Software Engineering with R

(early draft)

Panos Toulis
ptoulis@fas.harvard.edu

Harvard University
Department of Statistics

September 5, 2014

1 Introduction

Software engineering is the art of good software. Key objectives are ensuring that the code is **correct, efficient and readable**. In statistical computing, the programming task is twofold. First, the code has to be good in a statistical sense: is the chain mixing well? Is the likelihood function correct? Do the intervals have the expected coverage? Second, the code has to be good in a software-engineering sense: is the program structured well? Do the function/variable names make sense? How easy is it to extend the code? Is it fast or can it be improved? Is it well-documented? Unfortunately, the typical training of a statistician will only cover the former. The purpose of this document is to help with the latter.

1.1 Intended Audience

This document is primarily intended for the class STAT 221 at Harvard (Statistical Computing), but it could have a larger scope. Specifically, this document might be useful to those who have not encountered software engineering practices before, either statisticians or new programmers, but spend a lot of time programming with `R` (or will do so in the future). Note that although we are focusing on `R` the engineering practices presented here, exist for all languages.

1.2 Overview (Correctness, efficiency and readability)

Ok, let's take a quick tour before we delve into details. As mentioned before there are three main goals with software engineering: correctness, efficiency and readability.

A program is **correct** when it is doing what it is supposed to do. For that reason, *unit testing* is a framework of defining tests for code units that need to "pass" before the code is deemed correct. This is presented in Section 2. If we devise good tests and we get errors, then we will need to do some *debugging*. The main idea (usually) is to stop the execution when the error happens and then "browse" the state of the variables and look for the cause of the error. This process can be complemented by *logging*, which is usually a facility to print things (e.g. values of variables) on file or the console. The main convenience with logging comes with the option to enable/disable it at will, saving to file and choosing the log format (e.g. include date, timestamp, code lines etc).

A code is **efficient** when it is fast and is using the minimum of resources (memory, cpu and disk). Languages like`R` are especially sensitivite on how algorithms are implemented. For example, vectorization in `R` is usually orders-of-magnitude faster than other approaches. We will see an example of that in Section 5. In many cases however, our first implementation will be slow. In these cases, *profiling* is a great tool to see what's going wrong. A *profiler* will record the execution time of functions and measure running times and resource usage. `R` ships with a standard profiler which is very easy to use.

Last but not least, we need to make the code **readable**. The code should be written in a way that an external programmer would be able to understand the basics of its functionality and implementation, just by looking at it. For that reason, *code style* is critical, yet usually ignored. However, "good style" is to a large extent, a matter of personal taste, yet there are generally acceptable practices that it would be good to follow.

1.3 Working example

For clarity, we will work on a simple coding example. Assume that our goal is to write a function that will calculate the L2 norm of a vector x, i.e. $||x|| = \sqrt{\sum_i x_i^2}$. Some of these functions are totally inefficient and all of them contain errors. Let's see how we can use software engineering tools to solve our problems

```
# norms.R
# Functions to compute the L-2 norm of a column vector
# ||x|| = sqrt(x_1^2 + x_2^2 + ...)

# First implementation: Iterates over all elements
normOne <- function(x) {
    n = length(x)
    sum = 0
    for(i in 1:n)
        sum = sum - x[i]^2
    return(sqrt(sum))
}

# Second implementation: recursive definition
normTwo <- function(x) {
   n = length(x)
   if(n==1)
       return(abs(x[1]))
   sqrt(normTwo(head(x, n-1))^2 + x[n]^2)
}

# Third implementation: Uses vector multiplication x'x
normThree <- function(x) {
    sqrt(t(x) %*% xx)
}
```

At this moment, open up R and copy-paste the following script (assume this is our current state of the code base).

2 Unit testing

Unit testing bears similarities with statistical testing. In unit testing, we focus on a function or even a set of functions (e.g. an entire script), and then make a null hypothesis that "the code is correct". We then devise a series of code checks/test (unit test) that, if passed, will be evidence that the code is correct. As in statistical testing, we need to come up with a limited number of tests that will reject the null when it is false (i.e. uncover the bug if the code is incorrect).

The first decision here is what part of the code we will test, and usually we test as much of the code as we can. Testing only important or complicated functions is also an option, although personally I think all functions are important and need be tested (after all, errors happen more frequently when we are confident about things). In our working example, we choose to test all three functions separately.

The second decision is the actual tests. Assume that function $f(x)$ computes the L2 norm of the vector x. Then, what is a series of tests that $f(x)$ should pass? There is *no unique* answer to that question. For example, we choose to test the following:

- $f(x) == 0$, if $x == ()$, the empty vector
- $f(x) == 5$, if $x == (3, 4)'$
- $f(2x) == 2f(x)$ for a random vector x

The first two tests are *specific tests* i.e. we choose inputs for which we know the answer and check if our implementation gives the same answer. Specific tests are very useful as sanity checks, however they are a bit restricted as they focus on specific cases. The last check is a *generic test* as it tests a *property* of L2 norm functions, namely multiplication by scalar (scalability). Are there more tests we could make? Certainly? For example, we could check the triangle inequality by checking that $f(x + y) \leq f(x) + f(y)$ for random vectors x, y, check other inputs etc. It will always be up to you to devise a suite of tests that will be *complete*.

At this point, copy-paste the following testing code at the end of your file.

```
# Function that tests a "norm" function
norm.checks <- function(norm) {
  print("Starting tests..")
  checkTrue(is_near(norm(c()), 0))
  checkTrue(is_near(norm(c(3,4)), 5))
  x = rnorm(20)
  checkTrue(is_near(norm(2 * x), 2 * norm(x), tol=1e-6))
}

is_near <- function(x, x0, tol=1e-9) {
  return(abs(x-x0) < tol)
}
```

These implement the *checks* that comprise the test defined above. More about *checking* in the following section.

2.1 Checking

Function `checkTrue` will simply check if the argument is true. This is defined in the `RUnit` library. Several others are defined (not used here) like `checkEquals`, `checkException` and so on. Check the documentation for more.

In general, checking is the building block of code testing (however checking does not need part of a test). Checking mostly makes sure that the underlying values or structures are what they are supposed to be. One can imagine writing many different `CHECK_*` functions to check for different things. Look here for some more examples[1] I wrote for some frequently-used checks.

2.2 R tools for testing

One popular tool for unit testing in R is the `RUnit` package[2]. The idea is pretty simple:

1. Define the checks you want to do. Group them logically (in our case it is `norm.checks`) and based on what kind of code functionality they check (in our case they are algebraic checks for norm functions)

2. Create the testing code. These are functions that are named `test.*`. These functions can be spread on several files

[1]`https://github.com/ptoulis/r-toolkit/blob/master/checks.R`
[2]Install by running `install.packages("RUnit")`

3. Define a testing suite, as a group of .R scripts that contain testing code

In our case we are going to put both the testing functions and the testing suite in the same place. Since we have written the checking code, it is time to write the testing code. Copy-paste the following at the end of your script:

```
test.norm.one <- function() {
  norm.checks(normOne)
}
test.norm.two <- function() {
  norm.checks(normTwo)
}
test.norm.three <- function() {
  norm.checks(normThree)
}
```

The setup could be more efficient but this is ok too (makes it clear what functions we are testing, and which checks we are using). Now add the `RUnit` code that will use the testing code. At this moment install `RUnit` by:

```
> install.packages("RUnit")
```

Then paste the following at the end of the file.

```
run.tests <- function() {
  test.suite <- defineTestSuite(name="norm", dirs=".", testFileRegexp="norms.R")
  test.result <- runTestSuite(test.suite)
  printTextProtocol(test.result)
}
```

Now add `library(RUnit)` **at the top** of your script. By now, your script should like the one in `A.1`. Then run `run.tests()` to run your R unit tests! If everything is OK, you will see that all functions contain errors like the following:

```
> run.tests()
Executing test function test.norm.one  ... [1] "Starting tests.."
Timing stopped at: 0 0 0.001
Error in checkTrue(is_near(norm(c()), 0)) : Test not TRUE

 done successfully.

Executing test function test.norm.three  ... [1] "Starting tests.."
Timing stopped at: 0 0 0
Error in t.default(x) : argument is not a matrix
 done successfully.

Executing test function test.norm.two  ... [1] "Starting tests.."
Timing stopped at: 0.08 0 0.104
Error : evaluation nested too deeply: infinite recursion / options(expressions=)?
 done successfully.

RUNIT TEST PROTOCOL -- Wed Oct 23 23:33:37 2013
***********************************************
Number of test functions: 3
Number of errors: 2
Number of failures: 1


1 Test Suite :
norm - 3 test functions, 2 errors, 1 failure
FAILURE in test.norm.one: Error in checkTrue(is_near(norm(c()), 0)) : Test not TRUE

ERROR in test.norm.three: Error in t.default(x) : argument is not a matrix
ERROR in test.norm.two: Error : evaluation nested too deeply: infinite recursion / options(expressions

Details
***************************
Test Suite: norm
Test function regexp: ^test.+
Test file regexp: norms.R
Involved directory:
.
---------------------------
Test file: ./norms.R
test.norm.one: FAILURE !! (check number 1)
Error in checkTrue(is_near(norm(c()), 0)) : Test not TRUE

test.norm.three: ERROR !!
Error in t.default(x) : argument is not a matrix
test.norm.two: ERROR !!
Error : evaluation nested too deeply: infinite recursion / options(expressions=)?
}
```

Spend some time parsing the output. The unit tests fail on several places and the output tells which these are.

OK, now it is time for debugging!

3 Debugging

Start by typing `test.norm.one()` to get the error. This will be an ”argument of length zero” type-of error. Usually R’s messages are cryptic but with practice one can start making sense out of them. For example, here we suspect that a vector might be empty when it should not. To use the debugger, one could use the “browser” in R (i prefer it), which is an interactive tool for digging into the state of variables at the time of execution. Now type,

```
> options(error=recover)
```

This enables the browser. Now run again the command and observe the output.

```
> test.norm.one()
[1] "Starting tests.."
Error in checkTrue(is_near(norm(c()), 0)) : Test not TRUE

Enter a frame number, or 0 to exit

1: test.norm.one()
2: norms.R#46: norm.checks(normOne)
3: norms.R#35: checkTrue(is_near(norm(c()), 0))

Selection:
```

Let's parse the output. The first lines is the error description. Then there are 3 numbered choices. These represent the "stack trace" (i.e. the execution path of R functions before the error). For example, we now know that we first called `test.norm.one()`, then called `normChecks(normOne)` and then the code failed in function `CHECK_near(norm(c()), 0)` at line 35 of the file (code lines are also very helpful).

This is all very important but still not very helpful with solving it. Now type "3" and press ENTER. You should see following:

```
Selection: 3
Called from: eval(expr, envir, enclos)
Browse[1]>
```

Now the browser is invoked and you are ready to start digging in. You can treat the browser enviroment exactly as the typical R command line, with the only difference being that the browser has "frozen" at the particular state at the time of error. Now type the following:

```
Browse[1]> objects()
[1] "expr"   "msg"    "result"
```

The browser informs us that there are three variables, `expr, msg, result`. You can inspect their values by typing their name and pressing ENTER. This is generally very useful in large applications. Our case is much simpler. The

```
Browse[1]> expr
logical(0)
```

The `expr` variable is the one checked by the `checkTrue` function of `RUnit`. This should be `TRUE` or `FALSE`, but the browser informs us that it has the empty `logical(0)` value. This means that the `normOne` function returned an empty thing which it shouldn't. We **conclude** that:

- The error is when calling `normOne(c())`, i.e. with an empty vector
- The function returns an empty "thing" when it should return 0

We have now enough information to fix the error. Note that this problem is in all functions. So, make sure that *every norm* functions starts with:

```
        n = length(x)
        if(n==0) return(0)
```

If you are in the `Browser[1]` prompt just press ENTER to move to the `Selection:` prompt. To leave the `Browser` all together, press ESCAPE (from `Selection:` prompt). The process of debugging will involve some iterations. We fix a few things, then run tests again, fix again etc.

Now run the tests again and observe the following line:

```
> test.norm.one()
[1] "Starting tests.."
Error in checkTrue(is_near(norm(c(3, 4)), 5)) : Test not TRUE
In addition: Warning message:
In sqrt(sum) : NaNs produced

Enter a frame number, or 0 to exit

1: test.norm.one()
2: norms.R#45: norm.checks(normOne)
3: norms.R#35: checkTrue(is_near(norm(c(3, 4)), 5))
```

This means that `test.norm.one` test is failing because it is not calculating the correct norm for the vector (3,4). This shows the real power of unit testing. All we have to do now, is trace back what calculations `normOne` is doing for this particular input. In our case, it is not really hard to figure out the error. In the iterations we are subtracting instead of adding squares. So the correct code for the first norm function is:

```
normOne <- function(x) {
  n = length(x)
  if(n==0) return(0)
  sum = 0
  for(i in 1:n)
    sum = sum + x[i]^2
  return(sqrt(sum))
}
```

After fixing this and running `run.tests()` you will see the line:

```
test.norm.one: (3 checks) ... OK (0 seconds)
```

This means that everything is fine for `test.norm.one`. However there is still one error. Find it using the methods above. After fixing the errors, we have the correct code. At this point your code should look like the code in `A.2`. After running `run.tests()` you should see no errors:

```
Test file: ./norms.R
test.norm.one: (3 checks) ... OK (0 seconds)
test.norm.three: (3 checks) ... OK (0 seconds)
test.norm.two: (3 checks) ... OK (0 seconds)
```

4 Logging

Checking allows to catch pathological cases early. Some other cases might not be that crucial, yet we would like to know about them. Furthermore, the error might appear towards the end of a lengthy computation and so we would like to trace back what went wrong. *Logging* allows to log things (like values of a list, or messages) on a screen or a file. It is easy to write your own logging functions, however the `logging` package will probably be enough [3].

A *logger* is the object that handles logging actions. Each logger has one or more *handlers*. A handler is a function with a pre-specified *level* that receives the message and logs anything that is *at least as important* as its level. The most common levels used are `DEBUG`, `INFO`, `WARNING` and `ERROR`. There are many ways to change the format of your logs, add/remove handlers to a logger and so on (see documentation).

[3]Check out this tutorial here`http://logging.r-forge.r-project.org/sample_session.php`

In our example, say we want to log a `WARNING` when the input array has the same elements (for some reason this would be a bad sign for our application). Also, assume that we want to print to console and to a file as well. First install the `logging` package by `install.packages("logging")`. Then, re-write, the start of the `norms.R` as follows:

```
library(RUnit)
library(logging)
basicConfig()
addHandler(writeToFile, file="sample.log", level="WARNING")

normOne <- function(x) {
  n = length(x)
  if(n==0) return(0)
  sum = 0
  if(length(unique(x)) == 1)
    logwarn("List of length %d has same elements", length(x))

  for(i in 1:n)
    sum = sum + x[i]^2
  return(sqrt(sum))
}
```

Now try to input an array with 10 ones, i.e.:

```
> x = rep(1, 10)
> normOne(x)
2013-10-17 00:38:15 WARNING::List of length 10 has same elements
[1] 3.162278
```

You see that we got a warning for that. Also check out your current DIR that a file `sample.log` has been created with the same message. Note that we got two messages because we have two handlers, the `basic.stdout` that comes by default and the `writeToFile` that we added in the beginning. To see all of your handlers type:

```
> names(getLogger()[['handlers']])
[1] "basic.stdout" "writeToFile"
```

If you would to remove, say the one printing to console, then type:

```
> removeHandler("basic.stdout")
> normOne(x)
[1] 3.162278
```

The warning message will only be printed in the file now.

Last thing about levels. Recall that each handler has a prespecified level. Anything less important will not be printed. To see this, try adding the line:

```
loginfo("Calling normOne")
```

as a first command in `normOne`. You will see the message in console but it will not be saved in file. This is because the handler that is saving on file was created at the `WARNING` level and so the `INFO` level (which is less important) will not be logged.

In general, logging is useful but should not be overused. One main reason is that it could eat up valuable resources, especially when saving to files. Often, logging is used in the early stages of development and then it is silenced when the code is more mature.

5 Profiling

Now that we etablished correctness, we need to move on to efficiency. *Profiling* is the process in which we gather information about our code, including calls, running times, memory usage etc. R comes with a standard profiler that uses the duo `Rprof` and `summaryRprof`. Say we want to check how fast our methods are. Add this function to the `norms.R` script:

```
profile.norms <- function() {
  Rprof(filename="profile.txt",memory.profiling=T)
  for (i in 1:1000) {
    x <- rnorm(100)
    normOne(x)
    normTwo(x)
    normThree(x)
  }
  Rprof(NULL)
}
```

Your R script should like the one in `A.3`. Now run `profile.norms` and invoke the summary function like this:

```
> profile.norms()
> summaryRprof(filename="profile.txt")
$by.self
                 self.time self.pct total.time total.pct
"head"                1.06    26.37       2.96     73.63
"stopifnot"           0.76    18.91       1.46     36.32
"normTwo"             0.64    15.92       3.74     93.03
"head.default"        0.44    10.95       1.90     47.26
"sys.parent"          0.28     6.97       0.28      6.97
"match.call"          0.20     4.98       0.62     15.42
"sys.call"            0.14     3.48       0.42     10.45
"normOne"             0.14     3.48       0.24      5.97
"-"                   0.08     1.99       0.08      1.99
"=="                  0.08     1.99       0.08      1.99
"sqrt"                0.06     1.49       0.06      1.49
"^"                   0.04     1.00       0.04      1.00
"+"                   0.04     1.00       0.04      1.00
"profile.norms"       0.02     0.50       4.02    100.00
"rnorm"               0.02     0.50       0.02      0.50
"unique.default"      0.02     0.50       0.02      0.50

$by.total
                 total.time total.pct self.time self.pct
"profile.norms"        4.02    100.00      0.02     0.50
"normTwo"              3.74     93.03      0.64    15.92
"head"                 2.96     73.63      1.06    26.37
"head.default"         1.90     47.26      0.44    10.95
"stopifnot"            1.46     36.32      0.76    18.91
"match.call"           0.62     15.42      0.20     4.98
"sys.call"             0.42     10.45      0.14     3.48
"sys.parent"           0.28      6.97      0.28     6.97
"normOne"              0.24      5.97      0.14     3.48
```

```
"-"                     0.08      1.99      0.08     1.99
"=="                    0.08      1.99      0.08     1.99
"sqrt"                  0.06      1.49      0.06     1.49
"^"                     0.04      1.00      0.04     1.00
"+"                     0.04      1.00      0.04     1.00
"rnorm"                 0.02      0.50      0.02     0.50
"unique.default"        0.02      0.50      0.02     0.50
"unique"                0.02      0.50      0.00     0.00

$sample.interval
[1] 0.02

$sampling.time
[1] 4.02
```

There is a lot of info here. Focus on the `by.total` frame: "total.time" gives the total running time of a function and "self.time" is time spent entirely within the function. Clearly `normTwo` is a very inefficient implementation (not surprising since it is recursive). Also, `normOne` is not fast (takes 0.24 secs for 1,000 runs which is not something to be proud of). Notice that `normThree` is not even in the menu. It is just too fast. Functions that use linear algebra are usually the faster in `R`. We can also see what exactly is slow here. For example, we see that `head` is a very costly operation (used by `normOne`), since almost 3 secs out of the 4 in total are spent `head`'ing.

There are packages that add some visualization (e.g. `profr`) but `Rprof` should probably be enough.

6 Style

Beauty is in the eye of the beholder. It might look superficial (in the beginning) but as you build up as a developer, you will come to appreciate its importance. Good code needs good style. By style we refer to the following aspects:

- *Naming*. Variable and function naming. General structure and layout.
- *Comments*. Documentation and comments. These can be general (e.g. describing a function) or specific (commenting a line to explain what it does)
- *Spacing*. Spaces are important for readability. Inconsistencies can make the code unreadable. Same goes for identation.

There is no golden rule in style and each programmer follows his/her own style one way or another. However, some rules are golden. The first one is *consistency*. Whatever style you have, stick to it. The second is *succinctness*. Good and descriptive names for functions/variables and only necessary and sufficient code, trumps anything else. Good style immediately reveals to an external developer three important things:

1. what the code does
2. how it is doing it
3. how it could be changed/improved

Google has a nice guide for R style. You may find it here `http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml`. You should not consider as it as the de facto style guide but as a good template to define your own style.

7 Appendix

7.1 A.1 (after adding tests)

```
# norms.R
# Functions to compute the L-2 norm of a column vector
# ||x|| = sqrt(x_1^2 + x_2^2 + ...)
# First implementation: Iterates over all elements

library(RUnit)

normOne <- function(x) {
  n = length(x)
  sum = 0
  for(i in 1:n)
    sum = sum - x[i]^2
  return(sqrt(sum))
}
# Second implementation: recursive definition
normTwo <- function(x) {
  n = length(x)
  if(n==1)
    return(abs(x[1]))
  sqrt(normTwo(head(x, n-1))^2 + x[n]^2)
}
# Third implementation: Uses vector multiplication xx
normThree <- function(x) {
  sqrt(t(x) %*% xx)
}

# Function that tests a "norm" function
norm.checks <- function(norm) {
  print("Starting tests..")
  checkTrue(is_near(norm(c()), 0))
  checkTrue(is_near(norm(c(3,4)), 5))
  x = rnorm(20)
  checkTrue(is_near(norm(2 * x), 2 * norm(x), tol=1e-6))
}

is_near <- function(x, x0, tol=1e-9) {
  return(abs(x-x0) < tol)
}

test.norm.one <- function() {
  norm.checks(normOne)
}
test.norm.two <- function() {
  norm.checks(normTwo)
}
test.norm.three <- function() {
  norm.checks(normThree)
}
```

```
run.tests <- function() {
  test.suite <- defineTestSuite(name="norm", dirs=".", testFileRegexp="norms.R")
  test.result <- runTestSuite(test.suite)
  printTextProtocol(test.result)
}
```

7.2 A.2

```
# norms.R
# Functions to compute the L-2 norm of a column vector
# ||x|| = sqrt(x_1^2 + x_2^2 + ...)
# First implementation: Iterates over all elements

library(RUnit)

normOne <- function(x) {
  n = length(x)
  if(n==0) return(0)
  sum = 0
  for(i in 1:n)
    sum = sum + x[i]^2
  return(sqrt(sum))
}
# Second implementation: recursive definition
normTwo <- function(x) {
  n = length(x)
  if(n==0) return(0)
  if(n==1)
    return(abs(x[1]))
  sqrt(normTwo(head(x, n-1))^2 + x[n]^2)
}
# Third implementation: Uses vector multiplication xx
normThree <- function(x) {
  n = length(x)
  if(n==0) return(0)
  as.numeric(sqrt(t(x) %*% x))
}

# Function that tests a "norm" function
norm.checks <- function(norm) {
  print("Starting tests..")
  checkTrue(is_near(norm(c()), 0))
  checkTrue(is_near(norm(c(3,4)), 5))
  x = rnorm(20)
  checkTrue(is_near(norm(2 * x), 2 * norm(x), tol=1e-6))
}

is_near <- function(x, x0, tol=1e-9) {
  return(abs(x-x0) < tol)
}
```

```
test.norm.one <- function() {
  norm.checks(normOne)
}
test.norm.two <- function() {
  norm.checks(normTwo)
}
test.norm.three <- function() {
  norm.checks(normThree)
}

run.tests <- function() {
  test.suite <- defineTestSuite(name="norm", dirs=".", testFileRegexp="norms.R")
  test.result <- runTestSuite(test.suite)
  printTextProtocol(test.result)
}
```

7.3 A.3

```
# norms.R
# Functions to compute the L-2 norm of a column vector
# ||x|| = sqrt(x_1^2 + x_2^2 + ...)
# First implementation: Iterates over all elements
library(RUnit)
library(logging)
basicConfig()
addHandler(writeToFile, file="sample.log", level="WARNING")

normOne <- function(x) {
  n = length(x)
  if(n==0) return(0)
  sum = 0
  if(length(unique(x)) == 1)
      logwarn("List of length %d has same elements", length(x))
  for(i in 1:n)
      sum = sum + x[i]^2
  return(sqrt(sum))
}


# Second implementation: recursive definition
normTwo <- function(x) {
  n = length(x)
  if(n==0) return(0)
  if(n==1)
    return(abs(x[1]))
  sqrt(normTwo(head(x, n-1))^2 + x[n]^2)
}
# Third implementation: Uses vector multiplication xx
normThree <- function(x) {
  n = length(x)
  if(n==0) return(0)
  as.numeric(sqrt(t(x) %*% x))
}

# Function that tests a "norm" function
norm.checks <- function(norm) {
  print("Starting tests..")
  checkTrue(is_near(norm(c()), 0))
  checkTrue(is_near(norm(c(3,4)), 5))
  x = rnorm(20)
  checkTrue(is_near(norm(2 * x), 2 * norm(x), tol=1e-6))
}

is_near <- function(x, x0, tol=1e-9) {
  return(abs(x-x0) < tol)
}

test.norm.one <- function() {
  norm.checks(normOne)
}
test.norm.two <- function() {
  norm.checks(normTwo)
}
test.norm.three <- function() {
  norm.checks(normThree)
}

run.tests <- function() {
  test.suite <- defineTestSuite(name="norm", dirs=".", testFileRegexp="norms.R")
  test.result <- runTestSuite(test.suite)
  printTextProtocol(test.result)
}

profile.norms <- function() {
  Rprof(filename="profile.txt",memory.profiling=T)
  for (i in 1:1000) {
    x <- rnorm(100)
    normOne(x)
    normTwo(x)
    normThree(x)
  }
  Rprof(NULL)
}
```