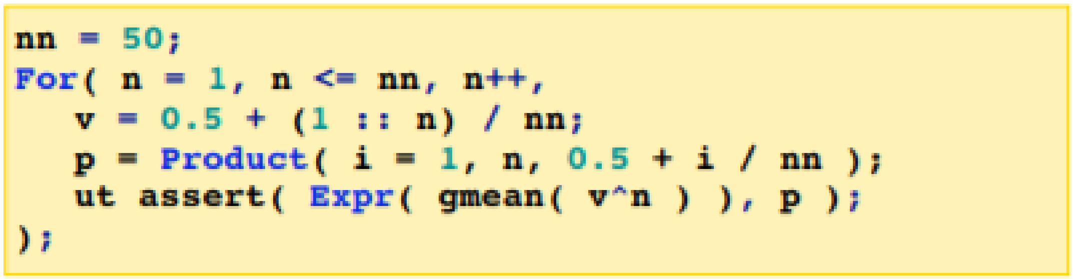


```
ut assert(
Expr( gmean
( 2e9 * 10 ^
(-7 :: 7) ) ), 2e9);
```



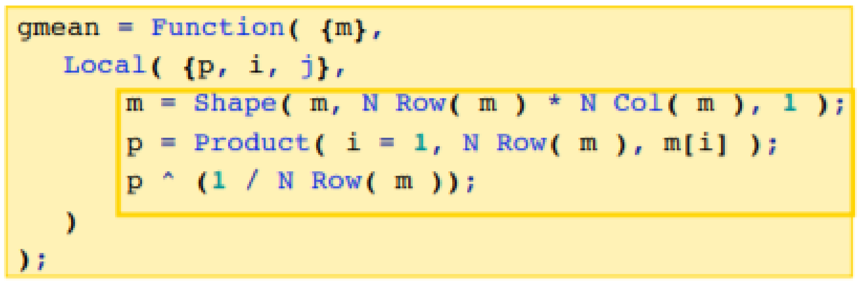
```
JSL Unit Test Summary
Successes = 10
Failures = 0
```

For more thorough testing, we write a loop that generates data sets and checks that **gmean** computes the right answer. So that we can know the answer without using **gmean**, we create the data so that each value is raised to the nth power so when **gmean** takes the nth root, it should just get back the product of the original values.



```
nn = 50;
For( n = 1, n <= nn, n++,
   v = 0.5 + (1 :: n) / nn;
   p = Product( i = 1, n, 0.5 + i / nn );
   ut assert( Expr( gmean( v^n ) ), p );
);
```

All tests still pass.☺ Notice there are now 60 tests since our added test was in a loop and was applied 50 times.
Finally, now that we have a good suite of test, we can confidently change the implementation of our function, knowing that any regression will trigger a unit test failure.
We change the implementation by reverting the nested product to a simple product after using the Shape operator to convert the matrix into a row vector.



```
gmean = Function( {m},
   Local( {p, i, j},
       m = Shape( m, N Row( m ) * N Col( m ), 1 );
       p = Product( i = 1, N Row( m ), m[i] );
       p ^ (1 / N Row( m ));
   )
);
```

The final test passes; the function is complete.☺
Note: You may download JSL-Unit from the JMP Scripts section of the JMP User community site: community.jmp.com.