CMPT 473 SOFTWARE QUALITY ASSURANCE

Project Report

DECEMBER 5, 2015

SCHOOL OF COMPUTING SCIENCE, SIMON FRASER UNIVERSITY
Daphne Ordas, Matthew Chow, Young Yoon Choi (Justin), Wei Wang
Team DJW

Contents

Contact Information ..... - 2 -
Software Specification ..... - 2 -
Test Plan Summary ..... - 3 -
Test Objectives ..... - 3 -
Functional Factors ..... - 3 -
Nonfunctional Factors ..... - 3 -
Resource & Constraints ..... - 4 -
Combinatorial Testing ..... - 5 -
Input Space Model ..... - 5 -
Constraints ..... - 8 -
ACTS Settings ..... - 8 -
Mutation Adequacy Testing ..... - 11 -
_ode.py Live Mutations ..... - 12 -
Quadpack.py Live Mutations ..... - 13 -
Quadrature.py Live Mutations ..... - 14 -
Unique and Interesting Mutations Observations ..... - 15 -
Recommendations and Conclusions ..... - 17 -
Performance Testing ..... - 18 -
Execution Speed Testing ..... - 18 -
Accuracy Testing ..... - 20 -
White-Box Testing ..... - 21 -
Test Result table ..... - 23 -
Conclusion ..... - 24 -
Failed Cases ..... - 25 -
Fuzzing Testing ..... - 25 -

Contact Information

Daphne Ordas	dordas@sfu.ca
Matthew Chow	mkc25 @sfu.ca
Young Yoon Choi (Justin)	yychoi@sfu.ca
Wei Wang	wwa53@sfu.ca

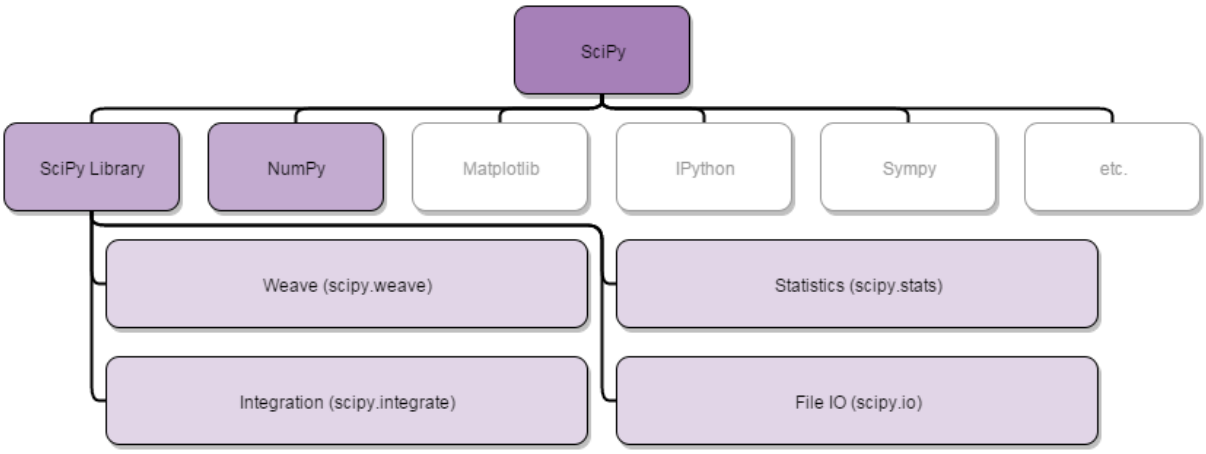
Software Specification

Name:	Scientific Computing Tools for Python ---- SciPy
Developers:	Community Library Project
Stable release:	0.16.1 (Oct 24, 2015)
Written in:	Python
Operating system:	Cross Platform
Available in:	English
Type:	Technical Computing
License:	BSD-New License (Open Source)
Website:	http://www.scipy.org/ https://github.com/scipy/scipy
Download:	http://sourceforge.net/projects/scipy/files/scipy/0.16.1/

> *SciPy (pronounced "Sigh Pie") is an open source Python library used by scientists, analysts, and engineers doing scientific computing and technical computing.*
>
> *SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering. SciPy builds on the NumPy array object and is part of the NumPy stack which includes tools like Matplotlib, pandas and SymPy. There is an expanding set of scientific computing libraries that are being added to the NumPy stack every day. This NumPy stack has similar users to other applications such as MATLAB, GNU Octave, and Scilab. The NumPy stack is also sometimes referred to as the SciPy stack.*
>
> *The SciPy library is currently distributed under the BSD license, and its development is sponsored and supported by an open community of developers.*
>
> *---- Wikipedia https://en.wikipedia.org/wiki/SciPy*

The full set of test subject consists of several libraries: Scipy Library, NumPy, Matplotlib, IPython, Sympy, Pandas etc. The test will focus on Statistics, File IO, and Weave packages within SciPy Library and NumPy. The general software structure is shown in the image below. Since it is constantly maintained by community, only a selected version will be tested among all group member to obtain a consistent result from following link:

> *http://sourceforge.net/projects/scipy/files/scipy/0.16.1/*



Test Plan Summary

Test Objectives

The overall objective of this test is to estimate the efficiency of current test frame and team while improve the quality of selected packages. Tests shall focus on following part according to various modules:

- Functional testing using combinatorial method - scipy.io
- Mutation adequacy evaluation - scipy.integrate
- Security analysis and testing - scipy.weave
- Test data adequacy evaluation with white-box - scipy.stats
- Performance Testing - numpy

Detailed illustrations will be presented for each section.

Functional Factors

The functional test will focus on correctness of common user interaction while digging into special extreme cases. Common user interaction includes, for example, computing an integral of a certain function and is the foundation of scientific computation. Thus, it is most widely used and its quality will determine other functionalities built on it. This is the key and priority of software testing for scientific applications. In addition, since extreme cases are fairly common in scientific computation like, for example, a matrix with millions of elements or overflow/underflow when calculate two huge numbers, we shall also focus on this other aspect.

Nonfunctional Factors

Nonfunctional tests shall target two parts: Efficiency and Accuracy. Efficiency is a big issue when come to scientific computation. The test shall predefine a tolerance to how fast the program shall generate the result and how large the space is needed to conduct such computations. Timeouts or memory overflows are fairly common in scientific computation even if the algorithm is theoretically implemented correctly.

Accuracy is another big issue when dealing with extreme numbers. The application is expected to generate a result as accurate as possible within the capability of the data structure it is using. For example, if the double accuracy float number is in 64bit format, the final result shall be correct at least 2^{63} or following industry double accuracy float standard defined by IEEE.

Other concerns may include various configurations and environments. The software packages shall be tested on different versions of Python, operating system, and hardware. This shall be done by utilizing various equipment possessed by each individual team member. Also, virtual machines and other technologies can be introduced to help test non resource focused test cases.

Resource & Constraints

Since scientific computation is quite well developed among different platforms, there are multiple existing resources that can act for references. For example, if a test case needs an oracle for a correct computational result, it can be generated by passing similar calls to MATLAB or Wolfram Alpha. In another words, the test oracles can be obtained from other well developed scientific computation tools.

Efficiency shall be the major constraint for this testing. Since the test cases consist of huge input data and its results, manual operation is much depreciated here. The test case generation shall be automated as much as possible while each case shall also be able to run on various platforms. Hardware limitations will be taken into consideration when designing the test cases.

Combinatorial Testing

SciPy supports MATLAB, IDL, Matrix Market, Wav sound, Arff and Netcdf file formats. For our purposes we will only be testing functions that deal with MATLAB files. MATLAB functions were chosen and prioritized over other formats because they contained the most flags and thus had the highest chance of inducing errors during testing.

SciPy has three functions for interpreting MATLAB files: loadmat, savemat, and whosmat. They are responsible for loading MATLAB files into SciPy, saving dictionaries of names and arrays into MATLAB compatible formats, and listing variables in MATLAB files.

We used the Octave tool to create our MATLAB files. While not a true representation of the MATLAB programming language, it is flexible and open source which serves our testing needs adequately.

Input Space Model

Parameters for scipy.io.loadmat (file_name, mdict=None, appendmat=True, **kwargs) - 11 flags, 10 optional

- **file_name**
  - Function: specifies input file
  - Type: string, non-optional
    - valid_file: valid MATLAB (.mat) file
    - invalid_file: file is not found
    - empty_file: empty file
- **m_dict***
  - Function: specifies dictionary to insert Matfile variables
  - Type: dictionary, optional
    - valid_dict: { [1, 2], [3, 4], [5, 6] }
    - invalid_ dict: { [[1, 2] [3, 4]], [5, 6] }
    - empty_dict: { }
- **appendmat**
  - Function: if true, append .mat at end of file
  - Type: boolean, optional
- **byte_order**
  - Type: enumeration, optional
    - ‘native’
    - ‘=’
    - ‘little’
    - ‘<’
    - ‘BIG’
    - ‘>’
    - invalid: any other chars or str than above
- **mat_dtype**
  - Type: boolean, optional
  - Function: If True, return arrays in same dtype.

- **squeeze_me**
  - Type: boolean, optional
  - Function: if true squeeze unit matrix dimensions
- **chars_as_strings**
  - Type: boolean, optional
  - Function: if True convert char arrays to string arrays
- **MATLAB_compatible**
  - Type: boolean, optional
  - Function: if True matrices are as MATLAB would load
- **struct_as_record**
  - Type: boolean, optional
  - Function: if True load MATLAB structs as numpy record arrays; if false load MATLAB structs as as old-style numpy arrays with dtype=object.
- **verify_compressed_data_integrity**
  - Type: boolean, optional
  - Function: if True, check compressed sequences in MATLAB files for corruption
- **variable_names**
  - Type: enumeration
    - all_vars_seq: a, b, c
    - no_vars_seq: x, y, z
    - invalid_vars: /, \
    - some_vars_seq: a

Parameters for scipy.io.savemat (file_name, mdict, appendmat=True, format='5', long_field_names=False, do_compression=False, oned_as='row') – 7 flags, 6 optional

- **file_name**
  - Function: specifies input file
  - Type: string, non-optional
    - valid_file: valid MATLAB (.mat) file
    - invalid_file: file is not found
    - empty_file: empty file
- **m_dict***
  - Function: specifies dictionary to insert Matfile variables
  - Type: dictionary, optional
    - valid_dict: { [1, 2], [3, 4], [5, 6] }
    - invalid_ dict: { [[1, 2] [3, 4]], [5, 6] }
    - empty_dict: { }
- **appendmat**
  - Function: if true, append .mat at end of file
  - Type: boolean
- **format**
  - Function: sets MATLAB version
  - Type: enumeration
    - '5'
    - '4'
    - form_str: '6'

        - invalid: '100'
- **long_field_names**
    - Function: if True max field name lengths is 63 chars; if false 31.
    - Type: boolean
- **do_compression**
    - Function: if True compress matrices
    - Type: boolean
- **oned_as**
    - Function: if 'column', write 1-D numpy arrays as column vectors. If 'row', write 1-D numpy arrays as row vectors.
    - Type: enumeration

Parameters for scipy.io.whosmat (file_name, appendmat=True, **kwargs) – 8 flags, 7 optional

- **file_name**
    - Function: specifies input file
    - Type: string
        - valid_file: valid MATLAB (.mat) file
        - invalid_file: file is not found
        - empty_file: empty file
- **m_dict***
    - Function: specifies dictionary to insert Matfile variables
    - Type: dictionary
        - valid_dict: { [1, 2], [3, 4], [5, 6] }
        - invalid_ dict: { [[1, 2] [3, 4]], [5, 6] }
        - empty_dict: { }
- **appendmat**
    - Function: if true, append .mat at end of file
    - Type: boolean
- **byte_order**
    - type: enumeration
        - 'native'
        - '='
        - 'little'
        - '<'
        - 'BIG'
        - '>'
        - invalid: any other chars or str than above
- **mat_dtype**
    - Type: boolean
    - Function: If True, return arrays in same dtype as would be loaded into MATLAB
- **squeeze_me**
    - type: boolean
    - function: if true squeeze unit matrix dimensions
- **chars_as_strings**
    - type: boolean

    - function: if True convert char arrays to string arrays
- **MATLAB_compatible**
    - type: boolean
    - function: if True matrices loaded as MATLAB matrices
- **struct_as_record**
    - type: boolean
    - function: if True, load MATLAB structs as numpy record arrays. If false load as old-style numpy arrays with dtype=object

* Official SciPy documentation details the flag as “m_dict”. However, when testing I had to use “mdict” as “m_dict” was not a valid command.

Constraints

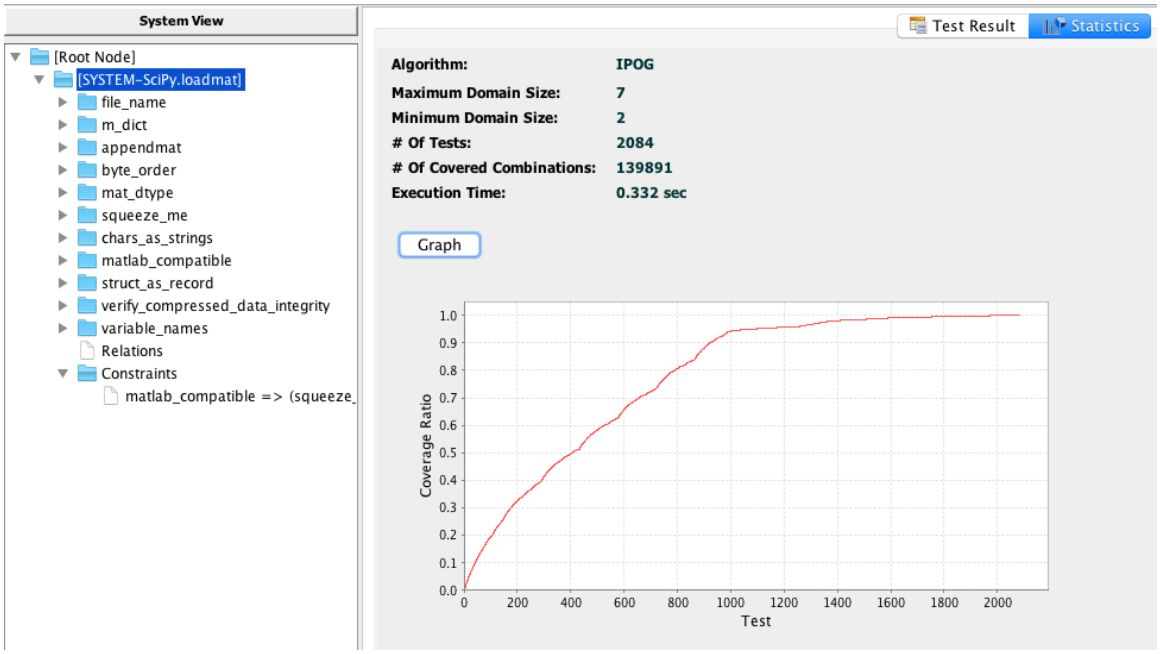
Only loadmat and whosmat had constraints. The mat_compatible flag, shared between these two functions, had the same constraint. When mat_compatible is True, squeeze_me and chars_as_string are implied or overwritten as False, while mat_dtype and struct_as_record as implied or overwritten as True.

```
MATLAB_compatible => implies squeeze_me=False,
chars_as_strings=False, mat_dtype=True, struct_as_record=True
```

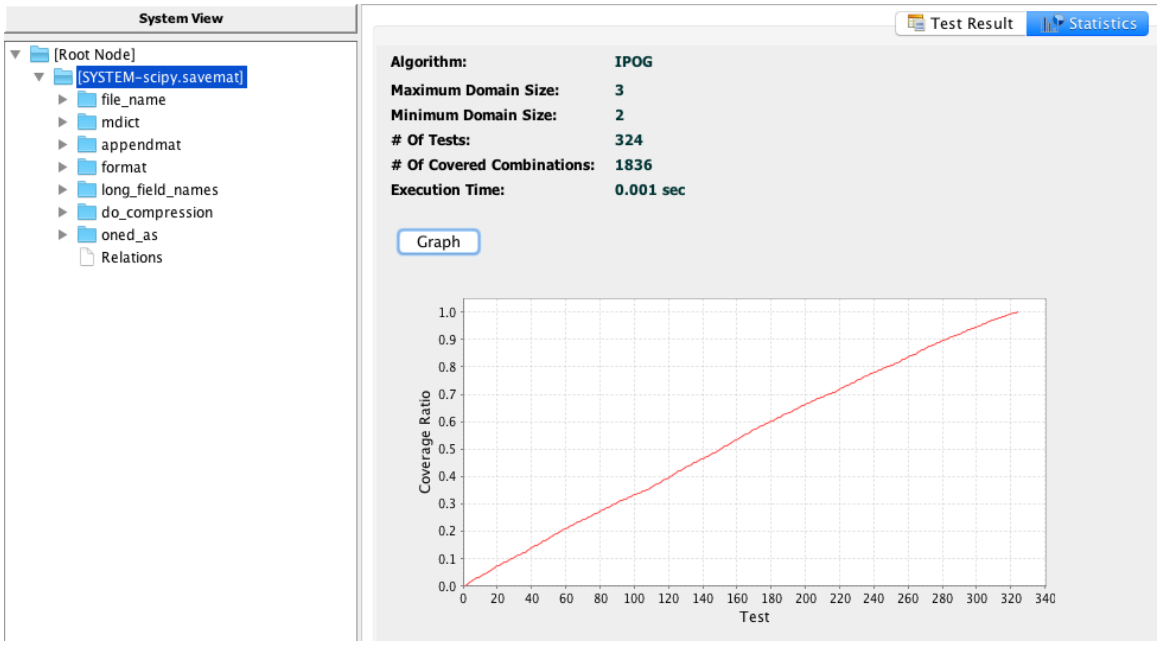
ACTS Settings

Algorithm: IPOG Strength: 6

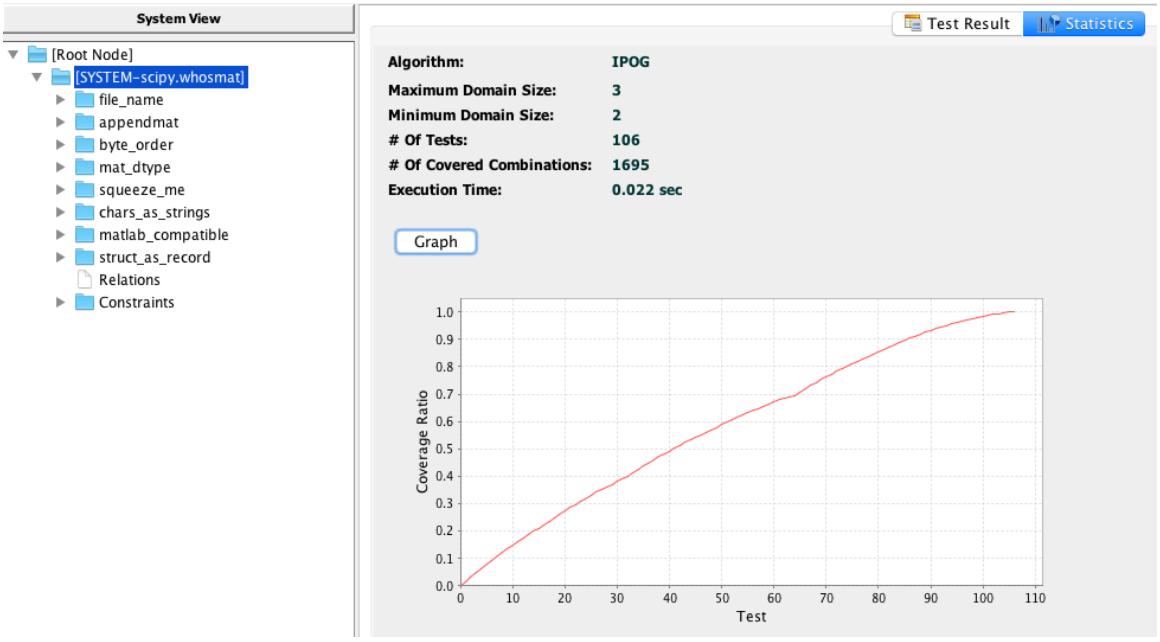
Using ACTS we generated 2084, 324, and 106 tests respectively, for a total of 2514 tests for these three functions. Combination wise we covered 139891, 1836, and 1695 respectively, for a total of 143422 combinations covered.



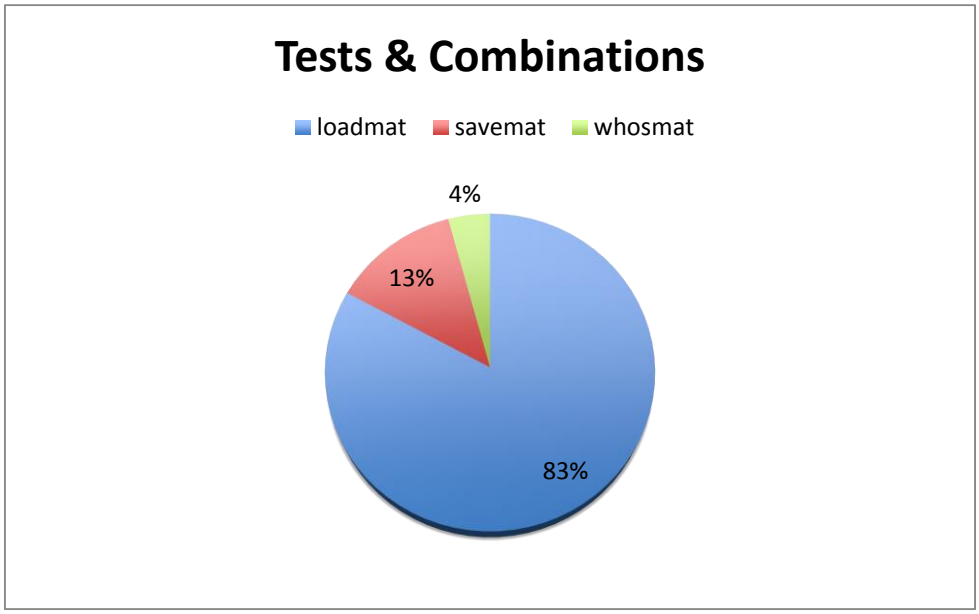
SciPy.io.loadmat in ACTS



SciPy.io.savemat in ACTS



SciPy.io.whosmat in ACTS

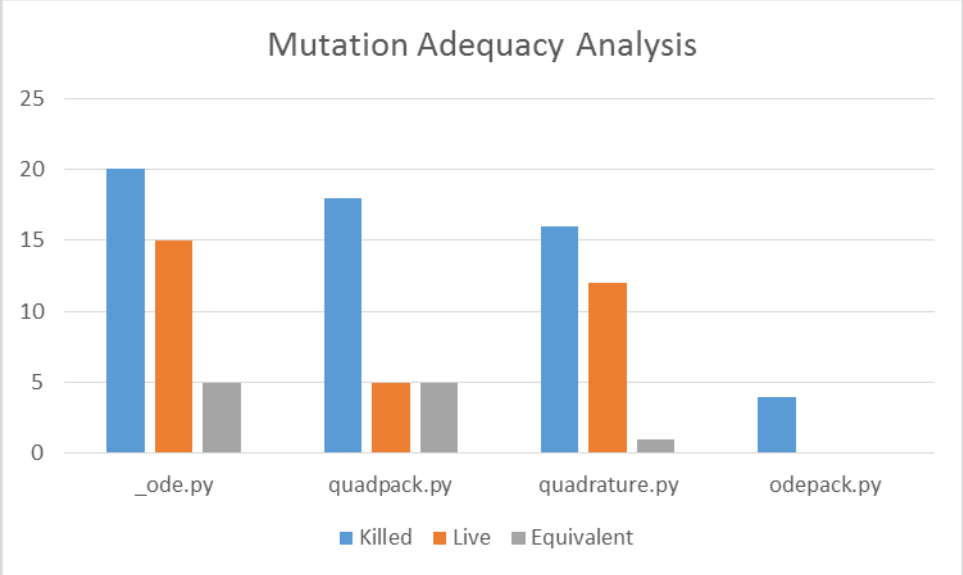


Script

We generated the python code by parsing CSV data in Java from ACTS. However we were unable to run our tests automatically as we couldn't run it outside the Python.

Mutation Adequacy Testing

The mutation adequacy was performed on the "integrate" folder, on the following files: _ode.py, quadpack.py, quadrature.py, and odepack.py. Instead of sticking to one file, there was a need to cover each of the files that corresponded to the test suite to increase the coverage, and also to test the essential functions. The "integrate" test suite contains 243 tests, which takes on average about 3 – 4 seconds to run. For each file, manual mutations were applied, and these mutations are absolute value insertion (ABS), arithmetic operator replacement (AOR), relational operator replacement (ROR), conditional operator replacement (COR), removing the self-reference (self), and replacing constants (RepC). There were 101 mutations applied in total. To analyze this data, we will use the formula take from class: TS=K/(M-E)



_ode.py Mutation Adequacy Analysis: 20/(40 - 5) = 0.571, or 57% mutation adequacy
quadpack.py Mutation Adequacy Analysis: 18/(28 – 5) = 0.782 or 78% mutation adequacy
quadrature.py Mutation Adequacy Analysis: 16/(29 - 1) = 0.571 or 57% mutation adequacy
odepack.py Mutation Adequacy Analysis: 4/(4 - 0) = 1, or 100% mutation adequacy
Overall Mutation Adequacy Analysis: 58/(101 - 11) = 0.644, or 64% mutation adequacy

From the above, we can see that respectively _ode.py and odepack.py had the worst mutation adequacy scores out of the other files, while odepack.py had the best. _ode.py and odepack.py had the worst scores probably due to the overall complexity of their functions, and these files were also the longest files of these 4. odepack.py was relatively simpler and shorter, which is why it had the best score. The overall mutation adequacy is surprising, as there were many test cases that encapsulated all the files. The top contributor to the overall score would probably be _ode.py, which means that this file should have a lot more tests designed for this file, as the most live mutants was found from this file.

Because there were many mutations applied during the process, we'll focus on the live and equivalent mutations and determine why they are live or equivalent.

_ode.py Live Mutations

Line Number	Mutation Applied	Why it's live or equivalent
505	AOR	This mutation is live because the code is not checking the answer; it is probably checking if the answer exists.
518	ROR	This mutation is live, but the cause of this mutation could not be determined.
690	RepC	This mutation is live because the constant isn't being checked for inside the test cases that were written.
691	RepC	This mutation is live because the constant isn't being checked for inside the test cases that were written.
692	RepC	This mutation is live because the constant isn't being checked for inside the test cases that were written.
696	COR	This mutation is live because the test doesn't check for the correctness of the default value.
723	ABS	This mutation is live, but the cause of this mutation could not be determined.
725	COR	This mutation is live because the test doesn't check for the correctness of the default value.
771	ROR	This mutation is live, but the cause of this mutation could not be determined.
807	ABS	This is an equivalent mutation, the formula probably takes in both positive and negative values.
855	RepC	This mutation is live because there are no checks if the value was changed in the code.
880	RepC	This mutation is live because there are no checks if the value was changed in the code.
881	RepC	This mutation is live because there are no checks if the value was changed in the code.

883	RepC	This mutation is live because there are no checks if the value was changed in the code.
893	ABS	This is an equivalent mutation, the formula probably takes in both positive and negative values.
899	ABS	This is an equivalent mutation, the formula probably takes in both positive and negative values.
911	ABS	This is an equivalent mutation, the formula probably takes in both positive and negative values.
1061	RepC	This mutation is live because there are no checks if the value was changed in the code.
1156	AOR	This is an equivalent mutation, because after switching the operators the answer turned out to be equivalent.
1162	AOR	This mutation is live because the test suite returned errors but the integration proceeded anyway.

Quadpack.py Live Mutations

Line Number	Mutation Applied	Why it's live or equivalent
317	ROR	This mutation is live because the if statement cases were not checked by the tests.
330	ROR	This is an equivalent mutation where 'and' and 'or' seem to be equivalent.
347	ROR	This is an equivalent mutation where 'and' and 'or' seem to be equivalent.
366	ABS	This mutation is live because in the code, the comment said it ignored the case, so the case probably wasn't tested by the test suite.
369	ABS	This mutation is live because the if statement cases were not checked by the tests.
384	AOR	This is an equivalent mutation because the list was just made bigger than the normal size,

		which could hold more things.
411	ROR	This is an equivalent mutation, where 'and' and 'or' seem to be equivalent here.
430	ROR	This mutation is live, but the cause of this mutation could not be determined.
689	ROR	This mutation is live because warnings appeared, but the test cases weren't prepared to handle the results of this mutation. The test suite crashed, but did not return a success or failure report. This mutation is considered interesting, and a more in-depth analysis is provided below.
743	ROR	This is an equivalent mutation, as the boundary seems to not be affected by the change in operator.

Quadrature.py Live Mutations

Line Number	Mutation Applied	Why it's live or equivalent
75	COR	This mutation is live because there are no checks if the value was changed in the code.
78	AOR	This mutation is live because the formula itself is not checked for correctness in the test suite itself.
187	ABS	This is an equivalent mutation, as the absolute value is taken in the end, even if a negative was applied.
308	AOR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
385	COR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
396	ROR	This mutation is live because the if statement's different cases aren't being checked for

		inside the test cases that were written.
399	ROR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
550	ROR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
673	COR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
694	COR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
822	COR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
826	AOR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.
836	COR	This mutation is live because the if statement's different cases aren't being checked for inside the test cases that were written.

Unique and Interesting Mutations Observations

An interesting observation to note was that self-mutations were completely killed, making the test suite adequate for these kinds of mutations. There is no worry about self-mutations. Out of all the live mutations that were listed in the above charts, there were two live mutations that were of interest while running the test suite. The first interesting, live mutation was found in quadpack.py, in line 689 (the mutation is in red):

if opts is **not** None: `opts = [dict([])] * depth`

This live mutation, when run with the test suite, made the test show a warning which read:

```
>>> scipy.integrate.test()
Running unit tests for scipy.integrate
NumPy version 1.6.1
NumPy is installed in /usr/lib/python2.7/dist-packages/numpy
SciPy version 0.16.1
SciPy is installed in /usr/local/lib/python2.7/dist-packages/scipy
Python version 2.7.3 (default, Jun 22 2015, 19:43:34) [GCC 4.6.3]
nose version 1.1.2
................................................................................
................................................................................
.........................................K....../usr/local/lib/python2.7/dist-p
ackages/scipy/integrate/quadpack.py:352: IntegrationWarning: The maximum number
of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  warnings.warn(msg, IntegrationWarning)
```

and made the test suite hang for a long time, without showing a pass or a failure. By reading this warning, one would expect that the function that this test is going to finish, because it doesn't really detract from the overall function of the program, and a maximum limit was set. This could be a potential bug which SciPy could watch out for and improve upon when making improvements to the program. The other interesting live mutation was from _ode.py, in line 1162(the mutation is in red): **`liw = 20 - n`** .This is what comes up when running the mutation with the test suite:

```
................................................................................
.................. lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
. lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
 lsoda--  warning.. iwork length is sufficient for now, but
       may not be later.  integration will proceed anyway.
       length needed is leniw = i1, while liw = i2.
      in above message,  i1 =         21   i2 =         20
.......................K.......................................
----------------------------------------------------------------------
Ran 243 tests in 4.046s
```

Amazingly, the tests were successful in running, even though many warnings were given. It was also amazing that the function that it was doing, integration, kept on proceeding even

with these errors. If these warnings are not heeded, it could lead to another part of the program becoming affected by this and become a combination of failures.

Recommendations and Conclusions

From the two interesting live mutants mentioned above, SciPy could be further improved by enforcing maximum values depending on the use of this value, such that the size of values do not get out of hand and the program could no longer handle it. Also, test cases such as boundary checking and checking that constant values are the values they're supposed to be would be useful towards the overall test coverage of the test suite, as a lot of mutations were not caught with these mutations. In addition, checking the absolute value of certain values and checking that the answers make sense instead of checking for just an existing answer would benefit the current set of tests greatly.

Finally, a lot of mutations were live because within each COR mutation or ROR mutation, the different cases for if statements weren't checked. To explain, the tests would go through those different control flows to get to those cases, to see if they were actually valid. As a suggestion, there should be test cases that focus on showing if the if statement cases are checked. Not all of the different parts of the if statement should be checked, as there would be too many repetitive test cases, but enough that it would catch a bit more of the COR mutations.

In conclusion, even though there are many test cases for the "integrate" folder, from the mutation adequacy score of 64% we can see that the test cases need a little work. The prime focus of the addition of test cases should be biased towards _ode.py, as it is the most complex and longest of the 4 files that were mutated.

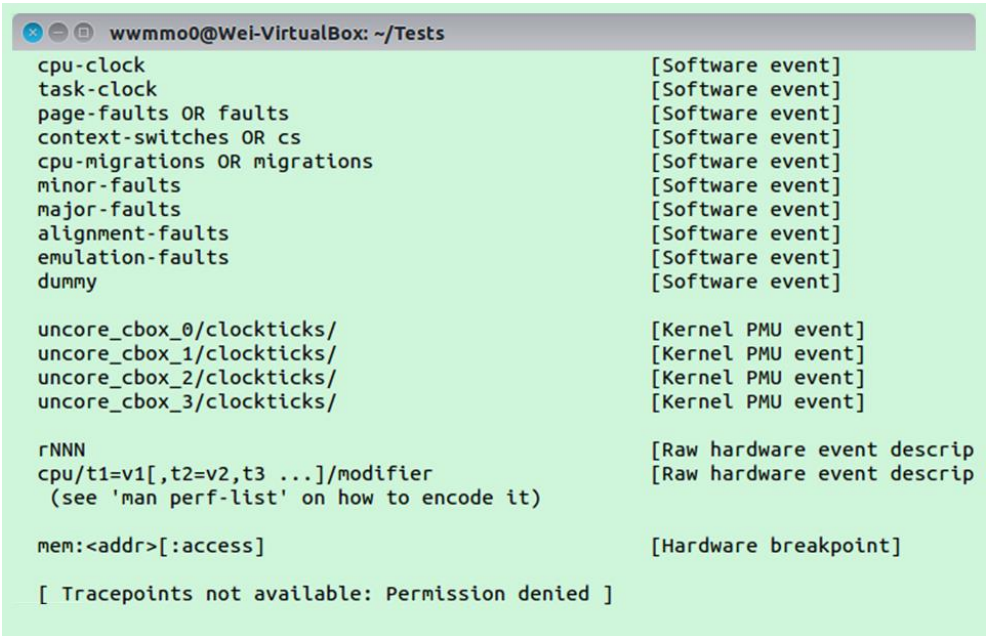
Performance Testing

Execution Speed Testing

Compared with other types of applications, scientific computation is more sensitive to performance. Thus, the major concern leads to the optimization of programming language and library. In this section, the project compared efficiency among MATLAB, Python, NumPy, Java, and C. Each environment has its own characteristics. The comparison is based on solving the following question:

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 +
        (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 + dy**2))
```

The example above is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows: It is required to solve for some unknown function u(x,y) such that $\nabla^2 u = 0$ with a boundary condition specified. This question basically requires iterating over a matrix with some computations. Matrices are a common data structure in math applications which is crucial to SciPy. Particularly, matrix functions are handled by NumPy. Thus, the comparison involves NumPy rather than other SciPy libraries. Meanwhile, matrix iterations (nested loops) are a good way to evaluate the efficiency of certain implementations. Since the project is not focusing on implementing complex programs but testing existing software, the testing code is obtained from http://scipy.github.io/old-wiki/pages/PerformancePython which is already offered by the SciPy project. The test only implemented extra code segments for particular environments, including C, Java etc. Evaluation is done by Linux built-in performance tool sets. Since testing is done on a virtual machine, the tool sets only supports the following feedback which are acquired by **`perf list`**:



The trial will take an average of 30 executions to reduce the impact of the caching system. CPU-Clock will be focused to record the execution time. However, any unusual Context-Switches or Page-Faults will also be monitored since they may indicate the program didn't optimize the execution enough.

NumPy V.S. Pure Python

Compared with pure Python, NumPy takes advantage of matrix handling. The implementation behind such a thing is not investigated. However, it does dramatically reduce the line of code the programmer needs to input:

```
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                 (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv
```

Also, the format is quite similar to MATLAB. People can adapt to this new language relatively fast compared with others. Test results are as followed:

NumPy (msec)	Pure Python (msec)
550.564730 (100%)	58545.773521 (10633.77%)

NumPy V.S. MATLAB

MATLAB (**mat**rix **lab**oratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. It is the major competitor of NumPy. The advantage of using NumPy is it does not require the stand alone platform to interpret the script, but can be easily integrated into any Python program. Also, it is free. The following is a reference on how to convert NumPy function to MATLAB format with test results:

http://mathesaurus.sourceforge.net/MATLAB-numpy.html

NumPy (msec)	MATLAB (msec)
550.564730 (100%)	63.366006 (11.51%)

Pure Python V.S. Other Languages

Python, as a scripting language, has performance much worse than other languages which need to compile before execution. The binary code to machine code can accelerate the speed dramatically. Meanwhile, Java needs JVM as a middle layer to execute its code. It greatly increased the ability cross different platform. However, the execution speed is reduced in between Python and CXX. Results are as follow:

Pure Python (msec)	Java (msec)	C++ (msec)
58545.773521 (100%)	19581.855509 (33.45%)	71.314734 (0.12%)

Execution Speed Testing Summary

	Test Results (msec)
NumPy	550.564730
MATLAB	63.366006
Pure Python	58545.773521
Java	19581.855509
C++	71.314734

As the result shows, NumPy shows a relative good performance. However, its competitor MATLAB went even further and achieved a better result than even C++.

Accuracy Testing

For scientific computations, accuracy is a great deal when it comes to extreme numbers. Even a tiny deviation on single intermediate step can cause serious problems in final result. Thus, accuracy become another factor for testing the library. This test is done by solving a 50x50 matrix with NumPy, MATLAB, and Wolfram Alpha. Source input is generated by MATLAB commands as follows:

```
fileID = fopen(Input.txt','w');
fprintf(fileID,'%10.10f', rand(50));
fclose(fileID);
```

After the generation, the matrix is solved with another vector d by command **`x=A\b`** in MATLAB, **`linalg.solve(A, b)`** for NumPy, and input from the file to Wolfram Alpha.

For test results, first of all, execution speed is not a factor for this section. Thus, Wolfram Alpha can have enough time to return a much accurate (longer) result. The other results from NumPy and MATLAB are both identical, despite setting float or double. Since MATLAB is running on the host system which is 64bits while NumPy is running on the guest system with 32bits. It proved that NumPy and MATLAB can both handle extremes number near machine epsilon. However, it also proved that both solutions from NumPy and MATLAB are WRONG! The reason is Wolfram Alpha seems using some special algorithm to generate extremely accurate results while NumPy and MATLAB both implemented the traditional method to solve a linear system. Thus, theoretically the algorithm cannot compute an accurate result for a 50x50 matrix. It is tested that roughly 22x22 is the maximum size that both NumPy and MATLAB can generate the accurate result.

White-Box Testing

Please look at second source code for this part. We had some problem with installing SciPy between group members, and as a result I got little bit different version of SciPy compare to others

https://drive.google.com/open?id=0B_Y41SnYLCXGNmRuRGZxZ1dmN3c

White-Box Testing is one of the ways to analyze how efficient and effective the current test module is for the certain project. Unlike the black-box method which examines input specification, white-box methods examine source code to evaluate test data adequacy, and it often refers to percentage coverage of source code. To efficiently analyze coverage, it often generates control flow graphs from source code.

Control flow graphs are made by the combination of nodes and edges. Nodes represent routines like, for example, if statements or switch statements, and edges represent control flow between nodes.

In the white-box method, there is a number of ways how to determine coverage percentage, which is called the coverage technique. There are two basic structural coverages, which are Statement Coverage and Branch Coverage. Statement coverage only covers the true condition, so we are going use Branch Coverage instead to achieve better coverage. Branch coverage aims for every possible edge at each decision point (IF, Switch …) executed at least once. This ensures that every reachable point from the source code can be tested at least once.

Process

This will be the overall process for our group to analyze the Stats module in the SciPy Library by White-Box Method:

- We first need to fully understand the project we are using and this will be most time consuming step
    - To explain, we need to understand how software works to analyze it perfectly
- Create the control flow graphs by analyzing source code, so we can analyze how many routines and control flows this module contains by drawing nodes and edges.
    - We found that it is hard to find a Control Flow Graph Generating Tool for Python, so we skipped this part
- Run a test provided by the SciPy Library to stats module and generate a coverage test result
    - Test prints out both statement coverage and branch coverage
    - Code: `sudo python runtests.py --coverage -s stats`
    - There were a few skips and errors on the test (See Figure 1)
- Removed Statement coverage data and arranged branches into excel file
    - See Figure 2
- Moved excel file data into table including line number and solution
    - Manually looked into source code and put codes on table
    - Only first few lines were shown if lines of code were too long

- Figure 3

- Analyze Test Data Adequacy for the project and provide some sample test cases that can be inserted into test module

```
289, 1349-1354, 1438-1441, 1485-1486, 1504-1507, 1578-1580, 1637, 1697, 1738, 18
18, 1860, 1956-1994, 2001, 2071, 2076-2081, 2157-2162, 2242, 2312, 2327-2328, 23
58, 2421, 2483, 2539-2540, 2583-2586, 2654, 2700-2701, 2713-2714, 2721, 2752-275
3, 2773-2774, 2779, 2846-2847, 2853-2856, 2959, 3018, 3083, 3112, 3127, 3135, 31
57, 3169-3170, 3173, 3177-3180, 3302-3303, 3323-3329, 3413-3416, 3607-3610, 3689
, 3698, 3707, 3719, 3725-3728, 3788, 3892, 3908-3911, 3983, 3998-4001, 4111, 412
6, 4152-4162, 4182-4185, 4342-4343, 4393, 4507-4510, 4585-4586, 4590-4592, 4633,
 4648, 4655-4658, 4703-4706, 4772, 4807, 4823-4827, 4870, 4892, 4950, 4960, 4964
, 4972-5004, 5035, 5045-5134, 5170-5175, 5201-5207, 5231, 5236-5258, 5319, 462->
463, 464->466, 522->523, 575->586, 698->699, 1093->1096, 1113->1115, 1124->1126,
 1257->1263, 1335->1341, 1484->1485, 1572->1576, 2070->2071, 2311->2312, 2699->2
700, 2751->2752, 2765->2771, 3082->3083, 3111->3112, 3126->3127, 3134->3135, 314
5->3169, 3156->3157, 3172->3173, 3296->3309, 3301->3302, 3320->3323, 3891->3892,
 3982->3983, 4107->4111, 4125->4126, 4133->4138, 4138->-4001, 4142->-4001, 4341-
>4342, 4632->4633, 4643->4648, 4771->4772, 4806->4807, 4869->4870, 4949->4950, 4
956->4972, 4959->4960, 4963->4964, 5230->5231, 5318->5319
----------------------------------------------------------------------------
--
TOTAL                                    7478    2320    2018     253    71%
----------------------------------------------------------------------
Ran 2525 tests in 36.713s

FAILED (KNOWNFAIL=4, SKIP=61, errors=2)
```

Test Result (Figure 1)

Name	Total Branch	Branch Not Covered	Branch Coverage	Missing
_binned_statistic.py	50	7	86%	148->149, 246->247, 331->332, 349->350, 368->369, 374->378, 463->464
_constants.py	0	0	100%	
_continuous_distns.py	154	17	89%	170->174, 433->435, 475->476, 504->505, 553->555, 602->603, 605->606, 1814->1822, 1816->1827, 1914->1918, 1923->1924, 1925->1928, 2334->2336, 3497->3502, 3661->3662, 3692->3693, 4171-
_discrete_distns.py	24	1	96%	511->515
_distn_infrastructure.py	432	51	88%	>1101, 1102->1103, 1175->1177, 1201->1203, 1254->1255, 1457->1459, 1490->1491, 1498->1505, 1499->1500, 1505->-1445, 1523->1524, 1527->1533, 1529->1530, 1533->1540, 2011->2012, 2124->2129, 2136->2137, 2187->2188, 2213->2215, 2268->2272, 2409->2410, 2428->2429, 2432->2433, 2441->2442, 2455->2456, 2469->2472, 2475->2478, 2518->2519, 2673->2675, 2748->2749, 2754->-2661, 2756->2757, 3209->3210, 3262->3263
_distr_params.py	0	0	100%	
_multivariate.py	148	28	81%	28->42, 30->31, 50->51, 58->59, 66->67, 74->75, 137->138, 233->235, 1119->1120, 1127->1128, 1137->1138, 1139->1140, 1162->1163, 1167->1172, 1168->1169, 1178->1179, 1187->1189, 1343->1346, 1701->1704, 1705->1706, 1717->1718, 1720->1721, 1725->1726, 1727->1728, 1960->1963, 2035->2041, 2101->2102, 2103->2104
_stats_mstats_common.py	25	3	88%	87->88, 89->90, 191->192
_tukeylambda_stats.py	8	0	100%	
contingency.py	12	0	100%	
distributions.py	0	0	100%	
kde.py	46	9	80%	167->168, 196->197, 256->257, 258->259, 304->305, 334->335, 341->342, 369->370, 373->374
morestats.py	294	29	90%	131->132, 203->204, 304->309, 354->355, 398->399, 550->551, 559->560, 756->757, 872->873, 884->885, 1098->1099, 1185->1186, 1256->1257, 1264->1268, 1273->1274, 1276->1277, 1278->1279, 1380->1385, 1440->1441, 1590->1591, 1602->1603, 1631->1634, 1700->1707, 1702->1705, 1721->1729, 1970->1971, 2212->2214, 2331->2332, 2521->2522
mstats.py	0	0	100%	
mstats_basic.py	294	56	81%	93->94, 123->124, 206->207, 303->304, 462->463, 524->529, 529->530, 540->547, 551->566, 557->564, 672->677, 707->708, 728->733, 782->783, 1002->1005, 1046->1047, 1086->1089, 1096->1100, 1219->1222, 1225->1228, 1234->1235, 1241->1242, 1244->1245, 1379->1382, 1398->1401, 1478->1484, 1479->1482, 1484->1490, 1485->1488, 1497->1498, 1505->1508, 1506->1507, 1508->1512, 1509->1510, 1513->1516, 1637->1640, 1799->1805, 1800->1803, 1805->1811, 1806->1809, 1818->1819, 1824->1827, 1825->1826, 1827->1831, 1828->1829, 1833->1837, 1872->1874, 1968->1973, 2166->2167, 2238->2239, 2385->2386, 2387->2388, 2395->2396, 2405->2406, 2418->2419, 2499->2500
mstats_extras.py	40	6	85%	65->66, 94->95, 147->148, 166->169, 261->262, 267->270
stats.py	491	46	91%	462->463, 464->466, 522->523, 575->586, 698->699, 1093->1096, 1113->1115, 1124->1126, 1257->1263, 1335->1341, 1484->1485, 1572->1576, 2070->2071, 2311->2312, 2699->2700, 2751->2752, 2765->2771, 3082->3083, 3111->3112, 3126->3127, 3134->3135, 3145->3169, 3156->3157, 3172->3173, 3296->3309, 3301->3302, 3320->3323, 3891->3892, 3982->3983, 4107->4111, 4125->4126, 4133->4138, 4138->-4001, 4142->-4001, 4341->4342, 4632->4633, 4643->4648, 4771->4772, 4806->4807, 4869->4870, 4949->4950, 4956->4972, 4959->4960, 4963->4964, 5230->5231, 5318->5319
Total	2018	253	87%	

Excel Data (Figure 2)

_binned_statistic.py		
Branch Condition	**Line #**	**Statements to be executed**
if N != 1:	**148**	bins = [np.asarray(bins, float)]
if N != 1 and N != 2:	**246**	xedges = yedges = np.asarray(bins, float) bins = [xedges, yedges]
if not callable(statistic) and statistic not in known_stats:	**331**	raise ValueError('invalid statistic %r' % (statistic,))
if M != D:	**349**	raise AttributeError('The dimension of bins must be equal ' 'to the dimension of the sample x.')
for i in np.arange(len(smin)): - if smin[i] == smax[i]:	**368**	smin[i] = smin[i] - .5 smax[i] = smax[i] + .5
for i in np.arange(D): • if !(np.isscalar(bins[i])):	**377**	edges[i] = np.asarray(bins[i], float) nbin[i] = len(edges[i]) + 1 # +1 for outlier bins
if (result.shape != nbin - 2).any():	**463**	raise RuntimeError('Internal Shape Error')
_constants.py		
Branch Condition	**Line #**	**Statements to be executed**

Uncovered Branches (Figure 3)

Test Result table

- Some duplicates were removed from the table
- Some uncovered branches are removed due to uncertainty of test result
- Some Python files do not have any branches or any uncovered branches
- With default else conditions, I put Statements to be executed == N/A

Look at StatsUncoveredBranchList.xlsx for full table

Recommended Test Cases to Be Inserted

- Most uncovered branches are due to uncovered boundaries or uncovered cases -> Put certain values to test boundary into input

```
if n == 1: return kstat(data, n=2) * 1.0/N
In morestats.py
```

- Test for wrong data object to a function

```
if not isinstance(sparams, tuple): sparams = tuple(sparams)
In morestats.py
```

- Test for many cases for different number of elements in array or list

```
if N != 1 and N != 2:
      xedges = yedges = np.asarray(bins, float)
```

```
bins = [xedges, yedges]
In _binned_statistic.py
```

- Test for case that calls unimplemented method(function)

```
if not callable(statistic) and statistic not in known_stats:
      raise ValueError('invalid statistic %r' % (statistic,))
In _binned_statistic.py
```

- Include non-scalar value into number array or list

```
for i in np.arange(D):

      if !(np.isscalar(bins[i])): edges[i] = np.asarray(bins[i],
            float)
      nbin[i] = len(edges[i]) + 1  # +1 for outlier bins
In _binned_statistic.py
```

- Test if error handling works correctly

```
if ier != 1: raise FitSolverError(mesg=mesg)
In _continuous_distns.py
```

- Testing for optional parameters

```
if plot is not None:    756   plot.plot(svals, ppcc, 'x')
      …
In morestats.py
```

- There are also a few cases that cannot be tested since that branch is for defensive programming (Ensure value is inserted correctly when it should be)

```
if d != self.d:   196   if d == 1 and m == self.d:
      …
In kde.py
```

Conclusion

There were a total of 2018 branches for the stats module and 253 uncovered branches by testing.

```
(2018 – 253) / 2018 = 87.5%
```

Overall, 87.5% is decent percentage for white-box testing, so we can state that this project is well tested for branch coverage. However, there are still few test cases that can be inserted to eliminate uncovered branches. There are two common reasons to be uncovered, one is due to missing simple test conditions and the other is due to defensive programming that will not be executed forever. Mostly, uncovered branches can be covered by adding test cases for certain simple conditions.

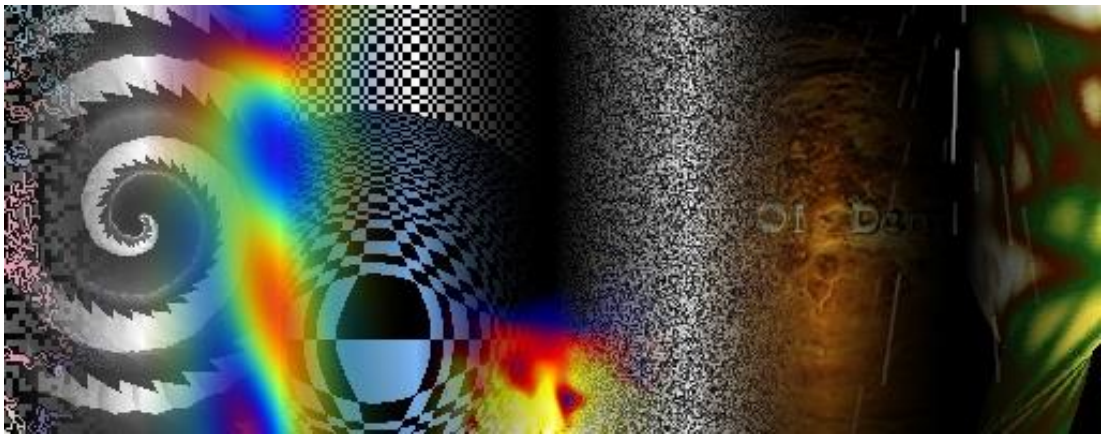
Failed Cases

This section is a summary of what efforts have been done but result in little conclusion. Testing on a large scale project may introduce some difficulties. Certain test methods are hard to apply to project such as SciPy when its application is hard to find. In addition, each team member are working on different parts as compared to previous assignments. Thus, even if similar tasks are assigned, individually members may still meet various difficulties.

Fuzzing Testing

Fuzzing testing has been deployed extensively for SciPy and its modules. However, little useful results are returned. The testing has two major concerns: First, the program being tested is better to have some input and output functionalities. Second, the tested part is better written in C style so it much likely encounters illegal memory access. However, since SciPy is kind of a special library in the scientific computation field, its application is much less than general C software. In addition, it is even more difficult to find a program which applies SciPy and has input output functions.

- NumPy Gumbo



Gumbo is a demo based on NumPy and PyGame which implemented various special effects utilize the library. It has been tested with AFL, PY-AFL in .PY, .PYC. EXE format. None of them successfully started the fuzzing.

- numpy.genfromtxt

```
s = StringIO("1, 1.3, abcde")
data = np.genfromtxt(s, dtype=[('myint','i8'),('myfloat','f8')
('mystring','S5')], delimiter=",")
```

genfromtxt is a ultimate function that reads data from any source of input. It is powerful for guessing the missing data and converts different incompatible data types. Thus, it will be a great example to fuzz since the program can recognize a wide range of input formats. However, it has tested a sample program that reads and displays the content of a TXT input as matrix on AFL, PY-AFL. Neither .PY, .PYC, .EXE can be fuzzed.