Practices make perfect: improved programming through software development best practices

Ethan Nelson

University of Wisconsin–Madison

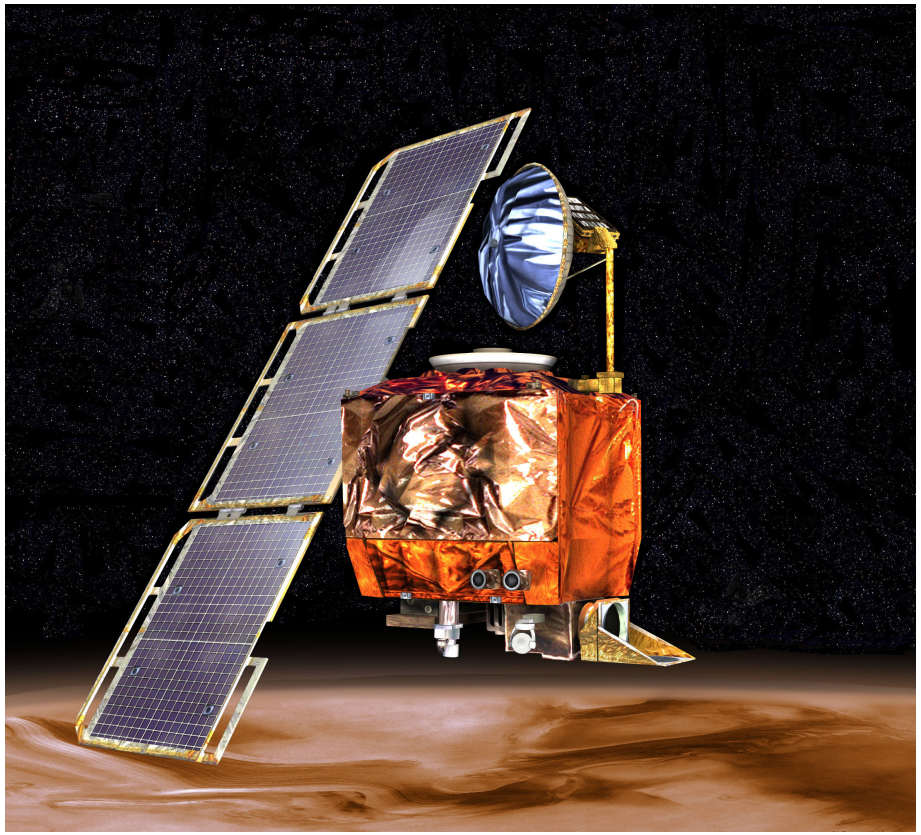
February 3, 2016

I will be discussing three best practices from software development that I have found extremely useful and important for scientific programming:

1. Modular code
2. Automated test development
3. Continuous improvement

- Scientists today are many things...
  - Writers
  - Readers
  - Communicators
  - Critical thinkers
  - Programmers
  - And others
- We develop, practice, and revise many of the above, so programming should receive some of the same attention.

Mars Climate Orbiter



Mars Climate Orbiter nearing Sept. 23 arrival

By Mary Hardin
JPL Universe
September 17, 1999

Mars Climate Orbiter, the first of two JPL spacecraft to reach Mars this year, is set to go into orbit around the red planet to become our first interplanetary weather satellite and a communications relay for the Mars Polar Lander, which will arrive at Mars this December.

The Orbiter will fire its main engine at about 1:50 a.m. Pacific Daylight Time (spacecraft time) on Thursday, Sept. 23, to slow itself down so that it can be captured in orbit around the planet.

"The curtain goes up on this year's Mars missions with the orbit insertion of Mars Climate Orbiter," said Dr. Sam Thurman, flight operations manager for the Orbiter at JPL. "Hopefully, the happily-ever-after part of the play will be the successful mission of the Mars Polar Lander that begins in December, followed by the mapping mission of the Orbiter that is set to begin next March."

Once captured in orbit around Mars, the Orbiter will begin a period of aerobraking. During each of its long, elliptical loops around Mars, the Orbiter will pass through the upper layers of the atmosphere each time it makes its closest approach to the planet. Friction from the atmosphere on the spacecraft and its wing-like solar array will cause the spacecraft to lose some of

- $328 million mission

Mars Climate Orbiter

Mishap Investigation Board

Phase I Report

November 10, 1999

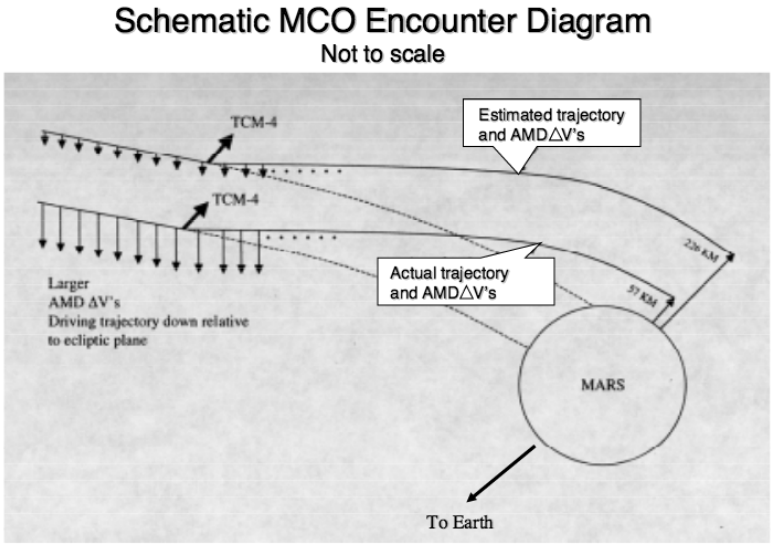


Figure 4

Root Cause:

- Failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models.

(lbf·s and not N·s)

Species distributions shift downward across western North America

Species distributions shift downward across western North America

"The above article, published online on 18 August 2014 in Wiley Online Library (wileyonlinelibrary.com), has been retracted by agreement between the authors, Dr Melanie Harsch and Associate Professor Janneke Hille Ris Lambers, journal Editor-in-Chief, Professor Stephen Long, and John Wiley & Sons Ltd. The retraction has been agreed for the following reasons: a coding error affected the results and therefore invalidated the broad-scale conclusions presented in the article. The article presented broad-scale patterns of species distribution shifts in response to recent climate change. Unfortunately, it has since been found that one approach used to account for sampling bias, the null model approach, was affected by the coding error."

`http://dx.doi.org/10.1111/gcb.12840`

Evidence of unusual late 20th century warming from an Australasian temperature reconstruction spanning the last millennium, J. Gergis et al. 2014.

Evidence of unusual late 20th century warming from an Australasian temperature reconstruction spanning the last millennium, J. Gergis et al. 2014.

"An issue has been identified in the processing of the data used in the study, which may affect the results. While the paper states that 'both proxy climate and instrumental data were linearly detrended over the 1921-1990 period', we discovered on Tuesday 5 June that the records used in the final analysis were not detrended for proxy selection, making this statement incorrect. Although this is an unfortunate data processing issue, it is likely to have implications for the results reported in the study. The journal has been contacted and the publication of the study has been put on hold."

Coauthor emailed statement from RetractionWatch.com

The Economic Effects of Climate Change, R. S. J. Tol, 2009

The Economic Effects of Climate Change, R. S. J. Tol, 2009

"Gremlins intervened in the preparation of my paper 'The Economic Effects of Climate Change' published in the Spring 2009 issue of this journal. In Table 1 of that paper, titled 'Estimates of the Welfare Impact of Climate Change,' minus signs were dropped from the two impact estimates, one by Plambeck and Hope (1996) and one by Hope (2006). In Figure 1 of that paper, titled 'Fourteen Estimates of the Global Economic Impact of Climate Change,' and in the various analyses that support that figure, the minus sign was dropped from only one of the two estimates. The corresponding Table 1 and Figure 1 presented here correct these errors. Figure 2 titled,'Twenty-One Estimates of the Global Economic Impact of Climate Change' adds two overlooked estimates from before the time of the original 2009 paper and five more recent ones."

`dx.doi.org/10.1257/jep.28.2.221`

- Errors are inherent in computer programming just like anything else we do as humans.
- But we can embrace practices that enable us to catch or minimize these errors in our professions.
- Don't test your programs with the science; test them with your computer!

- If your analysis is one long script and a problem arises, where do you start to debug?

Modular code

- Modular code is code that is broken up into small chunks, each with a set purpose.
- This is usually implemented as functions, classes, subroutines, etc. depending on the language.
- Modular code is also reusable code!
  - When you write a program to do something, you are likely going to do that thing more than once in your career.
  - Rather than continually inventing the wheel, modular code allows you to grow a garden of code.
- "The whole is greater than the sum of its parts." ∼Aristotle

```
the_file = h5py.File('/file/path','r')
data = the_file['CS']['zFactor'][:]*0.01
data1 = numpy.reshape(data,[3, 4])
data2 = numpy.rot90(data1, 1)
```

```
the_file = h5py.File('/file/path','r')
data = the_file['CS']['zFactor'][:]*0.01
data1 = numpy.reshape(data,[3, 4])
data2 = numpy.rot90(data1, 1)
```

```
def read_hdf(location, swath, product):
    the_file = h5py.File(location)
    data = the_file[swath][product][:]
    return data
def rearrange_data(input_data, nx, ny):
    data = input_data * 0.01
    data = numpy.reshape(input_data,[nx, ny])
    data = numpy.rot90(data, 1)
    return data
>> data = read_hdf('/file/path','CS','zFactor')
>> data = rearrange_data(data, 3, 4)
```

```
def read_hdf(location, swath, product):
    the_file = h5py.File(location)
    data = the_file[swath][product][:]
    return data
def rearrange_data(input_data, nx, ny):
    data = input_data * 0.01
    data = numpy.reshape(input_data,[nx, ny])
    data = numpy.rot90(data, 1)
    return data
>> z_data = read_hdf('/file/path','CS','zFactor')
>> z_data = rearrange_data(z_data, 3, 4)

>> v_data = read_hdf('/file/path2','CS','velocity')
>> v_data = rearrange_data(v_data, 3, 1)
```

- I find it helpful to separate my code routines into this general workflow:
  1. Data reading
  2. Data preprocessing
  3. Data analysis
  4. Data writing
  5. Data plotting

- Modular code requires some abstraction of what your code should accomplish.
- The process adds an initial overhead, but as you harvest your "code garden" by reusing functions, your time dividend pays off.
- Prebuilt functions can also be easily shared among research group members, colleagues, other scientists, the public...

- Modular code requires some abstraction of what your code should accomplish.
- The process adds an initial overhead, but as you harvest your "code garden" by reusing functions, your time dividend pays off.
- Prebuilt functions can also be easily shared among research group members, colleagues, other scientists, the public...

Executive Order 13642 (2013)

Making Open and Machine Readable the New Default for Government Information

"...the default state of new and modernized Government information resources shall be open and machine readable."

- How much will this expand to include government-sponsored research?

- If your analysis is one long script and a problem arises, where do you start to debug?
- Or, better yet, how will you know if a problem even exists?

- A bug is a test you wrote without knowing it!
- The concept of automated tests is focused on providing examples with known outputs for the computer to test a function or piece of code against.
- If something unexpected happens, the test will fail.
- Building on the usefulness of modular code, tests are most useful for predicting and assuring computer behavior when integrated into small chunks of code.
- "The universe rings true whenever you fairly test it." ∼C. S. Lewis

```
def rearrange_data(input_data, nx, ny):
    data = input_data * 0.01
    data = numpy.reshape(input_data, [nx, ny])
    data = numpy.rot90(data, 1)
    return data
```

```
def rearrange_data(input_data, nx, ny):
    '''
    >>> rearrange_data(numpy.array([3,4,8,9]),2,2)
    numpy.array([[.04,.09], [.03,.08]])
    '''
    data = input_data * 0.01
    data = numpy.reshape(input_data, [nx, ny])
    data = numpy.rot90(data, 1)
    return data

>> doctest.testmod()
TestResults(failed=0, attempted=1)
```

- Native testing availability depends on the language of choice:
  - Python: unittest, nosetest, doctest, pytest
  - C: CUnit, unity, Check, cUnit, CuTest
  - MATLAB: runtests
  - Java: JUnit
- If a tool doesn't exist for a language (e.g. IDL or Fortran), we can easily create our own functionality.

```
def rearrange_data(input_data, nx, ny):
    data = input_data * 0.01
    data = numpy.reshape(input_data, [nx, ny])
    data = numpy.rot90(data, 1)
    return data
def test_rearrange_data():
    expectation = numpy.array([[.04,.09], [.03,.08]]
    output = rearrange_data(numpy.array([3,4,8,9]),2
    if output != expectation:
        raise Exception("""test_rearrange_data()
        failed! Expected %s, got %s"""
        % (numpy.array_str(expectation),
        numpy.array_str(output))

>> test_rearrange_data()
>>
```

- True test-driven development starts the programming cycle with writing a test before you write your function:
  1. Identify what the function should do
  2. Write a test that calls the function with a given input and expects a given output
  3. Fill in the program and edit it until the test works

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    """
```

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
```

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
```

```
>> doctest.testmod()
Failed example:
  temperature(1.67E7, 0.3)
Expected:
  5597.523133627065
Got nothing
```

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
    if emissivity:
        temp = emis[0] * 5.67E-8 * flux ** 4.0
    else:
        temp = 5.67E-8 * flux ** 4.0
    return temp
```

```
def temperature(flux, *emis):
    """
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
    if emissivity:
        temp = emis[0] * 5.67E-8 * flux ** 4.0
    else:
        temp = 5.67E-8 * flux ** 4.0
    return temp

>> doctest.testmod()
Failed example:
...
Got:
    1.323E+21
```

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
    if emissivity:
        temp = (flux / emis[0] / 5.67E-8) ** .25
    else:
        temp = (flux / 5.67E-8) ** .25
    return temp
```

```
def temperature(flux, *emis):
    """
    Calculates the temperature
    from Stefan-Boltzmann law.
    >>> temperature(1.67E7, 0.3)
    5597.523133627065
    """
    if emissivity:
        temp = (flux / emis[0] / 5.67E-8) ** .25
    else:
        temp = (flux / 5.67E-8) ** .25
    return temp

>> doctest.testmod()
TestResults(failed=0, attempted=1)
```

- Writing tests provides an automated method to test your program.
- For scientific purposes, it may not be necessary or fitting *everywhere*.
- They should at least be used on your core analysis routines or calculators.
- As with implementing modular code, there is an upfront investment of time.

- If your analysis is one long script and a problem arises, where do you start to debug?
- Or, better yet, how will you know if a problem even exists?
- How can I improve my existing code?
- How can I take steps to improve future code I write?

- "Kaizen", or continuous improvement, is a common practice in the manufacturing world to standardize methods and eliminate waste.
- The same basic philosophy can be applied to programming as well.

改善

- Refactoring, simply put, means changing the guts of a program or function without changing the output.
- The process takes many forms, all with different goals.

- Make code more modular
  - Discussed at the outset.
- Make code more abstract/general
  - The fewer hard-coded values in your program–and therefore the more values given in a function call–the better.
- Make code more readable (replacing variables named temp1, temp2, temp3..., or adding comments)
  - If you died tomorrow, could we easily adopt your code to continue your legacy?
- Make code faster
  - Optimization is important, especially for operational purposes. But do not do it prematurely.
- "Whenever I have to think to understand what the code is doing, I ask myself if I can refactor the code to make that understanding more immediately apparent." ∼Martin Fowler

- Refactoring adds a fourth step to the test-driven development cycle.
  1. Identify what the function should do
  2. Write a test that calls the function with a given input and expects a given output
  3. Fill in the program and edit it until the test works
  4. Clean up and optimize the program

- Some automated refactoring tools exist, though they generally favor more software-y languages:
  - Java, Javascript, C++, Python to an extent...
- Ctrl+F is a semi-automatic tool, too.
- The manual process of refactoring helps you fully understand how your code works.

- Your research group members or peers probably review papers or articles you have written.
- Should the same apply for code?

- Code review is literally reading code, whether it is code you have written or code someone else has written.
- If you have ever helped someone try to debug their code, you have likely already done some code review.
- The code does not even have to be related to science...plenty of communities and hubs exist online that host code.
  - Github, Bitbucket, GitLab, Google Code...
- "The more that you read, the more things you will know. The more that you learn, the more places you'll go." ∼Dr. Seuss

- If you don't like reading on the computer, print the code out and read it on a hard copy.
- You can pick up on methods or ways of solving a problem you may not have thought of originally.

- Version control means tracking the development of your code over time.
- With tools like git and mercurial, you can save snapshots of your code at certain development or scientific milestones. Or whenever you feel like it.
  - Submitted a paper? "Commit" your code at that time so you can refer back to it when revisions come around.
  - Refactoring the code, but not sure if it will work? "Commit" your code so you can see what all that you changed.

Version control

on Feb 22, 2015

including orbit coords
ethan-nelson committed on Feb 22, 2015 — ed7d3a0

adding longitude orbit coordinates
ethan-nelson committed on Feb 22, 2015 — 6307eeb

adding latitude slice coordinates
ethan-nelson committed on Feb 22, 2015 — 1cb36de

on Feb 21, 2015

starting coordinate file for orbits
ethan-nelson committed on Feb 21, 2015 — f6cce8e

testing exponent frm
ethan-nelson committed on Feb 21, 2015 — 1c230a3

fixing stemp transparency
ethan-nelson committed on Feb 21, 2015 — aa9311d

adding new layers
ethan-nelson committed on Feb 21, 2015 — 180c94d

updating ignore file
ethan-nelson committed on Feb 21, 2015 — 680b933

Update index.html
ethan-nelson committed on Feb 21, 2015 — 960307d

adjusting coords for new cross sections
ethan-nelson committed on Feb 21, 2015 — e3ed49c

on Feb 20, 2015

increasing map coverage/attribution

Version control

25 index.html — View

```
@@ -47,8 +47,8 @@
                [[26.7010,-86.4091],[46.4409,-86.4091]],
                [[26.7010,-66.8973],[46.4409,-66.8973]] ];

-  var latline = new L.polyline(latarray[0]);
-  var lonline = new L.polyline(lonarray[0]);

   function updatelatcs(){
        var quantity = document.getElementById('latrange').value;
@@ -102,8 +102,8 @@
                        "<img id='loncs1' height='200' width='400' src='images/transparent.png'><br>";
        document.getElementById('images').replaceChild(imagediv,document.getElementById('images').childNodes[0]);

-       latline.addTo(map);
-       lonline.addTo(map);

        refreshimages();

@@ -120,6 +120,13 @@
                return;
        }

+       var frequencydiv = document.createElement('div');
+       frequencydiv.innerHTML = "<input type='button' id='sbutton' onclick='togglefrequency('s')' value='Show S band'>" +
+                        "<input type='button' id='kubutton' onclick='togglefrequency('ku')' value='Show Ku band'>" +
+                        "<input type='button' id='kabutton' onclick='togglefrequency('ka')' value='Show Ka band'>" +
+                        "<input type='button' id='wbutton' onclick='togglefrequency('w')' value='Show W band'>";
+ document.getElementById('frequencies').replaceChild(frequencydiv,document.getElementById('frequencies').childNodes[0]);
+
        var sliderdiv = document.createElement('div');
        sliderdiv.innerHTML = "<label for='vertrange'>Height:</label>" +
                        "<input id='vertrange' type='range' min='0' max='2'" +
@@ -142,6 +149,10 @@
        map.removeLayer(latline);
        map.removeLayer(lonline);
        }
+
```

- **Modular code**, code split into smaller pieces, fosters reusable and legible code.
- **Automated tests** provide a method to objectively assess whether your code is functioning properly.
- **Continually improving** code and programming skills helps both you and your code to become better.
- Utilizing these practices will help move you from assuming your code works to knowing.
- "I'm not a great programmer; I'm just a good programmer with great habits." ∼Martin Fowler

Questions?