NASA/TM—2017-219561

Verification Testing: Meet User Needs Figure of Merit

Bryan W. Kelly and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

October 2017

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Technical Report Server—Registered (NTRS Reg) and NASA Technical Report Server—Public (NTRS) thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers, but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., "quick-release" reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 757-864-6500

- Telephone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Program
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM—2017-219561

Verification Testing: Meet User Needs Figure of Merit

Bryan W. Kelly and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

October 2017

Acknowledgments

The primary author would like to thank his mentor, Bryan Welch; the SCENIC interns he worked with, Leo Steinkerchner, Andy Krieger, and Christian Gilbertson; the managers of SCENIC, Devon Griffin and Robyn Atkins, and the SCaN summer internship program managers Tim Gallagher and Lindsay Hill.

Level of Review: This material has been technically reviewed by technical management.

Available from

NASA STI Program
Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
703-605-6000

This report is available in electronic form at http://www.sti.nasa.gov/ and http://ntrs.nasa.gov/

Verification Testing: Meet User Needs Figure of Merit

Bryan W. Kelly* and Bryan W. Welch
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

Verification is the process through which Modeling and Simulation (M&S) software goes to ensure that it has been rigorously tested and debugged for its intended use. Validation confirms that said software accurately models and represents the real world system. Credibility gives an assessment of the development and testing effort that the software has gone through as well as how accurate and reliable test results are. Together, these three components form Verification, Validation, and Credibility (VV&C), the process by which all NASA modeling software is to be tested to ensure that it is ready for implementation.

NASA created this process following the CAIB (Columbia Accident Investigation Board) report seeking to understand the reasons the Columbia space shuttle failed during reentry. The report's conclusion was that the accident was fully avoidable, however, among other issues, the necessary data to make an informed decision was not there and the result was complete loss of the shuttle and crew. In an effort to mitigate this problem, NASA put out their Standard for Models and Simulations, currently in version NASA-STD-7009A, in which they detailed their recommendations, requirements and rationale for the different components of VV&C. They did this with the intention that it would allow for people receiving M&S software to clearly understand and have data from the past development effort. This in turn would allow the people who had not worked with the M&S software before to move forward with greater confidence and efficiency in their work.

This particular project looks to perform Verification on several MATLAB® (The MathWorks, Inc.) scripts that will be later implemented in a website interface. It seeks to take note and define the limits of operation, the units and significance, and the expected datatype and format of the inputs and outputs of each of the scripts. This is intended to prevent the code from attempting to make incorrect or impossible calculations. Additionally, this project will look at the coding generally and note inconsistencies, redundancies, and other aspects that may become problematic or slow down the code's run time. Certain scripts lacking in documentation also will be commented and cataloged.

1.0 Introduction

SCaN, Space Communication and Navigation, is the NASA program primarily focusing on enabling NASA missions to provide commands from ground controllers to their spacecraft, as well as to return telemetry and science data from NASA spacecraft to scientists via radio communications. SCENIC, the SCaN Center for Engineering, Networks, Integration, and Communications, is a project of SCaN that is working on a web-based user interface providing analysis of SCaN's communication assets interacting with NASA spacecraft. This project's goals were to look at several MATLAB® scripts and perform verification testing on them. That is to say, it was to look for potential bugs and to ensure that the scripts

*Summer Intern in Lewis' Educational and Research Collaborative Internship Project (LeRCIP), graduate student at Case Western Reserve University

were calculating their measurements correctly. The goal of these scripts was to determine if the communication architecture met hypothetical mission communication requirements on hypothetical sets of missions. However, numerous lines of code were unused, there was minimal documentation and it was initially difficult to see or understand how the code interacted.

These sorts of problems come about as a result of the code being a project that several people have worked on over multiple years with minimal documentation and explanation for how different areas of the code operate. There is a certain amount of expectation that anyone going in to use the code has a degree of understanding and knowledge not only about the general formulas and mathematics that is being used but also how the code implements these formulas (Ref. 1). Variable and script names generally do not give much indication as to what their function is. This leads to unnecessary difficulty when trying to work with the code.

In addition, this leads to unseen bugs as a result of combining multiple persons work causes unintended interactions. The code runs without errors, however the result that comes out is not correct. However, because the results are well within the feasible range of values, with no further testing or knowledge of the code one could reasonably assume that the code is working. In order to uncover these sorts of issues, verification testing is necessary.

There was also a documentation effort, much of which included creating workflows for the scripts. This included looking at all possible file paths that the code could take and noting limits of operation as needed. This was done to prevent the code from taking on impossible or code-breaking values. This is a very necessary part of verification as without it, it is possible for the code to try to make impossible calculations, such as dividing by zero, leading to nonworking code and user frustration (Ref. 2). This will be further discussed in Section 5.0. Section 2.0 will describe how performance was improved while Sections 3.0 and 4.0 will discuss the methodology used to find and correct bugs in the code and the process of standardizing conventions respectively.

2.0 Performance

One of the initial things this project looked at was for general errors and inconsistencies in the code. Numerous unused variables and subfunctions were removed in an effort to clean up the code and make it more readable, as well as reduce the size of the overall file size. Many of these unused variables were left over from previous methods of calculating variables and were set up in a way that they were on a path that would never be taken, but since they were not commented out it was not immediately apparent that this was the case.

For example, Figure 1 was a script in its entirety. As one can see, it was able to be replaced with a simple if statement in the higher script calling it. However, dozens of commented lines of code were making it difficult to immediately identify that, despite the fact that it seems most of the elseifs were checking for a variable that had long ago been permanently set to ICRF. The lower script had apparently not been changed as the programmer was likely unaware that there was a place downstream where this check was performed.

Figure 2 shows the resulting higher level script. Lines 93 to 95 were replaced with lines 64 and 65. These two lines of code are much easier for a programmer to read and understand than having to go to another function.

```
if strcmpi(sc.refframe, 'GCRF ')==1

elseif strcmpi(sc.refframe,'GRC ')==1 || strcmpi(sc.refframe, 'TDR ')==1

elseif strcmpi(sc.refframe, 'ICRF')==1
    [planet,~] = planetephem(Ttdb');
    sc.ECI = (sc.refpos' + planet{4}.ECI);%planet{3}.ICRF)';
elseif strcmpi(sc.refframe, 'ITRF2000 ')==1

elseif strcmpi(sc.refframe, 'ITRF-93 ')==1

elseif strcmpi(sc.refframe, 'ITRF-97 ')==1

elseif strcmpi(sc.refframe, 'MCI ')==1 || strcmpi(sc.refframe, 'MCI')==1

elseif strcmpi(sc.refframe, 'TEME ')==1

elseif strcmpi(sc.refframe, 'TOD ')==1
end
end
```

Figure 1.—An old, unnecessarily script

Old Version of interplanetaryaccess_mars

```
91 -        sc.refpos = (kepelems2pos(sc, JDtt, Ttdb))';
92      %Convert spacecraft position from the given coordinate frame to ECI
93 -        if strcmpi(sc.refframe, 'EME2000 ')~=1 || strcmpi(sc.refframe,'EME2000')~=1
94 -            sc = coord2ECI(sc, Ttdb);
95 -        end
96      %     sc.ECI = sc.refpos';
97       %Calculate the position of the spacecraft when its access is
98       %unobstructed by planets, the Sun, or the Moon
99 -        Ttdb = Ttdb';
```

New Version of interplanetaryaccess_mars

```
61 - sc.refpos = (kepelems2pos(sc, JDtt))';
62
63   %Convert spacecraft position from the given coordinate frame to ECI
64 - planet = planetephem(Ttdb);
65 - sc.ECI = (sc.refpos' + planet(4).ECI);
66
67   %Calculate the position of the spacecraft when its access is
68   %unobstructed by planets, the Sun, or the Moon
69 - planet = planetephem(Ttdb);
70 - sc = plntintr(sc,planet);
71
```

Figure 2.—A comparison of the higher level code before and after

3.0 Methodology

Following that, several checks were performed on individual functions to ensure that there were not any performance issues. Several problems were found here, namely with a list of the different missions. There were items misspelled, items missing, parse errors if certain items were nonzero. One of the more prominent errors, shown in Figure 3, was that string compare was only used to compare the first part of a string.

This was presumably done so that string compare could distinguish between the different missions, which were numbered in the list it was searching. However, because it was only looking at the beginning of the strings, for certain names whose first characters were the same, such as with HSF: Leo ops and HSF: Leo ops-servicing, the function could not distinguish between the two. This issue was resolved by adding a conditional that would halt the program at the problematic cases and remove the problem values, seen in Figure 4. Ideally changes would be made to the list of names, however that has already been standardized across multiple uses and is outside the scope of this project.

```
c=find(strncmp([out3{k}.requirements{:,1}],uselist{j},length(uselist{j})));
```

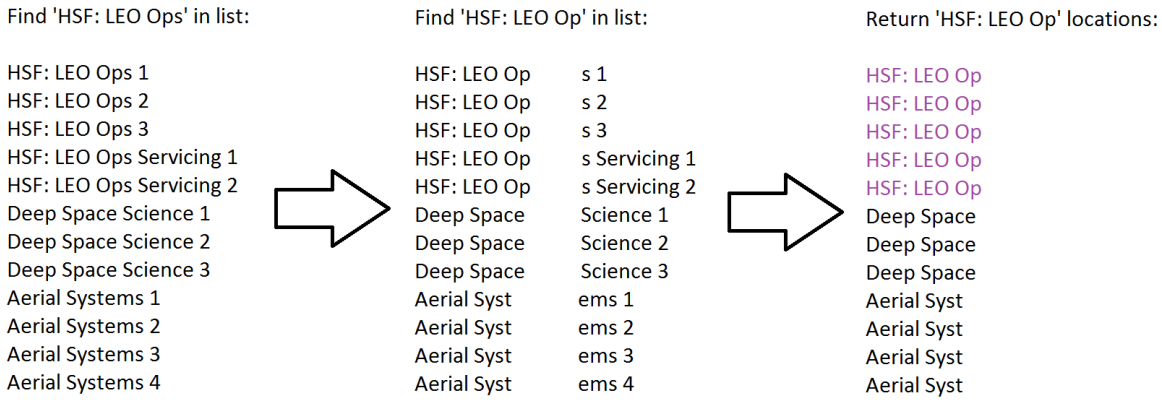

Figure 3.—Function erroneously lumping similar names together

```
64
65 -        if j == 1 || j == 13 % Added to prevent c from including incorrect values(it think
66 -            LastString = '                                                              ';
67 -            for s=1:length(c)
68 -                string = cell2mat(out3{k}.requirements{c(s),1});
69 -                if length(LastString)+1 < length(string) % Checks if current name is too l
70 -                    c=c(1:s-1);
71 -                    break
72 -                end
73 -                 LastString=string;
74 -            end
75 -        end
76
```

Figure 4.—If statement taking out incorrect names

However, other issues with such lists were also found. Using these sorts of strings is an easy to understand method of coding and differentiating between different use cases, but they are also very prone to breaking by something as simple as a slightly differently typed phrase(‘Deep Space Science’ and ‘Deep Space (Science)’ for example caused problems).

A full one quarter of the total cases were found to be defaulting to 1 every time because string names had changed between iterations and string compare was no longer functioning. This was not immediately obvious because the order of the list names and the order that the code ran through the cases was completely different. It was only after organizing the code so that the order was the same did it become clear that some cases were missing entirely. Other cases that existed were simply not used and were removed for clarification.

4.0 Standardization

Another issue managed was standardizing subfunctions. Several scripts contained their own, individual subfunctions, all of which were supposed functionally identical but still all had slight variations. This project looked to standardize these subfunctions into single functions that multiple scripts called. Having six almost identical subfunctions was confusing and unneeded.

However, it was found that in certain cases, the subfunctions contained functional differences, to the point where they gave completely different outputs to the same input, as seen in two separate versions of kep_solve in Figure 5. The two versions only have one functional line of code difference, which appeared to be leftover from a previous iteration of the function. Because of the exact inputs that the sample data had, this never became a significant issue.

In addition, a further issue with this particular function was that the line of code seemed to be a poor attempt to convert the value of M into radians from degrees. M was later input into sin, which expects its input in radians. It was getting it in degrees. While this was not causing any error outputs, the value output was wrong (sind is the correct function in this scenario). The error can be seen in Figure 6.

This demonstrates the danger of building on erroneous code, one error can lead to another. In this case the damage was limited, but if limits of operation and units are not noted and recorded then entire sections of code can rapidly become unusable. Cascading errors is something in particular this project looked for and sought to fix.

SNIP2 version of kep_solve

```
%% Solution of Kepler's Equation
function E = kep_solve(t,M,e)
% Eccentric anomaly (must be solved iteratively for Ek)
M = [132.238285927288];
t = [0.164613299865791];
e=0;
E_i=[];
E = zeros(1,size(t,2));
    for j=1:size(t,2)
        M=M-floor(M/2/pi)*2*pi;
        E_i = M(j);
        k = 1;
        err = 1e-10;
        while (k>err)
            y = e*sin(E_i)+M(j);
            k = abs(abs(E_i)-abs(y));
            E_i = y;
        end
        E(j) = E_i;
    end
end
```

```
ans =

    0.291394476516672
```

kepelems2pos version of kep_solve

```
%% Solution of Kepler's Equation
function E = kep_solve(t,M,e)
% Elliptic or Hyperbolic Anomaly (solved iteratively)
M = [132.238285927288];
t = [0.164613299865791];
e=0;
E_i=[];
E = zeros(1,length(t));
    for j=1:length(t)

        E_i = M(j);
        k = 1;
        err = 1e-10;
        while (k>err)
            y = e*sin(E_i)+M(j);
            k = abs(abs(E_i)-abs(y));
            E_i = y;
        end
        E(j) = E_i;
    end
end
```

```
ans =

   132.238285927288
```

Figure 5.—The two versions of kep_solve with identical inputs

SNIP2 version of kep_solve

```
%% Solution of Kepler's Equation
function E = kep_solve(t,M,e)
% Eccentric anomaly (must be solved iteratively for Ek)
M = [132.238285927288];
t = [0.164613299865791];
e=0;
E_i=[];
E = zeros(1,size(t,2));
    for j=1:size(t,2)
        M=M-floor(M/2/pi)*2*pi;
        E_i = M(j);
        k = 1;
        err = 1e-10;
        while (k>err)
            y = e*sin(E_i)+M(j);
            k = abs(abs(E_i)-abs(y));
            E_i = y;
        end
        E(j) = E_i;
    end
end
```

kepelems2pos version of kep_solve

```
%% Solution of Kepler's Equation
function E = kep_solve(t,M,e)
% Elliptic or Hyperbolic Anomaly (solved iteratively)
M = [132.238285927288];
t = [0.164613299865791];
e=0;
E_i=[];
E = zeros(1,length(t));
    for j=1:length(t)

        E_i = M(j);
        k = 1;
        err = 1e-10;
        while (k>err)
            y = e*sin(E_i)+M(j);
            k = abs(abs(E_i)-abs(y));
            E_i = y;
        end
        E(j) = E_i;
    end
end
```

```
y = e*sin(E_i)+M(j)

sin(E_i)
```

Figure 6.—E_i is incorrectly in degrees

5.0 Documentation

A final part of this project was documenting and commenting the scripts. Initially there were minimal comments, those that did exist were out of date. This project created workflows to show how the scripts interacted with one another, which scripts called which others, and which scripts were used multiple times. A section of one such workflow is shown below in Figure 7.

All scripts were given a header section with a description at the very least of the inputs and outputs as well as the general purpose of the script. This was meant to significantly lessen the time that a new designer would need to spend understanding the purpose behind the script. The header, at a minimum, contains a list of all the inputs and outputs and what they are meant to represent as well as the goal of the overall script.

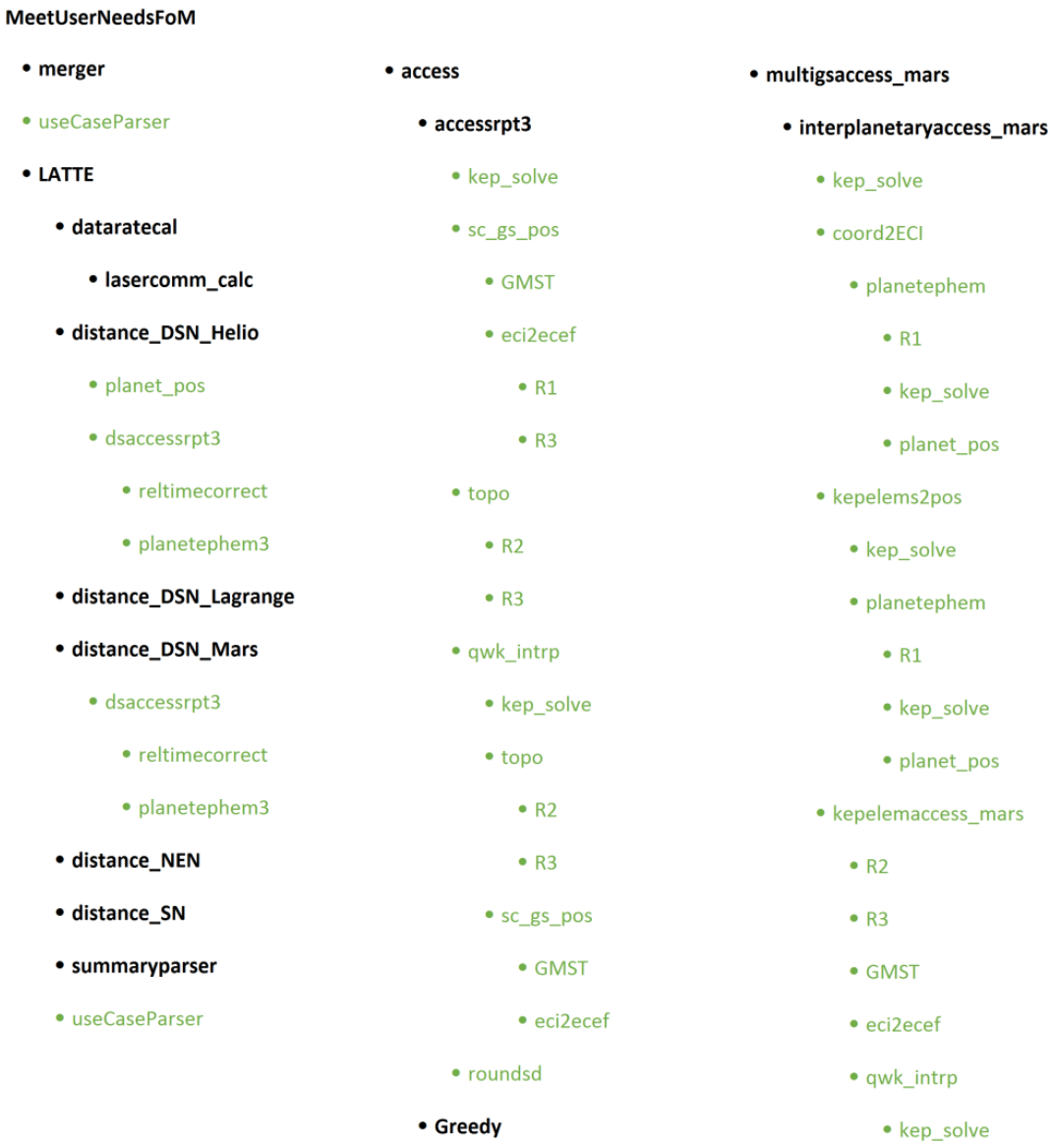

Figure 7.—A workflow chart created for clarity

General comments around certain loops and longer sections of code were added as needed for clarification as to what the code was actually doing. This was done with the purpose of helping programmers in the future work more efficiently on the code as well as better understand if there is an error in the flow of the logic.

6.0 Conclusion

This project was successfully able to streamline, simplify, and correct the code to the point where it is more readable and works with greater efficiency. The code originally took about 50 seconds to run in its entirety, now it takes under 20 seconds. Much of that is the result of preallocating, but eliminating unneeded code also cut down a lot. The sheer volume of code has been reduced from 219 KB worth of text to 133 KB worth.

Much of this is a result of removing bloated, unnecessary code or preallocating to reduce the memory burden. Going forward, further testing will be needed, namely unit testing on inputs and outputs and ensuring that calculations are correct. Long term, the MATLAB® scripts will need to be converted into .jar files, which are java based and thus one does not need MATLAB® to run them. These will be integrated into the SCENIC user interface, the end goal of most of SCENIC's capabilities.

References

1. NASA_STD_7009A, "Standard for Models and Simulations," *Unlimited Distribution-public release*, 2006-12-07.
2. SCENIC team, "Next Gen Figures of Merit (FoMs) and Attribute Working Groups," *Internal Instructional Document*, 2017.