Understanding the r.watershed module for development of a testing suite using GRASS's testing framework and Python.

Stephanie Wendel*

GIS 582 601, North Carolina State University, Raleigh, United States

April 28th, 2015

Introduction

GRASS GIS's r.watershed module allows for users to perform analysis on an input elevation to find key hydrological parameters and Revised Universal Soil Loss Equation (RUSLE) factors. Specifically, the function can generate flow accumulation, drainage direction, the locations of streams and watershed basins, and LS and S factors (Ehlschlaeger 2014). This functionality has been applied in the Metz et al (2010) study that examined the extraction of streams using new implementations of the least-cost path search algorithm used in the r.watershed module. In making these improvements, test results were compared with the method before and after the changes. The results showed there was a 350 times faster processing speed in the newer implementation of the software (Metz et al. 2010, p 3220). However, additional smaller grain tests need to be developed to make sure the tool functions consistently before and after iterative changes to the software. Unit testing allows for a breakdown and testing of the software into its most basic code level in an isolated environment to see if the given conditions produce the expected results (Sale 2014, 2.1). Within the r.watershed module, there are many parameters that can be tested against to insure consistent output to insure they are performing correctly.

Exploring and creating tests during the development of code is an integral part of an agile software development. This method of development focuses on testing throughout the creation of a product which allows for flexibility in design and improved quality of the functions based on feedback given by the stakeholders (Sale 2014, 5.1). Instead of waiting for the product to be completed and then tested, agile software development calls for testing and feedback collection to be done from start to finish. The developers can make changes based on the requirements of the users before the project is completed to create a better final output. Specific tests can be added or removed based on relevance or adjustments that are done to the source code. In this process, manual tests and investigation need to be done to understand what the expected results should be for a given test. One form of testing is the unit test which breaks code functionality down to its most basic pieces to test for expected values. It is either true that it is the expected value or it is false.

One way that unit testing can be done on GRASS is by using the testing framework. The GRASS testing framework is term used to describe the system of writing and running tests within GRASS which uses the gunittest package (GRASS Development Team, 2015). This package is based on the Python unittest package but has been specialized to focus on GRASS specific needs for testing such as creation of GRASS-aware HTML test reports or testing processes that should not be influenced by a process termination caused by a C library function (GRASS Development Team, 2015). These tests allow for the developers to run these test cases before committing the updated code to the master copy. This provides feedback to the developer on whether there is a potential problems with the code base. It might not catch everything, but it is a way to insure there is not a loss of functionality with an update.

*Email: sawendel@ncsu.edu, Unity ID: sawendel

Objective

The main objective of this project is to build a test suite for the r.watershed module in GRASS using the GRASS testing framework. Through this process, I will explore unit testing within agile software development, how the GRASS testing framework works, and processes found within the r.watershed module. My personal goals of this project were to get involved with some Python scripting and to learn about GRASS GIS which I have not been previously exposed. The test developed for this project will be submitted to be included in GRSS for testing the module in the current and future versions of the software.

Data and study site

The main dataset used within the test suite is the Elevation raster which is part of GRASS's North Carolina sample data. This has an area of interested in South-west Wake County, NC at a resolution of 10m (boundary coordinates 35:48:34.6N-35:41:15.0N, 78:46:28.6W-78:36:29.9W). The projection is North Carolina State Plane Meters. The Elevation raster can be downloaded with GRASS as part of the optional sample data install and is placed in the folder nc_spm_08_grass7. This will allow for testing to be conducted by any user of the testing suite without requiring them to have their own data, download additional files, or for the test to attempt to generate it. These data sources could be used and could provide a more complex testing which encouraged by the testing framework developers (GRASS Development Team, 2015). The Elevation raster in particular covered a good size region which fit the needs for the input surface used to extract the basins and accumulation outputs.

Approach

The agile software development method was used as a guide to reach my end goal of a test for the r.watershed module. The point of using this method of development is to cycle through the process to continue to improve on the functionality of the software while getting feedback. The process required me to first experiment with and understand the r.watershed module. Next, I was able to start exploring the testing framework and how to set it up in a Python script. With the basic understanding of of what the r.watershed test suite might look like, it was time to consider how I would store and share my code to get feedback. To do this, I used GitHub, a code sharing and management platform. This allowed me to get some feedback and to document my bugs/enhancements as well as plan my project somewhat. I was able to then start my code and cycle back through this process of research-design-develop-feedback loop. The following sections go into more detail on the process.

Software setup

Setting up the testing environment was the first step. For my tests, I used a Windows 7 64-bit machine using a non-administrator user. This is one of the supported operating systems and by using a non-administrator I make sure there are no permission issues that a lower user might have. The testing suites are meant to work with any developer version of GRASS. I installed GRASS version 7.1.svn-64925-31 which was at the time a current daily developer build of the software. More details on the setup workflow can be found in *Appendix 1*. I ran into a few issues installing the software which are discussed in *Appendix 3*. I also maintained a 7.0 version of the software for my manual tests of the module as well as finding expected values. Since the software currently does not include this test in the source code, this will have to be run manually. In order to run this as a standalone test, a GRASS session must be started and a location and mapset setup for the test to work. For the complete steps on how to run a manual test see *Appendix 1*.

GitHub

Collaboration is an important part of the agile software development methods and GitHub provides a public space to work with code. This website allows users to maintain code, submit issues and pull other user's code to modify and push back to the original. Users could post feedback and report issues found in the test script. Also, I was able to log my own issues here to keep track of the changes I needed to make. It also allowed me some flexibility in planning out different phases of my script so I could setup goals for the next iteration of my test suite. My plan was to setup milestones to mark different versions of my script. Each one would have enhancements logged for the new functionality I would build along with bugs I found in the code. My script would then be updated to fix what I had added for the milestone and the changes then committed and synced back up to the master copy on GitHub.

Building the tests

In order to develop these tests, I focused first on exploring the GRASS, the r.watershed module, and the testing framework. This research helped me understand the key pieces that I needed to build the tests. For the module itself, I explored the documentation as well as tried some basic tests to see how the module worked such as seeing what different parameters did to the results. Once I had a general understanding of the module, I was able to select some of the parameters I wanted to test, such as if the outputs were made or if the drainage direction output was within the expected values.

With the basic ideas of the parameters I wanted to test, I started to form the basic structure required of the testing framework. The chosen language of the testing framework is Python. The basic structure of the script needs the gunittest package to be imported, a test case class setup, and the inclusion of basic testing methods. The within the script a test case class is created, this is a particular class to the testing framework that knows how to run the GRASS specific assertion functions. Within the class, there are basic methods to create the environment used within the tests such as setting up the region used or removing the rasters from the mapset after each test. These are called test fixtures and are the setUpClass, tearDownClass, and teardown methods. Multiple test classes can be built in one test file, but only if the testing environment needs to change. The only other major part of the test script is the method created within the class to provide the actual tests. These tests just need to start with 'test_' to run the method from the TestCase class that was built. These tests are the unit tests and should be simple, testing one basic concept. In order to test the module within these tests, an assert function is used that is built in to the testing framework. These functions check that some predicate is fulfilled. The assertModule method is used to run r.watershed and gives more control to the framework to show if it worked or failed. The outputs generated from running the module in this way can also be tested using assert methods such as assertRasterMinMax which tests that the expected min/max match the generated values provided. It is through these assert methods that the module can be evaluated and tested. Overall, the framework requires very little to setup to perform these tests; however, it does require knowledge of the testing framework and module that is being tested. The testing document “Testing GRASS GIS source code and modules”, found in the references provides a full explanation on this setup.

To write the tests, I went back and forth between research and manually testing my ideas to writing them out and running the script. Through my own testing, I was able to review and analyze my script to evaluate if the results and tests made sense. When I reached a milestone in my development based on the issues I submitted to GitHub, I would commit the scripts changes back to the master copy. Instead of planning specifically for each milestone at the beginning of my project, I added the enhancements and bugs as I developed to allow for flexibility in the plan.

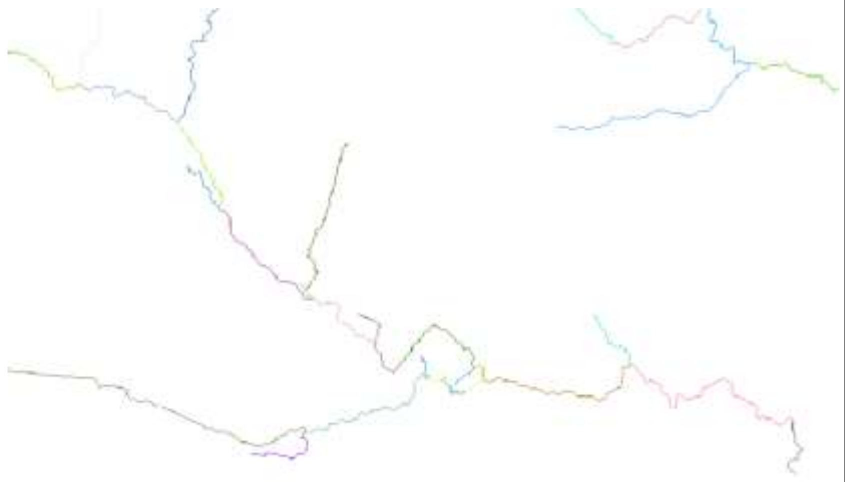
Once the test file is considered and incorporated into GRASS as a valid test suite, it can be run with all the other test files. Typically this will be done before committing the updates of the code to the master version.

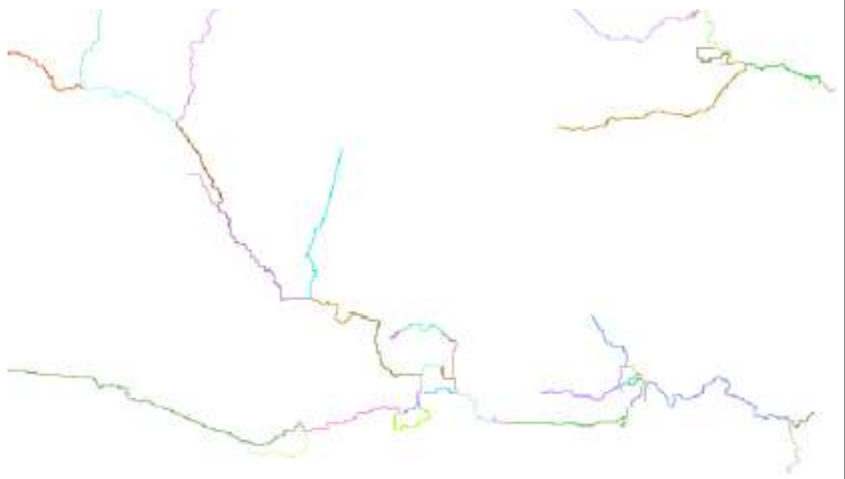
Results

There were four milestones for my project setup in GitHub. I created these as the version numbers of my tests. My first iteration of the test script included the basic setup of the testing framework and one unit test to see if the outputs were made. The second iteration added additional tests that are a little more complex and work more with the sample data, such as the test_fourFlag and test_watershedThresholdfail. My third iteration build the rest of the tests test_thresholdsize, test_drainageDirection, and test_basinValue. The last version was reserved for making final corrections and minor code changes. The final version consisted of six tests and 139 lines of code. The script and link to the GitHub page can be found in *Appendix 2*.

To get a basic understanding of working with the framework, I started with the test_OutputCreated() test. I created this test to evaluate if the output raster was generated from the module. I used the assertModule and assertRasterExists functions from within the framework to assess the outputs. After the final version, I found that I missed an output parameter in help. This parameter was overlooked but can be included in a future iteration.

Test_fourFlag was the second test built. It runs the r.watershed module twice, once with the 4 flag and once without. Running the module with the four flag, allows for only horizontal and vertical flow of water (Ehlschlaeger, C., 2014). The help also indicates that the stream and slope length outputs should be approximately the same as the ones run with diagonal flow included. To compare the outputs, the assertRasterNoDifference method was used. This ended up being the hardest test to decide if the values I provided should pass or fail. The assertRasterNoDifference method allows the user to apply a precision values calculated in the difference of the reference raster and the output. The outputs with the 4 flag were used as the actual and the ones without it were set as the reference. The method calculates the difference and passes if they are considered within the precision range different. The values I used for the precision were 100 for the streams and 10 for the slope length. Anything much less than these values would cause the test to fail. More investigation should be done by an expert to make sure these values make sense in this test. From the manual tests, the outputs do seem very similar. For example, in the south-west region of the outputs, there is some slight variation in the stream locations as seen in *Figure 1* and *2*. Changing the allowed flow direction changes the point at which these two branches connect. The statistics for both of the stream outputs can be seen in *Table 1* and *2*. The results are visually similar.

 *Figure 1 stream output without 4 flag*	`Table 1. stream r.univar results`
	
	`total null and non-null cells: 2025000`
	`total null cells: 2012344`
	`Of the non-null cells:`
	`----------------------`
	`n: 12656`
	`minimum: 2`
	`maximum: 256`
	`range: 254`
	`mean: 122.276`
	`mean of absolute values: 122.276`
	`standard deviation: 79.4923`
	`variance: 6319.03`
	`variation coefficient: 65.0105 %`
	`sum: 1547526`

 *Figure 2 stream output with 4 flag*	`Table 2. stream4 r.univar results`
	
	`total null and non-null cells: 2025000`
	`total null cells: 2006863`
	`Of the non-null cells:`
	`----------------------`
	`n: 18137`
	`minimum: 2`
	`maximum: 324`
	`range: 322`
	`mean: 149.257`
	`mean of absolute values: 149.257`
	`standard deviation: 102.576`
	`variance: 10521.8`
	`variation coefficient: 68.7245 %`
	`sum: 2707068`

The r.watershed module uses a threshold value to adjust the size of the basins and it is the minimum size of an exterior watershed basin (ones with one stream) in cells (Ehlschlaeger, C., 2014). Test_watershedThresholdfail tests whether the values provided in the threshold parameter meet the requirements of being an integer. Any negative value or zero will cause the tool to fail. Instead of running the assertModule method, I used the assertModuleFail to conduct this test. This method runs the module with the expectation that it will fail. If it fails, then the test is passed. The expectation is that a negative value or zero should cause the module to throw an error stating that the value must be an integer.

The test_thresholdsize function also works with the threshold parameter but this time it looks at the expected number of basin to be produced. I used the range of values from the manual tests in order to test to see if a particular size basin produce the minimum and maximum value and tested it with the assertRasterMinMax method. The first one uses 100000 as the threshold (*Figure 3*) and the second 10000 (*Figure 4*). The Univar module was used to find the min and max basin values and then used in the assertRasterMinMax to run the test. The values for 100k and 10k can be found in *Table 3* and *4* respectively. In the test, I do two threshold sizes, but this could be increased to make a wider test.

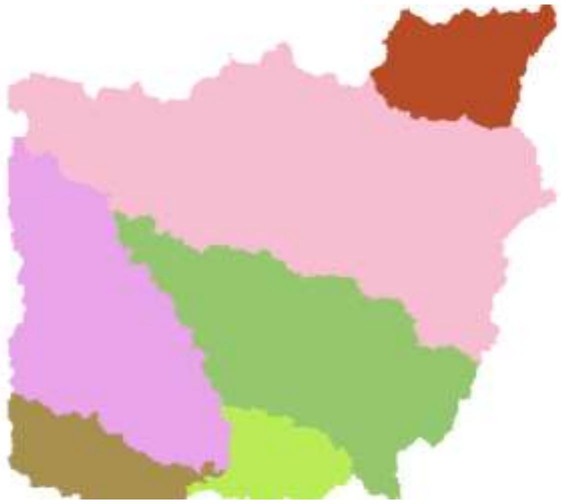


Figure 3 Basins with threshold of 100k

```
Table 3. Basin 100k Univar

total null and non-null cells: 2025000
total null cells: 470418
Of the non-null cells:
----------------------
n: 1554582
minimum: 2
maximum: 12
range: 10
mean: 7.06499
mean of absolute values: 7.06499
standard deviation: 3.57663
variance: 12.7923
variation coefficient: 50.6246 %
sum: 10983112
```

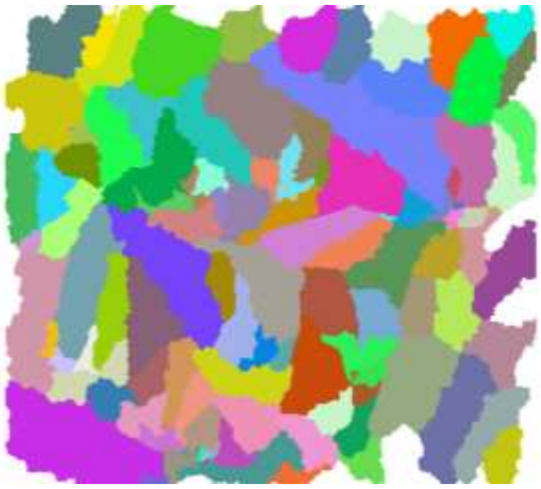


Figure 4 Basin with threshold of 10k

```
Table 4. Basin 10k Univar

total null and non-null cells: 2025000
total null cells: 145649
Of the non-null cells:
----------------------
n: 1879351
minimum: 2
maximum: 256
range: 254
mean: 123.411
mean of absolute values: 123.411
standard deviation: 83.5231
variance: 6976.1
variation coefficient: 67.6786 %
sum: 231933268
```

The fifth test was the test_drainageDirection. Similar to the threshold size test, I found the expected values using the Univar module by manually testing as seen in *Figure 5* and *Table 5*. The drainage output can be given in a range of -8 to 8. No matter the data, the values can be expected to fall within this range. According to the documentation, these values are multiplied by 45 to get a direction for a max of 360 (Ehlschlaeger, C., 2014). To make sure the output is within that range, the assertRasterMinMax method is utilized to test the output's range against these given values.

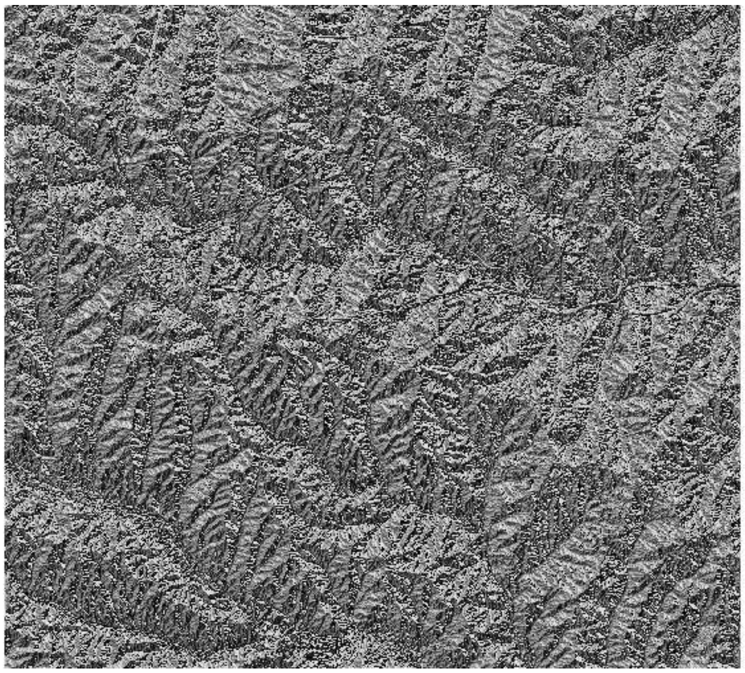


Figure 5 Drainage Output

```
Table 5. Drainage Univar

total null and non-null
cells: 2025000
total null cells: 0
Of the non-null cells:
----------------------
n: 2025000
minimum: -8
maximum: 8
range: 16
mean: 4.48374
mean of absolute values:
4.48515
standard deviation: 2.2943
variance: 5.26382
variation coefficient:
51.1694 %
sum: 9079565
```

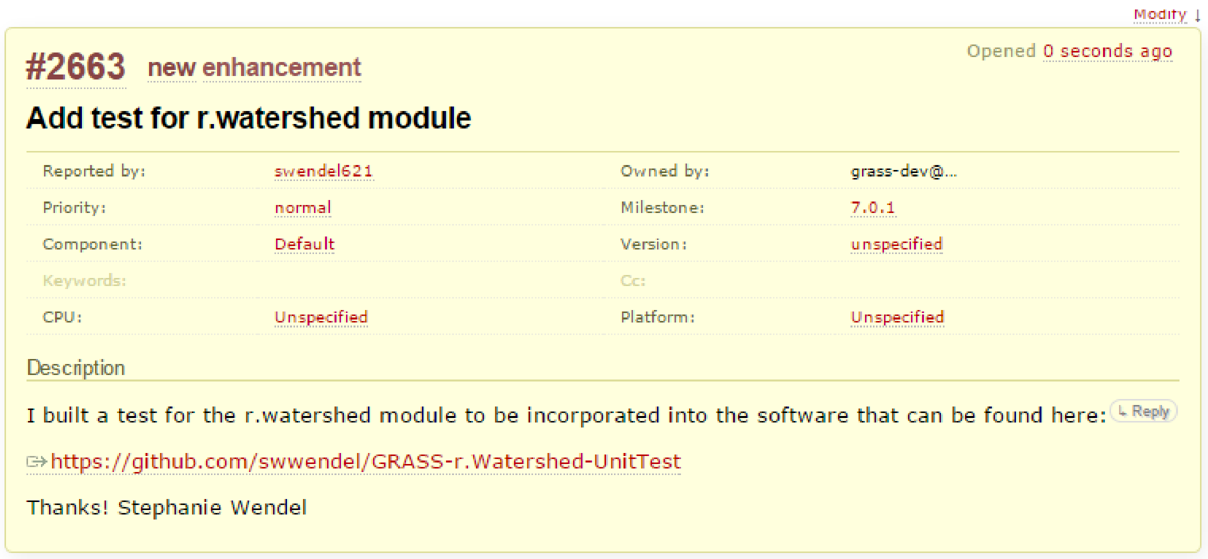
The final test is test_basinValue which makes sure the basin value is assigned a positive number. It basically just checks that the value is between 0 and a large number, in this case 1000000. In building this test, I found a potential documentation bug with the areas that were not made into a basin. The documentation set these cells to 0, but the tool set them to null. Null makes more sense within the output. More information can be found in Issue 2 of *Appendix 3*. By finding this in the development stage, the script was already useful in improving the software. A few more adjustments could be done to go a step further with the test and making sure the values are only even, positive numbers.

My results from running the tests can be seen in *Table 6*. I was able to get all six to pass, but as noted there is the possibility with changes to the precision that test_fourFlag could fail. The general feedback I noted for these modules can also be found in *Table 6*. In addition to evaluating myself, I received some feedback from Vaclav Petras, a GRASS developer and graduate PhD student at North Carolina State University. The suggestions he made showed me places in my code where I had some errors in my understanding of the testing framework's requirements such as unnecessary checks in my test_OutputCreated() function. In this function, I had an initial check to see if the maps already existed in the mapset. However, this was not needed as the tool would have failed because of no overwrite if they did. This function's focus was to look at the existence of the output and this would have change its purpose to be more along the lines of something used to test the overwrite parameter. I removed these lines of code to simplify the test. Another piece of feedback I received for this test was to make more informative messages in this function. I had setup the messages to report back the raster name that failed to be created but this actually could be confusing to the user or developer. It makes more logical sense for the message to report back the parameter that is not creating an output.

Table 6. Test functions with test result and feedback.

Test	**Success/Failure**	**Feedback**
test_OutputCreated	Success	Include output of TCI. New to version 7.0.
test_fourFlag	Both	Needs to be evaluated by expert to see if the precision values make sense in the test.
test_watershedThresholdfail	Success	Test the messages produce when it fails
test_thresholdsize	Success	More threshold sizes included in the test
test_drainageDirection	Success	Test more raster inputs
test_basinValue	Success	Test that values are only even numbers.

At the end of my development, I was able to contribute to GRASS GIS by submitting my project's code as an Enhancement to be included into a future build of the software (*Figure 6*). It has been already incorporated and works within the test suite. Others can use the test from this project as a starting place to develop further tests of the r.watershed module.



Modify ↓

#2663 new enhancement

Opened 0 seconds ago

Add test for r.watershed module

Reported by:	swendel621	Owned by:	grass-dev@…
Priority:	normal	Milestone:	7.0.1
Component:	Default	Version:	unspecified
Keywords:		Cc:	
CPU:	Unspecified	Platform:	Unspecified

Description

I built a test for the r.watershed module to be incorporated into the software that can be found here:

https://github.com/swwendel/GRASS-r.Watershed-UnitTest

Thanks! Stephanie Wendel

Figure 6 Ticket submitted to add the test I created to GRASS.

Discussion

My participation in agile software development was twofold. First, in building the test suite, I was able to contribute to GRASS development by providing a piece of code that can be used in this version and future ones to test the quality of the r.watershed module. By providing my code, I have increased the quality of the software and provided something that can be further developed and improved based on stakeholder needs. Secondly, I applied the agile development methodology to my own development of the test suite script. I did this through my milestones and building the small number of focuses tests for each one. Throughout the development, I tested my code and had it available for feedback. One problem I had was that I had a tendency to develop my code in big chunks without always getting the feedback I needed for proceeding on to additional tests. In terms of my project, I relied heavily on the research and tests I did to understand the r.watershed module instead of working within an expert or developer. I did have some feedback but it could have sought out more from different stakeholders. Without some of the time constraints I had, this would have been beneficial to my process. I also could have documented my changes and improvements more. I found myself focusing more on writing the actual code than considering the ways I was changing it or my thought process. While agile software development does not always focus on heavy documentation as an important element to the process, I could have improved my commit messages to be more explanatory in what was done for that commit to the master branch in GitHub. I could have also committed more often instead of waiting until I felt like I had completed the milestone.

Conclusion

This project gave me the opportunity to learn about the processes of agile software development by exploring GitHub, the GRASS testing framework, and unit testing within Python. Developing the tests scripts using the testing framework was easy to do given there is some knowledge of the methods of the framework and working with Python. The framework will make it easier to incorporated additional tests such as this one into GRASS GIS to provide the additional testing on each iteration of the code.

Future Work

Additional considerations for future work focus on improving on the tests for the r.watershed module. There are many more tests that could be done such as testing for more expected values from the outputs, the memory parameter to see if it is consuming the correct amount, and the error messages produced. In addition, random data could be used or even another dataset to give a wider spread of tests. There are a number of different assert methods I did not use from the testing frame work that might test things better or would show more important comparisons. Additional feedback from the GRASS development team should be acquired to see what might be the most important parts to test as well as to test for new functionality that they might be incorporating into the latest builds.

Acknowledgements

Special thanks to Vaclav Petras for the guidance provided in understanding the GRASS testing framework and for providing feedback on my tests.

References

Carpendter, T.M, Sperfslage, J.A., Georgakakos, K.P., Sweeney, T. and Fread, D.L., 1999. National threshold runoff estimation utilizing GIS in support of operational flash flood warning systems. Journal of Hydrology, 224 (1-2), 21-44.

Ehlschlaeger, C., 2014. r.watershed [online]. GRASS GIS 7.0.0svn Reference Manual. Available from: http://grass.osgeo.org/grass70/manuals/r.watershed.html [Accessed 12 February 2015].

GRASS Development Team, 2015. Testing GRASS GIS source code and modules [online]. GRASS GIS 7.1.svn Reference Manual. Available from: http://grass.osgeo.org/grass71/manuals/libpython/gunittest_testing.html [accessed 12 February 2015].

Metz, M., Mitasova, H. and Harmon, R.S. 2010. Accurate stream extraction from large, radar-based elevation models. Hydrology & Earth System Sciences Discussions, 7 (3), 3213-3235.

Neteler, M. and Mitasova, H., 2008. Open Source GIS: a GRASS GIS Approach. New York: Springer.

Sale, D., 2014. Testing Python, Applying unit testing TDD, BDD, and Acceptance testing. West Sussex: Wiley. Kindle Fire.

Appendix 1: Setting up GRASS for testing

Installation

The following outlines the steps needed to install a GRASS developer edition on windows and run this test.

On a windows machine, navigate to the following link to the OSGeo project site for GRASS:

http://grass.osgeo.org/download/software/ms-windows/

Find the header that says it has the development, daily builds. At the time of this project, the development builds were for GRASS GIS 7.1. Click the download link. This will take you to the current list of daily builds for that version. Choose one and download the executable.

Once downloaded, double click on the executable to install. Follow the step by step instructions for installation and make sure to install the North Carolina Sample data.

Running the Test Script

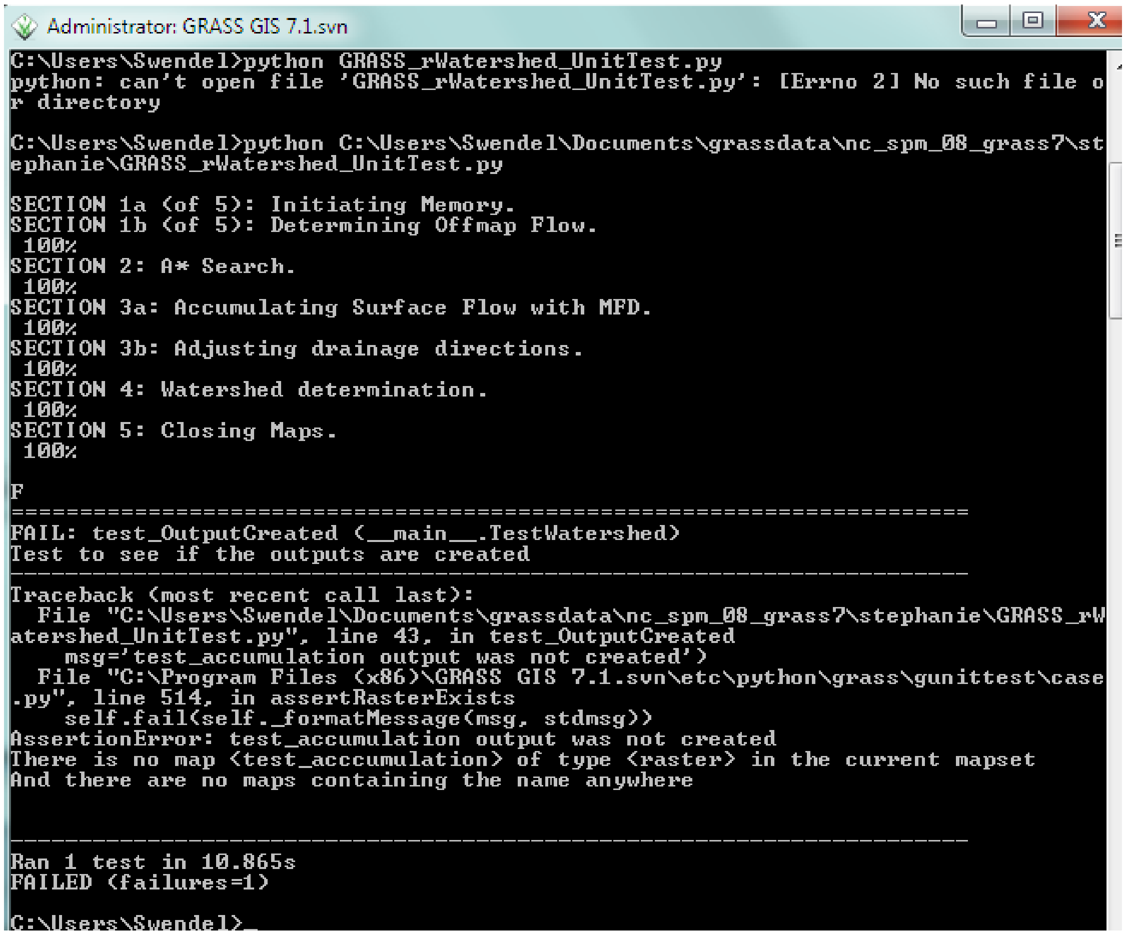
Once installed, double click on the GRASS Icon or find it in the Start Menu to launch the application.

In the Startup dialog, find the database directory where the North_Carolina sample data is located. Next set the GRASS location to the North_Carolina folder. Create a new mapset and click 'Start GRASS session'. The GUI will open, but this can be ignored. The session is needed to run the test as a standalone file.

Switch over to the Command Line view of GRASS. Type 'python' followed by the location to the .py file that contains the tests. Such as

```
python D:\HomeWork\Github\GRASS-r.Watershed-UnitTest\GRASS_rWatershed_UnitTest.py
```

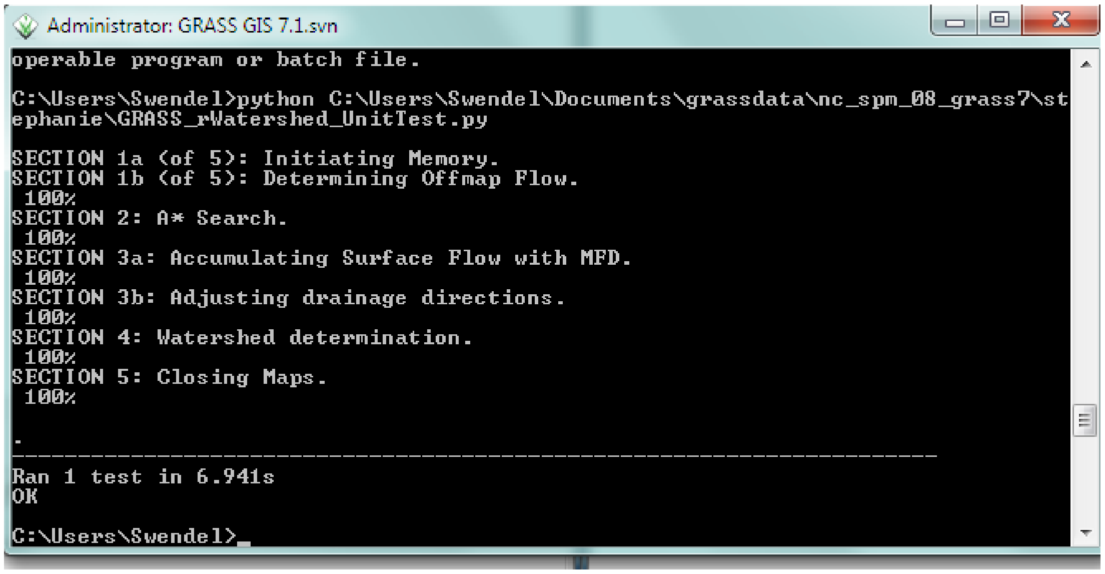
Hit enter. The test will run. The tool messaging will appear in the command line window. It will show text that may be seen if the tool was ran within the GUI. At the end it will show how many tests were successful or failed or if there were errors. Some examples are listed below in *Figures 7-9*. This will help to determine problems in the r.watershed module that need to be looked into further by a developer of the module. Notice in *Figure 9*, the messaging tells the user which test failed. In this example's case, it was the test_OutputCreated test. The testing framework also has the ability to provide some basic messaging about what might be the problem. For example, "There is no mapset <test_accumulation> of type <raster> in the current mapset.



Administrator: GRASS GIS 7.1.svn

```
C:\Users\Swendel>python GRASS_rWatershed_UnitTest.py
python: can't open file 'GRASS_rWatershed_UnitTest.py': [Errno 2] No such file o
r directory

C:\Users\Swendel>python C:\Users\Swendel\Documents\grassdata\nc_spm_08_grass7\st
ephanie\GRASS_rWatershed_UnitTest.py

SECTION 1a (of 5): Initiating Memory.
SECTION 1b (of 5): Determining Offmap Flow.
 100%
SECTION 2: A* Search.
 100%
SECTION 3a: Accumulating Surface Flow with MFD.
 100%
SECTION 3b: Adjusting drainage directions.
 100%
SECTION 4: Watershed determination.
 100%
SECTION 5: Closing Maps.
 100%

F
======================================================================
FAIL: test_OutputCreated (__main__.TestWatershed)
Test to see if the outputs are created
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Swendel\Documents\grassdata\nc_spm_08_grass7\stephanie\GRASS_rW
atershed_UnitTest.py", line 43, in test_OutputCreated
    msg='test_accumulation output was not created')
  File "C:\Program Files (x86)\GRASS GIS 7.1.svn\etc\python\grass\gunittest\case
.py", line 514, in assertRasterExists
    self.fail(self._formatMessage(msg, stdmsg))
AssertionError: test_accumulation output was not created
There is no map <test_acccumulation> of type <raster> in the current mapset
And there are no maps containing the name anywhere


----------------------------------------------------------------------
Ran 1 test in 10.865s
FAILED (failures=1)

C:\Users\Swendel>_
```

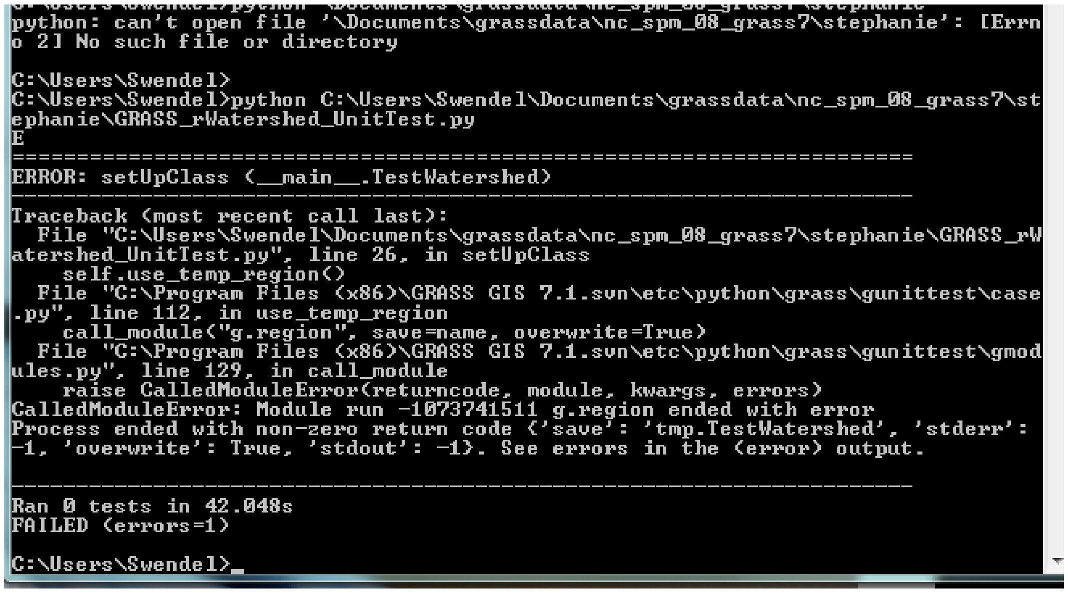
Figure 7 A test failure within the command line interface.



Administrator: GRASS GIS 7.1.svn

```
operable program or batch file.

C:\Users\Swendel>python C:\Users\Swendel\Documents\grassdata\nc_spm_08_grass7\st
ephanie\GRASS_rWatershed_UnitTest.py

SECTION 1a (of 5): Initiating Memory.
SECTION 1b (of 5): Determining Offmap Flow.
 100%
SECTION 2: A* Search.
 100%
SECTION 3a: Accumulating Surface Flow with MFD.
 100%
SECTION 3b: Adjusting drainage directions.
 100%
SECTION 4: Watershed determination.
 100%
SECTION 5: Closing Maps.
 100%

.
----------------------------------------------------------------------
Ran 1 test in 6.941s
OK

C:\Users\Swendel>_
```

Figure 8 A test success within the command line interface.



```
python: can't open file '\Documents\grassdata\nc_spm_08_grass7\stephanie': [Errn
o 2] No such file or directory

C:\Users\Swendel>
C:\Users\Swendel>python C:\Users\Swendel\Documents\grassdata\nc_spm_08_grass7\st
ephanie\GRASS_rWatershed_UnitTest.py
E
======================================================================
ERROR: setUpClass (__main__.TestWatershed)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Swendel\Documents\grassdata\nc_spm_08_grass7\stephanie\GRASS_rW
atershed_UnitTest.py", line 26, in setUpClass
    self.use_temp_region()
  File "C:\Program Files (x86)\GRASS GIS 7.1.svn\etc\python\grass\gunittest\case
.py", line 112, in use_temp_region
    call_module("g.region", save=name, overwrite=True)
  File "C:\Program Files (x86)\GRASS GIS 7.1.svn\etc\python\grass\gunittest\gmod
ules.py", line 129, in call_module
    raise CalledModuleError(returncode, module, kwargs, errors)
CalledModuleError: Module run -1073741511 g.region ended with error
Process ended with non-zero return code {'save': 'tmp.TestWatershed', 'stderr':
-1, 'overwrite': True, 'stdout': -1}. See errors in the (error) output.

----------------------------------------------------------------------
Ran 0 tests in 42.048s
FAILED (errors=1)

C:\Users\Swendel>_
```

Figure 9 Error in a test within the command line interface.

Appendix 2: The script

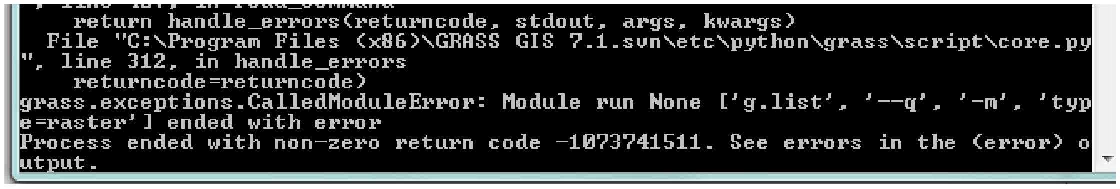
The Github version of this script can be found here: https://github.com/swwendel/GRASS-r.Watershed-UnitTest. Bugs and enhancements found within the different tests and general progress in the code can be tracked through the milestones and issues section.

```
"""
Name:        GRASS_rWatershed_UnitTest
Purpose:     This script is to demonstrate a unit test for GRASS's r.Watershed
             module for GIS582 at NCSU. This Unit Test must be placed in the
             directory testsuite of the GRASS installation.

Author:      Stephanie Wendel - sawendel
GRASS:       7.1.svn-r665096-56
Version:     1.4
Modified:    4/20/2015
Copyright:   (c) sawendel 2015
Licence:     GNU GPL
"""

# import grass testing module gunittest
import grass.gunittest
from grass.gunittest import TestCase, test

#test case for watershed module which is derived from
grass.gunittest.TestCase
class TestWatershed(grass.gunittest.TestCase):

    #Setup variables to be used for outputs
    accumulation ='test_accumulation'
    drainage ='test_drainage'
    basin ='test_basin'
    stream ='test_stream'
    halfbasin ='test_halfbasin'
    slopelength='test_slopelength'
    slopesteepness = 'test_slopesteepness'
    elevation = 'elevation'

    @classmethod
    def setUpClass(cls):
        """Ensures expected computational region and setup"""
        #Always use the computational region of the raster elevation
        cls.use_temp_region()
        cls.runModule('g.region', raster=cls.elevation)

    @classmethod
    def tearDownClass(cls):
        """Remove the temporary region"""
        cls.del_temp_region()

    def tearDown(cls):
        """Remove the outputs created from the watershed module after each
test
        is run."""
        cls.runModule('g.remove', flags='f', type='raster',

name='{0},{1},{2},{3},{4},{5},{6},{7},{8}'.format(cls.accumulation,
            cls.drainage, cls.basin, cls.stream, cls.halfbasin,
            cls.slopelength, cls.slopesteepness, 'test_lengthslope_4',
            'test_stream_4'))

    def test_OutputCreated(self):
        """Test to see if the outputs are created"""
        #run the watershed module
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='10000', accumulation=self.accumulation,
            drainage=self.drainage, basin=self.basin, stream=self.stream,
            half_basin=self.halfbasin, length_slope=self.slopelength,
            slope_steepness=self.slopesteepness)
        #check to see if accumulation output is in mapset
        self.assertRasterExists(self.accumulation,
            msg='accumulation output was not created')
        #check to see if drainage output is in mapset
        self.assertRasterExists(self.drainage,
            msg='drainage output was not created')
        #check to see if basin output is in mapset
        self.assertRasterExists(self.basin,
            msg='basin output was not created')
        #check to see if stream output is in mapset
        self.assertRasterExists(self.stream,
            msg='stream output was not created')
        #check to see if half.basin output is in mapset
        self.assertRasterExists(self.halfbasin,
            msg='half.basin output was not created')
        #check to see if length.slope output is in mapset
        self.assertRasterExists(self.slopelength,
            msg='length.slope output was not created')
        #check to see if slope.steepness output is in mapset
        self.assertRasterExists(self.slopesteepness,
            msg='slope.steepness output was not created')

    def test_fourFlag(self):
        """Test the -4 flag to see if the stream and slope lengths are
        approximately the same as the outputs from the default module run"""
        #Run module with default settings
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='10000', stream=self.stream,
            length_slope=self.slopelength, overwrite=True)
        #Run module with flag 4
        self.assertModule('r.watershed', flags='4', elevation='elevation',
            threshold='10000', stream='test_stream_4',
            length_slope='test_lengthslope_4')
        #Use the assertRastersNoDifference with precsion 100 to see if close
        #Compare stream output
        self.assertRastersNoDifference('test_stream_4', self.stream, 100)
        #Compare length_slope output
        self.assertRastersNoDifference('test_lengthslope_4',
            self.slopelength, 10)

    def test_watershedThreadholdfail(self):
        """Check to see if it will allow for a threshold of 0 or a
negative"""
        self.assertModuleFail('r.watershed', elevation=self.elevation,
            threshold='0', stream=self.stream, overwrite=True,
            msg='Threshold value of 0 considered valid.')
        self.assertModuleFail('r.watershed', elevation=self.elevation,
            threshold='-1', stream=self.stream, overwrite=True,
            msg='Threshold value of 0 considered valid.')

    def test_thresholdsize(self):
        """Check to see if the basin output is within the range of values
        expected"""
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='100000', basin=self.basin, overwrite=True)
        # it is expected that 100k Threshold has a min=2 and max=12 for this
data
        self.assertRasterMinMax(self.basin, 2, 12)
        # it is expected that 100k Threshold has a min=2 and max=256 for this
data
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='10000', basin=self.basin, overwrite=True)
        self.assertRasterMinMax(self.basin, 2, 256)

    def test_drainageDirection(self):
        """Check to see if the drainage direction is between -8 and 8."""
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='100000', drainage=self.drainage)
        #Make sure the min/max is between -8 and 8
        self.assertRasterMinMax(self.drainage, -8, 8,
            msg='Direction must be between -8 and 8')
    def test_basinValue(self):
        """Check to see if the basin value is 0 or greater"""
        self.assertModule('r.watershed', elevation=self.elevation,
            threshold='10000', basin=self.basin)
        #Make sure the minimum value is 0 for basin value representing unique
positive integer.
        self.assertRasterMinMax(self.basin, 0, 1000000,
            msg='A basin value is less than 0 or greater than 1000000')

if __name__ == '__main__':
    test()
```

Appendix 3: Issues

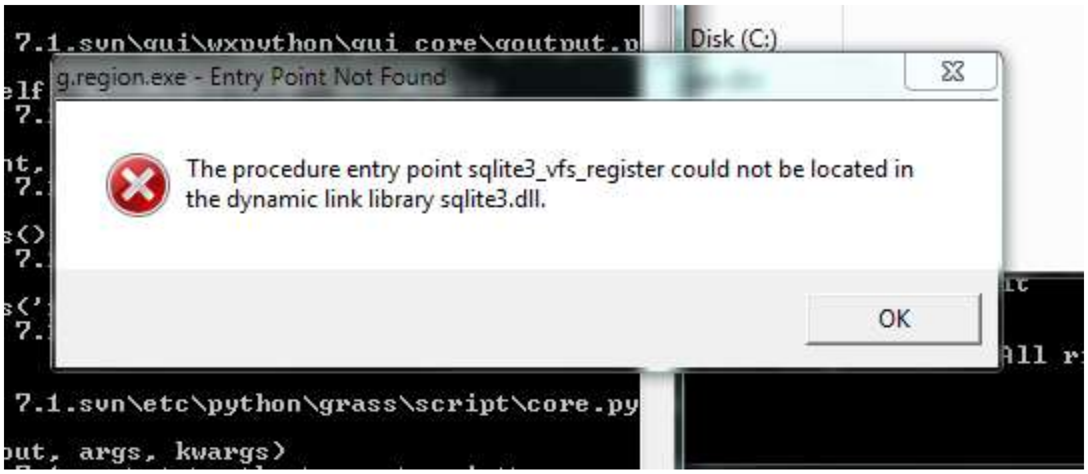
Issue 1:

I had some problems with this installation when I tried to start my GRASS session. It failed to load and the process ended with a non-zero return code -1073741511 (*Figure 10)* and a message box saying "The procedure entry point sqIite3 register could not be located in the dynamic link library sqIite3.dll" (*Figure 11*).



```
    return handle_errors(returncode, stdout, args, kwargs)
  File "C:\Program Files (x86)\GRASS GIS 7.1.svn\etc\python\grass\script\core.py
", line 312, in handle_errors
    returncode=returncode)
grass.exceptions.CalledModuleError: Module run None ['g.list', '--q', '-m', 'typ
e=raster'] ended with error
Process ended with non-zero return code -1073741511. See errors in the (error) o
utput.
```

Figure 10 Error -1073741511 in command line interface.



g.region.exe - Entry Point Not Found

The procedure entry point sqlite3_vfs_register could not be located in the dynamic link library sqlite3.dll.

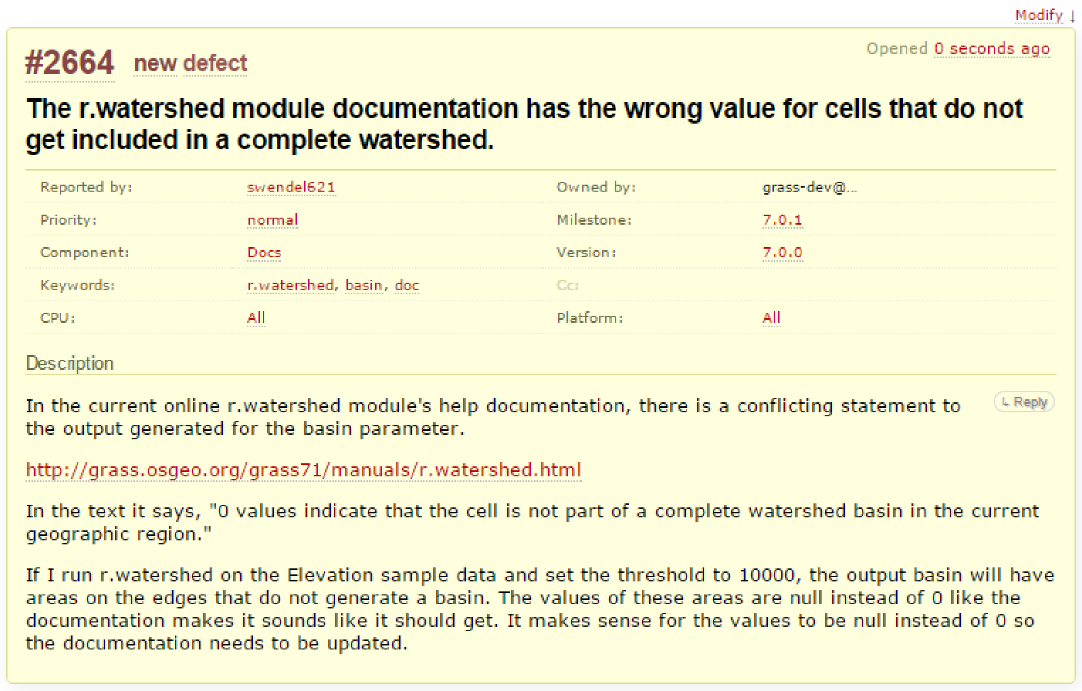
OK
Disk (C:)
7.1.svn\gui\wxpython\gui_core\goutput.p
7.1.svn\etc\python\grass\script\core.py
put, args, kwargs)

Figure 11 Popup message showing the sqlite3.dll warning.

I uninstalled this version and attempted to run the 7.1.svn-64925-38 build instead which also failed with the same error. It also gives a warning that no addons metadata file available. In working with Vaclav Petras, he identified that the issue was related to a sqlite3.dll conflict which can be found in more detail on this page: http://grasswiki.osgeo.org/wiki/WinGRASS_errors. I renamed the other versions of sqlite3.dll I had on this machine and the software worked as expected.

Issue 2:

In testing the basin output, I found that the documentation did not match the results I was seeing. It says, “0 values indicate that the cell is not part of a complete watershed basin in the current geographic region.” When I did my manual tests of r.watershed these areas were given a value of null instead of 0. The range can also be seen to start at 2 instead of 0. I consulted with Vaclav Petras and he suggested the documentation may be wrong. It makes more sense for the value to be null instead of zero. I created an account on http://trac.osgeo.org/grass/ and submitted a ticket for this to be investigated further. The ticket can be seen in *Figure 12*.



Modify ↓

#2664 new defect

Opened 0 seconds ago

The r.watershed module documentation has the wrong value for cells that do not get included in a complete watershed.

Reported by:	swendel621	Owned by:	grass-dev@...
Priority:	normal	Milestone:	7.0.1
Component:	Docs	Version:	7.0.0
Keywords:	r.watershed, basin, doc	Cc:	
CPU:	All	Platform:	All

Description

Reply

In the current online r.watershed module's help documentation, there is a conflicting statement to the output generated for the basin parameter.

http://grass.osgeo.org/grass71/manuals/r.watershed.html

In the text it says, "0 values indicate that the cell is not part of a complete watershed basin in the current geographic region."

If I run r.watershed on the Elevation sample data and set the threshold to 10000, the output basin will have areas on the edges that do not generate a basin. The values of these areas are null instead of 0 like the documentation makes it sounds like it should get. It makes sense for the values to be null instead of 0 so the documentation needs to be updated.

Figure 12 Ticket 2664 showing doc bug submitted to OSGeo.