MIT Libraries | DSpace@MIT

MIT Open Access Articles

# *A Verification Driven Process for Rapid Development of CFD Software*

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

**Citation:** Galbraith, Marshall C., Steven Allmaras, and David L. Darmofal. "A Verification Driven Process for Rapid Development of CFD Software." American Institute of Aeronautics and Astronautics, 2015.

**As Published:** http://dx.doi.org/10.2514/6.2015-0818

**Publisher:** American Institute of Aeronautics and Astronautics

**Persistent URL:** http://hdl.handle.net/1721.1/106675

**Version:** Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

**Terms of use:** Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

# A Verification Driven Process for Rapid Development of CFD Software

Marshall C. Galbraith*, Steven R. Allmaras†, and David L. Darmofal‡
*Massachusetts Institute of Technology, Cambridge, MA 02139*

**Previous work by the authors has demonstrated a high-order fully-automated output-error based mesh adaptation method suitable for solving the Reynolds-Averaged Navier-Stokes equations. The high-order of accuracy is achieved with a discontinuous Galerkin discretization. While the adaptation method has proven to provide significant reduction in computational cost relative to second-order methods, the authors are currently exploring alternate high-order finite element discretizations to further reduce the computational cost. However, the previously developed software framework is not suitable for all discretizations of interest. Hence, a new software framework is being developed with enhanced maintainability and flexibility relative to the previous framework. This paper focuses on strategies employed to accelerate the development of the new software framework. A software development environment that promotes a verification driven process for software development is presented. The development environment encourages developers to incorporate the verification principles of Verification and Validation as part of the software development process to promote maintainability and collaboration. The software development is further accelerated through the use of automatic differentiation, which is used here to automatically compute the linearization of a mathematical model. This paper outlines an implementation of automatic differentiation with minimal computational overhead relative to manually written linearizations.**

## I. Introduction

THE objective of this paper is to describe a method for team-based rapid development of CFD software. The method is designed to promote both maintainability and flexibility of the software framework. Maintainability is achieved through a development environment that is designed to incorporate software verification tools directly into the development process. Incorporating verification tools into the development process not only improves maintainability, but also allows for improved collaboration amongst a team of developers through a continuous integration server with several levels of verification testing. Hence, the verification tools and regression testing allow multiple developers to actively work on separate features, or collaborate on the same feature, of the software framework with confidence that defects are not introduced by the new features.

The flexibility of the software framework is further enhanced through the use of automatic differentiation to compute the linearization of mathematical models. Automatic differentiation relieves the need to implement linearizations of mathematical models, a process that is tedious and error prone. This also improves maintainability by significantly reducing the number of source lines of code that need to be verified by the developers.

A key focus of the authors' research group is the development of a computational fluid dynamics (CFD) method that can autonomously and efficiently solve the compressible Reynolds-averaged Navier-Stokes (RANS) equations to a specified level of discretization error for relevant aerodynamic quantities. Key ingredients in the method are a discontinuous Galerkin (DG) finite element discretization, a preconditioned Newton-Krylov solver, output-based error estimation, and anisotropic simplex- and hex-based adaptation. Over the past ten years, this method has been implemented in the CFD software named *Project X* [1]. This software demonstrated that fully-automated higher-order adaptive methods can be developed for compressible RANS approximations of aerodynamic flows and provide significant savings in computational costs relative to second-order methods.

Most recently, alternatives to the DG finite element discretization are considered in an effort to further reduce computational costs, such as Hybridized DG [2–4] (including Embedded DG [5]) and continuous Galerkin methods (based on SUPG [6]/GLS [7] finite-element formulations). Unfortunately, the current software framework in *Project X* was not designed to incorporate a wide verity of discretizations, and efforts have begun to develop a new software framework. The overall development process is designed so that the resulting software is not only easier to maintain but also more flexible (i.e. allowing new ideas to be quickly tested). Maintainability and flexibility are key qualities for a research environment that places high value on rapid (though careful) evaluation of new ideas.

*Postdoctoral Associate, Department of Aeronautics and Astronautics, AIAA Member.
†Research Engineer, Department of Aeronautics and Astronautics, AIAA Senior Member.
‡Professor, Department of Aeronautics and Astronautics, AIAA Associate Fellow.

A verification driven process for software development that promotes maintainability of the software as well as collaboration amongst software developers is presented here. The verification driven process is based on the principles of Verification and Validation[8] (V&V). Oberkampf and Roy[9] summarize V&V as: "*Verification* is the process of assessing software correctness and numerical accuracy of the solution to a given mathematical model. *Validation* is the process of assessing the physical accuracy of a mathematical model based on comparisons between computational results and experimental data." The primary focus of the verification driven development process is software correctness. This is achieved by following a number of recommendations by Oberkampf and Roy[9] and Kleb et al.[10], namely the use of source code management software, unit tests, source code coverage, dynamic testing, and regression testing. Many of these concepts are based on the Agile[a] principles of software development. This paper outlines the specific tools used to create a development environment that promotes the verification driven process. The development environment is designed to make the verification process seamless so that developers can focus on answering research questions rather than fixing software defects. It is important to note that verification driven development process can be implemented with a number of different software tools besides the ones used here.

The purpose of verification testing is to discover software defects as soon as possible, preferably as soon as they are introduced. Thus, the verification tests suite should be executed as frequently possible, and the test suite must be extensive in order to discover defects. Incorrect modifications to software code can only be detected if the code is executed by the verification test suite. Thus, the verification driven development process is designed to encourage developers to write unit tests concurrently with the development of new software features. However, developers also have the flexibility to only execute the portions of the verification test suite related to the feature they are actively developing. Execution of the entire test suite is automated as part of the continuous integration procedure.

A verification driven development process is essential for continuous integration. Continuous integration is the procedure for sharing source code between developers (typically with a source code management repository), and is most effective when driven by the results of executing the verification test suite. The continuous integration process should encourage developers to commit source code frequently to the repository in order to build a detailed history of the source code changes, minimize differences between developers working copies of the code, and catch mistakes quickly by executing the complete verification test suite. A number of different continuous integration strategies exist. Some strategies prevent developers from committing source code to the repository that does not pass the verification test suite, which ensures that the source code in the repository always passes the verification tests, but restricts developers' ability to collaborate and discourages developers from committing frequently. Other strategies put less restrictions on collaboration by allowing source code that fails the verification tests into the repository, which is subsequently shared with all developers. This comes at the cost of not knowing if the source code in the repository passes all the verification tests, and can hinder developers from collaborating on different features concurrently. A continuous integration strategy is presented here that promotes frequent commits, allows developers to collaborate, and guarantees that the repository retains a copy of the source code that passes all verification tests. This is achieved by only automatically sharing source code that passes the verification test suite amongst developers. Manual sharing of source code that fails the verification test suite using the repository is still possible. Promoting collaboration amongst multiple developers further accelerates the development process.

A key aspect to improving flexibility in the new software framework is the use of automatic differentiation[11–13]. The adaptation process and Newton-Krylov solver in *Project X* both require a complete linearization of the mathematical model without approximations. However, deriving and implementing the linearization is a tedious, time consuming, and error prone endeavor. Research questions associated with *Project X* were sometimes averted due to the daunting task associated with linearization of a particular mathematical model. An integral part of improving the flexibility of the new software framework is the use of automatic differentiation to eliminate the need to manually derive and implement the linearization of a mathematical model. However, automatic differentiation can incur a significant increase in computational overhead. To mitigate the computational cost, an automatic differentiation scheme based on C++ expression templates[14,15] that incurs only a minimal computational overhead compared to a manually written linearization is proposed. Details of both the theory behind automatic differentiation and the expression template implementation are presented.

# II. Software Development Environment

The development environment defined here is a suite of software tools and software development practices used to facilitate and accelerate the development of research focused CFD software. The software tools are used to automatically generate a range of build configurations, to promote unit and integration testing, to verify code coverage of the tests, to verify proper memory management, and to verify adherence to a coding standard. All of these tools are selected with the intent to aid an individual developer in writing verified CFD software. The following sections outline the specific tools selected to create the verification driven development process. However, it is important to note that the specific tool choices presented here are not as important as the principles behind the process.

[a]http://www.agilealliance.org/

## A. Build Configuration

Having robust tools for compiling and linking executables and libraries is an integral part of any software development process. Smaller projects with only a few source code files often rely on one, or maybe a handful, of Makefiles[b]. However, managing portable Makefiles that support a range of compilers and operating systems is challenging. The process of locating dependencies is further complicated by dependencies that may or may not be installed as part of the operating system. Different operating systems may place dependencies in different locations, even when considering only Unix-like operating systems. In addition, the availability of compilers may differ between sites. Hence, the Makefiles need to be configurable for different compilers as well as operating systems.

A number of different tools exist to alleviate this issue of managing build configurations. For example, CMake[c] and Autotools[d] use a set of scripts to generate custom Makefiles for each build configuration. Alternatively, the tools SCons[e] and Waf[f] completely replace Makefiles with a set of Python[g] scripts.

The development environment outlined here utilizes CMake to generate build configurations. CMake is fundamentally designed to simplify the process of locating library dependencies. CMake also provides a user interface for customizing any given build configuration without modification to the scripts. This means that any developer can introduce customizations to a build configuration as needed without modifying the general scripts.

All of the build configuration tools allow for out-of-source builds. That is, the intermediate binary files produced by the compiler are not placed in the same folders as the source code. This not only makes working with source code repositories simpler, but also allows a developer to create a number of different build configurations to, for example, ensure that the source code functions with different compilers or compiler flags. Compiling with many compilers improves the portability of the source code, because not all compilers are created equal. Different compilers can produce different warnings for the same source code. In addition, compiler vendors often introduce extensions to the programming language. These are programming techniques that are not part of the language standard, but the compiler developers deemed important enough to implement. However, this means that not all compilers support all extensions. Thus it is necessary to ensure proper functionality with a range of compilers.

The CMake scripts in the development environment presented here are designed to use the name of the build directory to guide the build configuration. The developers are encouraged to create a directory, often called *build*, where they place all their build configurations. This is of course optional, and all build configuration folders could be placed anywhere the developer deems suitable. However, the scripts are configured to not allow in-source builds to prevent accidental in-source build configurations. Accidentally configuring the build in-source results in a large number of configuration files being generated in the source tree which is tedious to clean up. The build directories inside an example *build* is shown in Fig. 1. The CMake scripts look for keywords in the build directories to guide the build configuration in each directory. All keywords are case insensitive and independent of order which they appear in the build directory name. In this example, the keywords *release*, *debug*, and *coverage* all refer to the compiler flags desired for a build configuration. The keyword *release* enables optimization flags, *debug* enables debug flags, and *coverage* enables flags for generating coverage information. The second keyword indicates the desired compiler, GNU[h], Clang[i], or Intel[j]. The CMake scripts automatically generates the build configuration with the desired compiler when these keywords appear in the name of the build directory.

```
build/
  debug_gnu
  release_gnu
  debug_intel
  release_intel
  debug_clang
  release_clang
  coverage
```

Figure 1: Example Build Configuration Folder Structure

[b] http://www.gnu.org/software/make/
[c] http://cmake.org/
[d] http://www.gnu.org/software/autoconf/, http://www.gnu.org/software/automake/
[e] http://www.scons.org/
[f] https://code.google.com/p/waf/
[g] https://python.org
[h] http://gcc.gnu.org/
[i] http://clang.llvm.org/
[j] https://software.intel.com/en-us/intel-compilers/

It is important to note that all keywords are strictly optional and only provide guidance for the initial build configuration. Once a build directory is initialized, a developer can use CMake's user interface to further customize the build configuration by changing compiler flags, library dependencies, or the compiler. The use of keywords in the build directory names tends to expedite the process of creating new build directories, and mitigates issues with the developer forgetting to configure the build to reflect the name of the directory. The use of the keywords also brings conformity in naming conventions and improves communication within the group of developers.

### B. Unit-testing

The purpose of unit testing is to verify that the software is functioning properly on a component level, i.e., individual functions/subroutines. Regardless of implementation, functions and subroutines operate on input and produces an output, and the purpose of unit testing is to verify their operation. The development environment is designed to minimize the effort required to generate and execute unit tests by automatically generating `make` targets for each unit test file.

Many developers verify proper operation of functions and subroutines through the use of print statements. That is, the developer devises some input for a function, and then visually verifies the output from the function by printing the result to the command line or a file. Once the function produces the expected output, the print statements are typically commented out or removed from the source code. Alternatively, the output is visually verified using a interactive debugger instead of print statements.

Methods that rely on visual verification only ensure that the function is operating properly when it is initially implemented. Furthermore, the function has only been verified to operate correctly with the hardware, operating system, system dependencies, and compiler that the developer used during the initial implementation. Any future modification to the function by any developer, including the original developer, could unintentionally alter the operation of the function. Note that the modification could occur in the function itself, in a subroutine that the function depends on, or in a system library that the function uses. Retaining print statements in the source could provide feedback to a developer about the output of the function, but this would require the developer to possess the knowledge to scrutinize the output in order to verify correct functionality.

Unit testing is intended to provide developers with the means of verifying the functionality of individual software components with any suitable hardware, operating system, or compiler. The verification process for unit testing is similar to the previously described process with print statements. The developer of a function devises inputs for the function that produce an expected output. However, rather than visually verifying the output using print statements, the developer writes a unit test, which is a function itself, that compares the output of the function with the expected output. So unlike the use of visual inspection of the output, where the knowledge of the expected output is retained by the developer, the expected output of the function is retained within the unit test. Hence, the unit test can be verified with a number of hardware, operating system, and compiler configurations. Unit tests also serve as a documentation for developers with examples on how to use the tested functions. Software that relies extensively on unit testing for verification tends to become modular. This is often a direct result of requiring well defined interfaces for various components, such as linear algebra, residual evaluation, partial differential equations, etc., so that they can be tested. A modular software framework is more easily adaptable to needs of new, and often unforeseen, algorithms.

Because a majority of the work to verify functionality of a subroutine is to devise its input, no significant effort beyond the visual verification of functionality is required to implement unit tests. Oberkampf and Roy[9] observed in a university setting that the ratio of debugging to development time spent by students working on even small scientific codes is approximately five times higher for those that did not use unit tests. Hence, unit tests have the potential to significantly accelerate the development process.

Based on the aforementioned benefits, unit tests are an integral part of the verification driven development process presented here. However, it is imperative that the implementation of unit tests requires minimal effort by developers. A number of different unit testing frameworks exist for this purpose; for example Boost-test[k], Google-test[l], and CppUnit[m] for the C++ programming language, pFUnit[n] and FRUIT[o] for Fortran, and CuTest[p] and CUnit[q] for the C programming language, to name a few. Unit testing frameworks exist for many other programming languages as well. Here, the Boost-test framework was selected for the development environment. This decision was influenced by dependencies on additional Boost libraries rather than any particular features of the Boost-test framework.

[k] http://www.boost.org/
[l] https://code.google.com/p/googletest/
[m] http://sourceforge.net/projects/cppunit/
[n] http://opensource.gsfc.nasa.gov/projects/FUNIT/index.php
[o] http://sourceforge.net/projects/fortranxunit/
[p] http://cutest.sourceforge.net/
[q] http://cunit.sourceforge.net/

```
1  #include <boost/test/unit_test.hpp>
2  #include "SparseLinAlg/LinearSolver.h"
3  #include "SparseLinAlg/Matrix_TriDiag.h"
4  #include "SparseLinAlg/Vector.h"
5
6  BOOST_AUTO_TEST_SUITE( SparseLinAlg )
7
8  BOOST_AUTO_TEST_CASE( LinearSolver_UMFPACK )
9  {
10   LinearSolver< Matrix_TriDiag > Solver("UMFPACK");
11
12   //Create the sparse matrix and vectors
13   const unsigned int nRow = 5;
14   Matrix_TriDiag A(nRow);
15   Vector x(nRow), b(nRow), b2(nRow);
16
17   //Initialize the tri-diagonal matrix with rows of -1, 2, -1
18   A.init(-1, 2, -1);
19
20   //Create a vector
21   b[0] = 0.5;
22   b[1] = 1;
23   b[2] = 2;
24   b[3] = 1;
25   b[4] = 0.5;
26
27   //Solve the linear system Ax = b.
28   x = Solver.inverse(A)*b;
29
30   //Compute another vector from the solution
31   b2 = A*x;
32
33   //The vectors should now be the same!
34   for (unsigned int i = 0; i < nRow; i++)
35     BOOST_CHECK_CLOSE( b[i], b2[i], 1e-12 );
36 }
37
38 BOOST_AUTO_TEST_SUITE_END()
```

Figure 2: Example Unit Test of a Sparse Matrix Solver Interface

| Example Unit Test Directory Structure | Make Targets |
|---|---|
| unit/DenseLinAlg/ | make MatrixD_btest |
| CMakeLists.txt | make MatrixS_btest |
| MatrixD_btest.cpp | make DenseLinAlg |
| MatrixS_btest.cpp | |
| | make SurrealD_btest |
| unit/Surreal/ | make SurrealS_btest |
| CMakeLists.txt | make Surreal |
| SurrealD_btest.cpp | |
| SurrealS_btest.cpp | make unit |

Figure 3: Unit Test folder Structure and Automatically Generated Make Targets

An example unit test of a sparse matrix solver is shown in Fig. 2. The class "*LinearSolver*" on line 10 is a general interface for both direct and iterative sparse matrix solvers. This unit test verifies that the interface with the direct solver UMFPACK[16] is functioning properly. The test allocates a tri-diagonal matrix and initializes with a value of -1 on the off diagonals and 2 on the diagonal on line 18. The vector, *b,* is given an initial value on lines 21 through 25, and the solution, *x*, is computed on line 28. A second vector, *b2*, is computed from the multiplication of the matrix *A* and the vector *x*. If the solver interface is functioning properly, the vectors *b* and *b2* should be equal. This equality is verified on lines 34 and 35. An error message is produced if the two vectors are not equal (within a tolerance) when the unit test is executed.

The development environment is designed to make compiling and executing unit tests as simple as possible once a unit test is written. To accomplish this, a convention is adopted that all unit test files are placed in a single directory structure, with a root directory named *unit*, separate from the main source code and suffixed with the name '_btest'. An example of the unit test directory structure is shown in Fig. 3. The files 'CMakeLists.txt' are the CMake script files and only contain a single function call. The argument to this CMake function is the names of the libraries on which the unit tests in each respective directory depend. The CMake function searches the directory for files with the suffix '_btest' and automatically generates Makefile targets to compile and execute the unit tests. These `make` targets are also shown in Fig. 3. Hence, a developer only needs to create the new unit test file in the appropriate directory and re-execute CMake in order to generate the `make` targets. These `make` targets are designed to both compile and execute the unit test with a single `make` command. A `make` target is generated for each unit test file, which allows a developer to execute each unit test file separately. This is useful when, for example, refactoring source and the developer needs to reintegrate components one at a time. Hence, unit tests can be incrementally brought back to functionality and executed rather than delaying testing until all of the refactoring is completed. A `make` target is also generated for each folder, with the same name as the folder, in the unit test directory structure. This allows a developer to execute all unit tests in a folder with a single command, which can be useful when testing features that span multiple unit test files. Finally, the `make` target *unit* allows a developer to execute all unit tests with a single command.

### C. Coverage

Unit testing answers the question "do the tests pass or fail?" and coverage information answers the question "what source code is tested?" Coverage information provides the developer with feedback about which lines of code are reached during execution of a test. The GNU compiler suite instruments an executable with coverage tracking by adding the flag *--coverage* to the compile and link flags. The software package LCOV[r], which is a set of Perl[s] scripts, is used to produce a set of HTML documents from the GNU coverage data. Example HTML documents from LCOV are shown in Fig. 4. The HTML documents produced by LCOV contain a main page with an overall summary. This page provides links to HTML documents that show specifically which lines of code are executed and which lines of code are not. Lines that are executed as part of a test are highlighted with blue, and lines that are not executed are highlighted with red.

The CMake scripts are designed to automatically provide additional make targets to simplify the process of generating coverage HTML documents. Coverage information for any unit test executable can be generated by a developer by appending '_coverage' to the unit test make targets as shown in Fig. 5. However, these targets are only available if the build is configured to use the coverage compiler flags. The last make target *coverage_show* opens the main HTML document with the default web browser.

Having make targets for each individual unit test provides the developer with the means to ensure that each unit test file provides complete coverage of the piece of source code it is intended to test. If the coverage is not complete, unintentional modifications to untested code will not be caught by the test suite. It is important to note that the coverage information provided by the GNU compiler is cumulative. That is, if a developer executes two unit tests sequentially, the coverage information will contain the coverage data for both executables. Hence, the make target *coverage_clean* is provided to remove old coverage files if needed.

[r] http://ltp.sourceforge.net/coverage/lcov.php
[s] http://www.perl.org/

**LCOV - code coverage report**

| | | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Current view: | top level | | | | |
| Test: | coverage.info | Lines: | 5500 | 5805 | 94.7 % |
| Date: | 2014-05-02 | Functions: | 10093 | 10680 | 94.5 % |
| Legend: | Rating: low: < 75 % medium: >= 75 % high: >= 90 % | Branches: | 10522 | 25729 | 40.9 % |

| Directory | Line Coverage | | | Functions | | Branches | |
|---|---|---|---|---|---|---|---|
| BasisFunction | | 95.7 % | 396 / 414 | 84.0 % | 158 / 188 | 23.7 % | 81 / 342 |
| DenseLinAlg | | 100.0 % | 4 / 4 | 100.0 % | 31 / 31 | - | 0 / 0 |
| DenseLinAlg/DynamicSize | | 99.1 % | 779 / 786 | 98.0 % | 1705 / 1739 | 35.1 % | 1422 / 4051 |
| DenseLinAlg/DynamicSize/MatMul | | 98.8 % | 85 / 86 | 87.5 % | 14 / 16 | 32.8 % | 143 / 436 |
| DenseLinAlg/DynamicSize/lapack | | 100.0 % | 40 / 40 | 100.0 % | 8 / 8 | 28.4 % | 42 / 148 |
| DenseLinAlg/StaticSize | | 100.0 % | 414 / 414 | 93.3 % | 3635 / 3896 | 51.3 % | 2428 / 4733 |
| DenseLinAlg/StaticSize/MatMul | | 97.5 % | 39 / 40 | 98.9 % | 89 / 90 | 52.2 % | 830 / 1590 |
| DenseLinAlg/tools | | 100.0 % | 21 / 21 | 90.0 % | 9 / 10 | 85.7 % | 12 / 14 |
| Python | | 100.0 % | 282 / 282 | 94.4 % | 219 / 232 | 43.0 % | 773 / 1798 |
| Quadrature | | 100.0 % | 495 / 495 | 100.0 % | 40 / 40 | 55.0 % | 308 / 560 |
| Residual | | 79.3 % | 803 / 1013 | 93.4 % | 171 / 183 | 33.2 % | 576 / 1734 |
| SparseLinAlg | | 99.6 % | 264 / 265 | 88.6 % | 723 / 816 | 34.6 % | 347 / 1003 |
| SparseLinAlg/Direct | | 96.0 % | 288 / 300 | 97.8 % | 45 / 46 | 50.0 % | 332 / 664 |
| SparseLinAlg/Krylov | | 97.1 % | 101 / 104 | 61.3 % | 19 / 31 | 23.5 % | 290 / 1236 |
| SparseLinAlg/Preconditioners | | 100.0 % | 14 / 14 | 51.7 % | 15 / 29 | 33.3 % | 4 / 12 |
| SparseLinAlg/tools | | 100.0 % | 14 / 14 | 88.9 % | 8 / 9 | 62.5 % | 10 / 16 |
| Surreal | | 99.7 % | 983 / 986 | 98.4 % | 2996 / 3046 | 39.2 % | 2515 / 6414 |
| pde/CauchyRiemann | | 83.3 % | 45 / 54 | 57.1 % | 68 / 119 | 25.4 % | 62 / 244 |
| pde/NS | | 93.8 % | 360 / 384 | 96.6 % | 112 / 116 | 46.6 % | 257 / 552 |
| tools | | 82.0 % | 73 / 89 | 80.0 % | 28 / 35 | 49.5 % | 90 / 182 |

Generated by: LCOV version 1.10

(a) LCOV Front Page

Top

**LCOV - code coverage report**

| | | | Hit | Total | Coverage |
|---|---|---|---|---|---|
| Current view: | top level - tools - SANSException.cpp (source / functions) | | | | |
| Test: | coverage.info | Lines: | 14 | 29 | 48.3 % |
| Date: | 2014-05-02 | Functions: | 5 | 7 | 71.4 % |
| Legend: | Lines: hit not hit \| Branches: + taken - not taken # not executed | Branches: | 13 | 50 | 26.0 % |

```
         Branch data        Line data    Source code
   1                    :              : #include "SANSException.h"
   2                    :              : #include <iostream>
   3                    :              : #include <stdarg.h>
   4                    :              :
   5 [ + - ][ + - ]:           554 : SANSException::SANSException() : errString("")
             [ + - ]
   6                    :              : {
   7         [ + - ]:           277 :   barrier = "\n#================================================#\n";
   8                    :           277 : }
   9                    :              :
  10                    :              : //==========================================================================
  11                    :             0 : char const* SANSException::what() const throw()
  12                    :              : {
  13         [ # # ]:             0 :   outString  = barrier;
  14         [ # # ]:             0 :   outString += "#                     SANS ERROR                     #";
  15         [ # # ]:             0 :   outString += barrier;
  16         [ # # ]:             0 :   outString += "\n";
  17                    :              :
  18         [ # # ]:             0 :   outString += errString;
  19                    :              :
  20         [ # # ]:             0 :   outString += "\n";
  21         [ # # ]:             0 :   outString += barrier;
  22         [ # # ]:             0 :   outString += "\n";
  23                    :              :
  24                    :              :   //boost test has a limit on the string size due to the use of vsprintf
  25                    :              :   //Backtrace information can be much longer than the buffer, so just dup
  26                    :              :   //it to the screen
  27         [ # # ]:             0 :   std::cout << outString.c_str();
  28                    :              :
  29         [ # # ]:             0 :   return "";
  30                    :              : }
  31                    :              :
  32                    :              : //==========================================================================
  33                    :           436 : AssertionException::AssertionException( const std::string assertion )
  34                    :              : {
  35         [ + - ]:           218 :   errString += "Developer Error!!!\n";
  36 [ + - ][ + - ]:           218 :   errString += "Assertion \"" + assertion + "\" failed.";
             [ + - ]
  37                    :           218 : }
  38                    :              :
  39                    :              : //==========================================================================
  40                    :             0 : DeveloperException::DeveloperException( const std::string message )
  41                    :              : {
  42         [ # # ]:             0 :   errString += "Developer Error!!!\n";
  43         [ # # ]:             0 :   errString += message;
  44                    :             0 : }
  45                    :              :
  46                    :              : //==========================================================================
```

(b) LCOV Code Coverage

Figure 4: LCOV Example Output

| Example Unit Test Directory Structure | Make Targets to Generate Coverage HTML Files |
|---|---|
| unit/DenseLinAlg/<br>  CMakeLists.txt<br>  MatrixD_btest.cpp<br>  MatrixS_btest.cpp<br><br>unit/Surreal/<br>  CMakeLists.txt<br>  SurrealD_btest.cpp<br>  SurrealS_btest.cpp | make MatrixD_btest_coverage<br>make MatrixS_btest_coverage<br>make DenseLinAlg_coverage<br><br>make SurrealD_btest_coverage<br>make SurrealS_btest_coverage<br>make Surreal_coverage<br><br>make unit_coverage<br><br>make coverage_show<br>make coverage_clean |

Figure 5: Unit Test folder Structure and Automatically Generated Make Targets for Coverage Information

### D. Coding Standard Checking

A coding standard is a set of guidelines that define the style of the software code, e.g. indentation, variable naming conventions, and other general syntactical choices. Having a coding standard promotes collective ownership of the source code. That is, no individual developer has ownership of any part of the code. Instead, the group has ownership of the code and equal responsibility for the quality of the code as a whole. Hence, any developer is permitted to make changes to any part of the code, whether to implement a new feature or fix a flaw. Again, the intent here is to promote collaboration within the group of developers. Ideally, any developer will be able to look at a piece of code without the urge to fix any of the formatting nor recall if they were the original author of that particular piece of source code.

Kleb et al.[10] relied on a contractor to manually verify adherence to their set coding standard (with the intent to eventually automate the process). A number of software tools for style checking are available, such as KWStyle[t], AStyle [u], and Vera++[v] to name a few. Style checking using Vera++ has been incorporated into the development environment. The CMake scripts are configured to automatically detect source code files, and create Makefiles that execute Vera++ to check the formatting style of any file that has been modified. Hence, the style checking occurs automatically as part of the build process. The Vera++ checks does not modify any files, rather it alerts the developer to any formatting that does not adhere to the coding standard.

### E. Static Code Analysis

Static analyzer codes are designed to complement the error and warning checking performed by the compiler. These tools are designed to find errors and coding constructs that are considered generally unsafe but that the compiler will likely overlook. This includes, for example, accessing arrays out of bounds, memory leaks, and the lack of a copy constructor when a class contains pointers to allocated memory (C++), to name a few. A number of static analyzers exist that are designed for individual or range of programming languages, such as Lint[w], Cppcheck[x], Clang[y], and many other commercially available software packages. Some Integrated Development Environment (IDE) software, such as Eclipse[z] and Xcode[aa], also have built in static analyzers that will highlight potential errors as the code is actively developed.

Cppcheck is included in the build process by configuring the CMake scripts to run the static analyzer prior to invoking the compiler on a given source code file. Hence, the static analyzer checks each file as just before it is compiled. Cppcheck also has a whole program analysis, and an additional make target, *cppcheck*, is available to the developer to invoke this analysis.

### F. Dynamic Code Analysis

Memory checking serves to verify proper memory management, such as memory is not leaked by the software, no operations step beyond array bounds, and uninitialized memory is not used. A memory leak occurs in a program when memory is allocated, but not properly deallocated. This situation could result in the program continuously allocating

[t] http://kitware.github.io/KWStyle/
[u] http://astyle.sourceforge.net/
[v] https://bitbucket.org/verateam
[w] http://www.splint.org/
[x] http://cppcheck.sourceforge.net/
[y] http://clang-analyzer.llvm.org/
[z] http://www.eclipse.org/
[aa] https://developer.apple.com/xcode/

memory until all available hardware memory is consumed. The software package Valgrind[ab] is a general framework for dynamic analysis tools. Valgrind contains a memory error detector, two thread error detectors, a cache profiler, a call graph cache profiler, and a heap usage profiler.

The heap and stack memory error detectors and profiler of Valgrind are incorporated into the development environment. The heap memory error detector verifies proper allocation/deallocation of memory, usage of arrays within their bounds, and ensures that no uninitialized memory is used. The stack memory error detector ensure proper usage of static sized arrays that are placed on the stack. The profiler, callgrind, provides statistics for a developer to examine execution timing statistics. Additional make targets are automatically generated by the CMake scripts to promote the usage of Valgrind as part of the development process. These make targets are the unit test make targets suffixed with '_memcheck', '_stackcheck' and '_callgrind'. Some example make targets for executing Valgrinds heap memory error detector are shown in Fig. 6. The make target *memcheck* executes Valgrind on all unit tests that correspond to individual unit test files.

| Example Unit Test Directory Structure | Make Targets to Execute Valgrind |
|---|---|
| | `make MatrixD_btest_memcheck` |
| `unit/DenseLinAlg/` | `make MatrixS_btest_memcheck` |
| `CMakeLists.txt` | `make DenseLinAlg_memcheck` |
| `MatrixD_btest.cpp` | |
| `MatrixS_btest.cpp` | `make SurrealD_btest_memcheck` |
| | `make SurrealS_btest_memcheck` |
| `unit/Surreal/` | `make Surreal_memcheck` |
| `CMakeLists.txt` | |
| `SurrealD_btest.cpp` | `make unit_memcheck` |
| `SurrealS_btest.cpp` | |
| | `make memcheck` |

Figure 6: Unit Test folder Structure and Automatically Generated Make Targets for Valgrind

## G. Regression-testing

Regression testing often refers to tests that execute the software with a given set of inputs, and compares the output with previously recorded results. Note that the previously recorded output in this situation are typically not known to be analytically correct, but rather simply the output produced by the code at some point in time where the developer believed the output to be correct. Kleb et al.[10] refers to the files that store the previously recorded results as "golden files". Unfortunately, this type of testing does not actually test if the code is functioning properly, but rather simply tests if the code has changed. Furthermore, because these tests execute large portions of software code (by definition), a regression test failure only implies that something in the code no longer functions as expected, but gives no guidance as to what part of the code has failed. In contrast, a unit tests failure, which verifies the software on the component level, can give a direct indication of what component of the software is not functioning properly. Finally, it's not possible to define when a golden file should be generated. Suppose that a subtle bug was present in the software when a golden file was generated. The resolution of the bug could lead to a different output and hence a failure of the regression test even though the software has been improved. Hence, the authors have chosen to emphasize code verification through unit testing rather than regression testing, and hence minimize the dependence on regression tests for verification. Regression tests are limited to cases where the discretization error in the solution can be computed directly. This includes cases where the analytical solution is known, such as test cases proposed by Ghia et al.[17], and cases with analytical solutions from the High-Order Workshops[18]. The Method of Manufactured Solutions[19–21] is also used to broaden the spectrum of analytical solutions. In each case, the software is verified by computing the order of accuracy on a series of refined meshes and compared with the theoretically expected order of accuracy. The order of accuracy of the software is verified with a range of partial differential equations, such as scalar advection-diffusion, Cauchy-Riemann, Euler, Navier-Stokes, and RANS equations. Given the longer execution time of regression tests, coverage information is used to carefully select test cases to minimized redundancy.

## H. Method of Manufactured Solutions

Oberkampf and Roy[9] suggested that the Method of Manufactured Solutions (MMS) is one of the most rigorous tests for verification of scientific computing software. MMS is applied by selecting a desired analytical solution and modifying the governing PDE with a source term such that the selected analytic solution satisfies the modified PDE. Oberkampf and Roy, as well as many other authors, recommend implementing MMS source terms through the use of

[ab] http://valgrind.org/

symbolic manipulation software. Unfortunately, this approach often leads to several pages of source code produced by the symbolic manipulation software that may require modification before it can be compiled. In addition, the method requires that multiple different source terms be derived depending on the choice of manufactured solution. The large quantity of source code generated from this process is difficult to manage and ensure correctness.

The strong form of the equations can be used to simplify the process of generating MMS source term. Consider the steady two-dimensional inviscid Euler equations written in conservative form,

$$\nabla \cdot \vec{F}(U) = 0, \tag{1}$$

where the conservative variables are $U = \begin{bmatrix} \rho, & \rho u, & \rho v, & \rho E \end{bmatrix}^T$, the inviscid fluxes are

$$\vec{F}(U) = \left[ \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho u v \\ \rho u H \end{pmatrix}, \begin{pmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ \rho v H \end{pmatrix} \right], \tag{2}$$

$\rho$ is the density, $\begin{bmatrix} \rho u, & \rho v \end{bmatrix}$ is the momentum vector, $\rho E$ is the total energy, $H$ is the total enthalpy, and $p$ is the static pressure. Assuming that the manufactured solution is given as a set of analytical functions in terms of primitive variables $Q(x,y) = \begin{bmatrix} \rho(x,y), & u(x,y), & v(x,y), & p(x,y) \end{bmatrix}^T$, Eq. 1 is modified to include the source term, $S_{mms}(Q)$, as

$$\nabla \cdot \vec{F}(U) = S_{mms}(Q), \tag{3}$$

where

$$S_{mms}(Q) = \nabla \cdot \vec{F}(Q). \tag{4}$$

By definition, the exact solution to Eq. 3 is

$$U = U(Q). \tag{5}$$

Traditionally, the source term in Eq. 4 is derived through symbolic manipulation software, which results in long complex code depending on the complexity of the analytical functions $Q$. A simpler method is to implement a subroutine that computes the strong form of the partial differential equation using the chain rule, i.e.

$$S_{mms}(Q) = \nabla \cdot \vec{F}(Q) = \begin{pmatrix} \frac{\partial \rho}{\partial x} u + \rho \frac{\partial u}{\partial x} \\ \frac{\partial \rho}{\partial x} u^2 + 2\rho \frac{\partial u}{\partial x} + \frac{\partial p}{\partial x} \\ \frac{\partial \rho}{\partial x} u v + \rho \frac{\partial u}{\partial x} v + \rho u \frac{\partial v}{\partial x} \\ \frac{\partial \rho}{\partial x} u H + \rho \frac{\partial u}{\partial x} H + \rho u \frac{\partial H}{\partial x} \end{pmatrix} + \begin{pmatrix} \frac{\partial \rho}{\partial y} v + \rho \frac{\partial v}{\partial y} \\ \frac{\partial \rho}{\partial y} v u + \rho \frac{\partial v}{\partial y} u + \rho v \frac{\partial u}{\partial y} \\ \frac{\partial \rho}{\partial y} v^2 + 2\rho \frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} \\ \frac{\partial \rho}{\partial y} v H + \rho \frac{\partial v}{\partial y} H + \rho v \frac{\partial H}{\partial y} \end{pmatrix}. \tag{6}$$

The implementation of the strong form is dependent on the choice of variables in $Q$, and the variables used here were chosen to simplify the equations. The strong form of the partial differential equation is also needed for SUPG[22] and GLS[7] finite element discretizations, and is hence readily available for such discretizations. Given the simplicity, the strong form of the partial differential equation is easily incorporated into codes that rely on other discretizations. The subroutine that computes Eq. 6 takes $Q$ and $\nabla Q$ as it's arguments, which, by definition, are analytically available. Strong forms of second order partial differential equations are also simple to implement; in which case the Hessian of $Q$ is required. Hence, a single subroutine, for any given partial differential equation, can be used to verify proper functionality of the code with a range of analytical functions. Since the analytical solution is known, by design, the rate of the error decay with grid refinement is compared with the theoretical expected order of accuracy as a measure of correctness.

## III. Continuous Integration

Continuous Integration aids the process of collaboration with multiple developers. It is first and foremost a mindset, and should not be confused with the tools used to implement the process. The purpose of Continuous Integration is to mitigate conflicts between changes to the software introduced by multiple developers working with separate copies of the software, i.e. it is intended to promote collaboration. Collaboration can be in the form of multiple developers implementing independent features of the software, multiple developers collaborating on the same feature of the software, or simply providing developers with the ability to assist each other.

The implementation of a Continuous Integration process typically relies on source code management software (i.e. CVS[ac], SVN[ad], Git[ae], Mercurial[af], etc.) with a centralized repository, an automated build process, a suite of tests to

[ac] www.cvshome.org

[ad] http://subversion.apache.org/

[ae] http://git-scm.com/

[af] http://mercurial.selenic.com/

verify the functionality of the built software, and a system for reporting the success/failure of the build and testing process. The source code management software is the primary tool that enables collaboration. A repository not only holds the most recent version of the software, it also tracks the incremental changes introduced by developers. Source code management software also assists with resolving conflicts where multiple developers have modified similar regions of source code files. However, an efficient use of a source code repository requires all developers to commit their changes to the repository as often as possible. Not only does committing often help prevent conflicts, it also reduces the amount of code changes that need to be scrutinized in case changes have caused failures in the test suite. Hence, errors are hopefully caught as soon as they are introduced into the code. In order to catch errors quickly, most Continuous Integration procedures utilize a server-based automated testing procedure that is triggered when a developer commits new code to the source code repository. There are three common strategies for choosing when to test during the commit process, pre-commit, post-commit, and pre-merge. The key desired features of the commit testing strategy are:

- Automated server side test suite build and execution
- Promotes frequent commits
- Developers are able to collaborate even with failures in the test suite
- Synchronization to multiple machines
- Repository always contains source code that passes *all* tests

Shifting the burden of executing the test suite to the repository server ensures that the test suite is executed regularly and increases the odds of discovering errors as soon as they are introduced. Furthermore, the likelihood of catching errors as they are introduced is increased if developers have the ability to, and are encouraged to, commit frequently. The third requirement to allow collaboration even when there are failures in the test suite is imperative to allow for collaboration amongst developers on complex feature development. Ideally a subset of the developers should have the ability to share code that contains test failures without sharing those failures with the remaining developers. The fourth requirement deals with the prospect of developing in a heterogeneous computing environment. It is important that the test suite passes on all systems deemed supported. Hence, each individual developer needs to have the ability to synchronize the source code between different machines with the supported systems. Finally, retaining a copy of the source code in the repository that passes all tests means that a version of the software is always readily available for production use. The benefits and deficiencies of three different commit testing strategies with respect to these requirements are discussed in the following sections.

### A. Pre-commit Testing

Pre-commit testing builds and executes the test suite before any code is allowed into the repository. This ensures that the repository only contains code that has passed all tests and can never contain any code with test failures. This type of testing can be executed on either a developer's workstation or on a testing server. One possible implementation of this strategy is to ask all developers to execute the test suite before committing any code to the repository. Hence, it is left to the developer to remember to run the tests before performing a commit. This strategy is unfavorable as inevitably a developer may forget/ignore to run the tests before a commit and hence possibly commit failing code to the repository. A more rigorous implementation is to configure the source code repository to trigger the build and execution of tests when a new commit is received on the repository server. If there is a failure in the test suite, the new commit is rejected automatically from the repository and the developer responsible for the commit is notified. The configuration satisfies the first two requirements of automation and retaining functioning code in the repository, but it does not meet the remaining features. Most notably, this strategy not only discourages but completely disallows frequent commits to the repository. That is, a developer can only commit to the repository once a new feature has been completely implemented and verified. Performing incremental commits as parts of a new feature are implemented is simply not allowed. Of course, this also means that the repository cannot be used as a mechanism to synchronize the source code between multiple machines unless all tests pass. This strategy also renders collaboration impractical between multiple developers on a new feature that requires sharing the source code in a state with test failures.

### B. Post-commit Testing

Post-commit testing allows all new commits into the source code repository without first testing them. A typical implementation will configure the source code management to trigger a build and execution of the test suite with the newly accepted commit. If there is a failure in the test suite, either the responsible developer or the entire development team is notified of the failure. This strategy only satisfies the automation requirement. The inherent nature of allowing all commits into the repository means that any commit could potentially cause the repository to contain code that does not pass all tests. While post-commit testing does allow for synchronization of the source code to multiple machines

through the repository, the inherent nature of post-commit testing exposes the new commit to all developers. Hence, developers are reluctant to use the repository for synchronization before there is certainty that the test suite will execute without failure. Similarly, the post-commit testing discourages developers from collaborating on new features that are partially completed or temporarily cause the testing suites to fail. This is also the reason why this strategy discourages frequent commits.

### C. Pre-merge Testing

Pre-merge testing leverages the branching and merging abilities of source control management software. Branching is a technique that allows developers to manage multiple working copies of source code within a single repository. Typically, branching is used to create a temporary feature branch that is used to implement a new feature. Once the feature is implemented, the temporary feature branch is merged back into the *master* branch and the new feature is part of the main code. If the feature is not satisfactory, the temporary feature branch can be discarded without any harm to the *master* branch. The commands to merge the feature branch into the *master* branch is typically initiated by the developer. In the pre-merge testing strategy, the developer pushes the feature branch to the repository instead of merging the changes into the *master* branch. After the updated feature branch is received by the repository, the test suite is executed on the feature branch automatically by the testing server. Upon a successful build and execution of the test suite, the testing server merges the feature branch into the *master* branch. This strategy is similar to the post-commit testing strategy in the sense that all commits to a feature branch are accepted into the repository. However, it is different because the feature branch is separate from the *master* branch. Hence, even though all commits are accepted into the repository, only changes that do not fail the test suite are shared with other developers. Therefore, developers can commit frequently without concern if any given commit fails the test suite. If a given commit fails the test suite, those code changes are not shared with the other developers even though they are accepted into the repository. This means that the repository can be used for synchronization purposes. This also means that developers are able to collaborate on code that fails the test suite. Since any given commit that fails the suite is not shared with all developers, a subset of developers working on the new feature can collaborate even when there are failures in the test suite. Finally, the strategy of testing before merging means that a dedicated branch can be used to retain a copy of the source code that passes ALL tests at all times. If any feature branch added to the repository fails the test suite, those changes are not merged into the dedicated branch.

### D. Adopted Continuous Integration Process

Git is used for source code management, and Jenkins[ag] as the continuous integration server. The specific tools selected to implement the continuous integration process are not critical, and the process could be implemented using other software packages. However, Git was selected because it was designed with the intent of simplifying the branching and merging process, and Jenkins was chosen because it has the pre-merge testing feature built into it. Jenkins is also supported by a large user base and relies on a plug-in framework to allow a number of users to contribute to the project. It has a web based interface for both configuring Jenkins as well as individual testing actions.

The Git repository is configured to contain several different branches. Each developer is provided with a personal development branch in the git repository where they can perform their development work. The individual developer branches follow the naming convention *username/develop*, where *username* is the login name of the developer on the repository server, and Jenkins is configured to only execute tests for branches that matches the pattern */develop*. A branch named *develop* is used to synchronize the source code amongst developers, and the *master* branch contains the copy of the source code that has passed all tests. Sharing of source code amongst the developers is achieved by configuring each developer's local Git repository to merge source code from both their personal branch and the shared *develop* branch on the repository server into the personal branch of the local repository on their workstation.

Jenkins is configured to compile and execute the commit test suite after any commit of the developer branches to the main repository. Before building the test suite, Jenkins checks out the developer branch and merges it with the *develop* branch in a temporary workspace on the testing server. Note that this merge is not performed in the main repository. Performing the merge before building and executing the test suite means that the result of the merge is tested. No merge conflicts will be created from the merge with the *develop* branch if the developer has kept their development branch up to date. After a successful completion of the commit test suite, the merged *develop* branch is committed back to the repository server. At this point, the new features introduced by the developer are available to all other developers. If the commit test suite fails, an email is sent to the developer, and the temporary *develop* branch is not committed back to the main repository.

[ag] http://jenkins-ci.org/

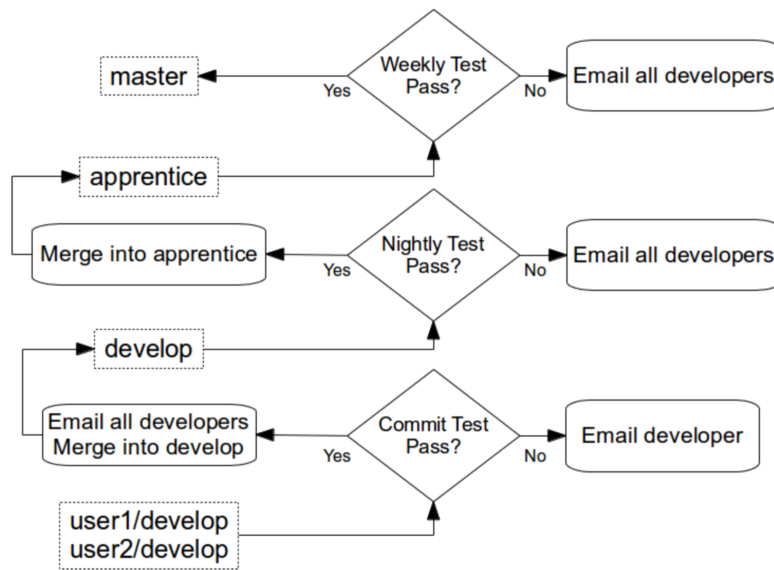


Figure 7: Complete Continuous Integration Process

Ideally, all verification tests would be executed upon every commit by a developer. Unfortunately, the execution time for all verification tests cannot be kept to reasonable duration on the order of minutes. Hence, the verification tests are separated into three different groups based on the required execution time. The three verification test groups are, in order of increasing execution times, *Commit*, *Nightly*, and *Weekly*. The complete testing process is illustrated in Fig. 7. The source code is only merged into a new branch upon successful completion of a testing group. That is, source code committed by a developer from a *username/develop* branch is merged into the *develop* branch if the *Commit* tests pass, and an email is sent to all developers to notify them that they should update their local copies of the source code. On a nightly basis, the source code in the *develop* branch is tested and upon success is merged in to the *apprentice* branch. Finally, the source code in the *apprentice* branch is merged into the *master* branch when weekly tests pass. Note that a test failure results in an email sent to developers and the merge is not performed. Hence, only code that passes the tests is automatically propagated towards the *master* branch.

Table 1: Commit Verification Tests

| Verification Test | Passing Requirement |
|---|---|
| Compile with GNU, Clang, and Intel | No Errors or Warnings |
| Style Checking with Vera++ | No Errors or Warnings |
| Execute Unit Tests | No Errors |
| Static Analysis with Cppcheck | No Errors or Warnings |
| Code Coverage | Line Coverage cannot Decrease |

### *1. Commit Verification Tests*

The commit verification tests need to execute as quickly as possible to promote collaboration, but also need to be thorough enough to catch any compiler or functional errors created by a developer. The commit test executes the Vera++ checks and monitors the build log for compiler warnings. If Vera++ or compiler warnings are produced during the build, the commit is considered a failure and not merged with the *develop* branch. In addition, compiler warnings and errors are tracked during the build. The commit is also marked as a failure if any compiler warnings are produced. Appropriate compiler flags are used to check for all possible warnings, such as the flags -Wall, -Wextra (GNU), and -Weverything (Clang). While compiler warnings do not necessarily imply improper code, elimination of compiler warnings often leads to cleaner and less error prone source code. Unit tests only require seconds to execute, by definition, and are used to verify functionality of all components. Any failure in the unit tests results in a failed commit. In addition, the coverage of the unit tests from the commit is logged and the percentage of lines covered by the unit tests is not allowed to decrease. A reduction in the percentage of lines covered also results in a failed commit. Requiring the coverage to increase serves to remind developers to create unit tests for newly written code. If necessary (and appropriate), the acceptable percentage coverage threshold can be reduced manually through the Jenkins user interface. This might occur when functions are refactored or deprecated code is removed. If all tests pass, the commit is marked as successful and the merged commit is pushed to the main repository. Hence, the commit verification tests

ensure proper build, formatting, and component functionality of the software. This is deemed sufficient to share the commit with other developers.

#### 2. *Nightly Verification Tests*

The nightly verification tests are reserved for tests that require hours to execute. Thus, the nightly verification tests execute the regression tests, i.e., each test that executes on the order of minutes, and executes memory checking on the unit tests. The memory checking is executed on a nightly basis as it generally increases execution time by an order of magnitude. The latest source code in the *develop* branch is tested during the nightly verification tests. Thus, it is possible for a commit to introduce a memory error in the unit tests or a functional error in the regression tests that would be shared with all developers through the *develop* branch. However, this is deemed an acceptable compromise between promoting collaboration and preventing a single developer from sharing errors with the group. The *develop* branch is merged with the *apprentice* branch upon a successful nightly regression test. Hence, the apprentice branch always contains source code that has passed both the commit and nightly verification tests.

#### 3. *Weekly Verification Tests*

The weekly verification testing is reserved for tests that executes on the order of a day. One intent of developing with unit tests and coverage information is to limit the number of tests that require long execution times. However, some three-dimensional regression tests will inevitably require long execution times. Currently, only memory checking of the regression tests is performed using the source code from the most recent *apprentice* branch on a weekly basis. If the weekly verification test is a success, the source code in the *apprentice* branch has passed *all* tests, and the *apprentice* branch is merged into the *master* branch. Thus, the *master* branch always contains source code that has passed *all* of the verification tests. Any non-development use of the software would use the source code from the *master* branch.

## IV. Automatic Differentiation

Automatic differentiation is different from symbolic differentiation and numerical differentiation. Symbolic differentiation often relies on symbolic manipulation software, or manual derivations, to obtain the linearization of a mathematical model. The analytical expression of the linearization to the mathematical model must be translated into source code once it is derived. Symbolic manipulation software can often translate the resulting expression into source code. However, the resulting source code is often inefficient, hard to read, and may require some modifications before it can be compiled. In addition, constructing accurate unit tests to verify the symbolic linearization is challenging. The previous software framework, *Project X*, relied on symbolic differentiation to compute linearizations, and the correctness of the manually written linearization is often questioned when unexpected behavior is observed.

Numerical differentiation relies on finite difference approximations to compute the linearization of a mathematical model. However, finite difference approximations can suffer from round-off errors and subtractive cancellation, and an optimal step size is challenging to obtain. Small changes in the step size can lead to significantly different values in the resulting linearization[11,13]. This is particularly true if the mathematical model incorporates logical statements. In such cases, the logical path of the mathematical model can be altered by the choice in the step size, which results in an incorrect linearization.

Automatic differentiation linearizes the source code mathematical model directly. The method exploits the fact that software, no matter how complicated, executes a series of arithmetic operations (addition, subtraction, multiplication, and division) and elementary functions (sin, cos, exp, etc.). The linearization of these operations can be computed analytically by repeatedly applying the chain rule. Automatic differentiation is usually implemented through the use of a pre-processor or custom data types.

Martins et al.[13] demonstrated the use of custom data types that track the partial derivative with respect to one variable. Hence, multiple function evaluations are required to compute partial derivatives. Similar to Phipps and Pawlowski[23], a custom data type is presented here that computes all partial derivatives with a single function evaluation. The data type, referred to as *Surreal*, stores a value of a variable, and a vector of its derivatives. Derivatives are computed using the chain rule though a complete set of C++ elementary functions and operator overloaded arithmetic operators. Details of the implementation of the *Surreal* data type is presented in the following example where the notation,

$$f = \left\langle \ \hat{f}; \quad \left[ \begin{array}{ccc} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array} \right] \right\rangle \tag{7}$$

is used to represent the *Surreal* variable $f$ with a value of $\hat{f}$ and a vector of derivatives $\left[ \begin{array}{ccc} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array} \right]$. If $f$ is given by,

$$f\left(x_1, x_2, x_3\right) = x_1 x_2 + a x_3 \tag{8}$$

then the derivatives of $f$ with respect to the vector $\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$ are given by,

$$\frac{\partial f}{\partial x_1} = x_2 \qquad \frac{\partial f}{\partial x_2} = x_1 \qquad \frac{\partial f}{\partial x_3} = a. \tag{9}$$

The *Surreal* arguments $x_1$, $x_2$, and $x_3$ for the function $f(x_1, x_2, x_3)$ are,

$$\begin{aligned} x_1 &= \left\langle\ \hat{x}_1;\ \begin{bmatrix} \frac{\partial x_1}{\partial x_1} & \frac{\partial x_1}{\partial x_2} & \frac{\partial x_1}{\partial x_3} \end{bmatrix} \right\rangle = \left\langle\ \hat{x}_1;\ \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \right\rangle \\ x_2 &= \left\langle\ \hat{x}_2;\ \begin{bmatrix} \frac{\partial x_2}{\partial x_1} & \frac{\partial x_2}{\partial x_2} & \frac{\partial x_2}{\partial x_3} \end{bmatrix} \right\rangle = \left\langle\ \hat{x}_2;\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right\rangle \\ x_3 &= \left\langle\ \hat{x}_3;\ \begin{bmatrix} \frac{\partial x_3}{\partial x_1} & \frac{\partial x_3}{\partial x_2} & \frac{\partial x_3}{\partial x_3} \end{bmatrix} \right\rangle = \left\langle\ \hat{x}_3;\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \right\rangle. \end{aligned} \tag{10}$$

Note that the derivative vector for each argument is trivial to evaluate. The function value and derivatives of Eq. 8 are computed through repeated use of the chain rule as,

$$\begin{aligned} f &= x_1 x_2 + a x_3 \\ &= \left\langle\ \hat{x}_1;\ \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \right\rangle \left\langle\ \hat{x}_2;\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right\rangle + a \left\langle\ \hat{x}_3;\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \right\rangle \\ &= \left\langle\ \hat{x}_1 \hat{x}_2;\ \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \hat{x}_2 + \hat{x}_1 \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right\rangle + \left\langle\ a\hat{x}_3;\ a \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \right\rangle \\ &= \left\langle\ \hat{x}_1 \hat{x}_2;\ \begin{bmatrix} \hat{x}_2 & \hat{x}_1 & 0 \end{bmatrix} \right\rangle + \left\langle\ a\hat{x}_3;\ \begin{bmatrix} 0 & 0 & a \end{bmatrix} \right\rangle \\ &= \left\langle\ \hat{x}_1 \hat{x}_2 + a\hat{x}_3;\ \begin{bmatrix} \hat{x}_2 & \hat{x}_1 & a \end{bmatrix} \right\rangle \\ &= \left\langle\ \hat{f};\ \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{bmatrix} \right\rangle. \end{aligned} \tag{11}$$

The *Surreal* data type can be implemented with either a static or dynamic sized derivative array. The appropriate version depends on whether the number of derivatives is known at compile time or at execution time. For example, the function in Eq. 8 always has three partial derivatives, which is known at compile time. The abbreviated class *SurrealS*, shown in Fig. 8, takes advantage of this by using the size of the derivative array as a template argument. The template argument allows the class to be customized for a variety of functions with different numbers of partial derivatives, and at the same time it allows the compiler to perform more aggressive optimizations, such as loop unrolling and vectorization, due to the static sized array. However, the number of partial derivatives is not always known at compile time. The *Surreal* class could, for example, be used to compute the derivative of a spline with respect to the knots. In this situation, the number of knots is not known at compile time. The abbreviated class *SurrealD*, shown in Fig. 9, dynamically allocates the derivative array as part of the constructor, and releases the memory in the destructor. This version of the *Surreal* class provides the flexibility to compute any number of partial derivatives, but reduces the amount of optimization that can be performed by the compiler. The static sized class can be used to compute a run time known number of partial derivatives that is larger than the static sized array through repeated calls to the function. This technique benefits from the compiler optimization, but requires repeated calculations of the function. The exact trade off of which of the two techniques for computing a run time known number of derivatives is not explored.

Two techniques for implementing C++ operator overloading to compute derivatives are presented here, a traditional implementation and an implementation based on expression templates [14]. Equation 8 is depicted as a binary expression tree, as show in Fig. 10, to facilitate the discussion of the two different operator overloading strategies. The leaves at the top of the tree represent the variables in the expression, and the nodes represent the arithmetic operators used to reduce the expression to a single value. Example source code is given for the *SurrealS* class for brevity, though the two operator overloading techniques are applied to both the *SurrealS* and *SurrealD* classes.

```
1   template<int N>
2   class SurrealS
3   {
4   public:
5
6   ...
7
8   protected:
9     double v_;        // value
10    double d_[N];     // derivative array
11  };
```

Figure 8: Abbreviated *Surreal* Class with Static Sized Derivative Array

```
1   class SurrealD
2   {
3   public:
4     // allocate the derivative array in constructor
5     explicit SurrealD( const int n ) : n_(n), d_(new double[n]) {}
6
7     // release the memory
8     ~SurrealD() { delete [] d_; }
9
10  ...
11
12  protected:
13    int n_;           // size of the derivative array
14    double v_;        // value
15    double *d_;       // derivative array
16  };
```

Figure 9: Abbreviated *Surreal* Class with Dynamic Sized Derivative Array

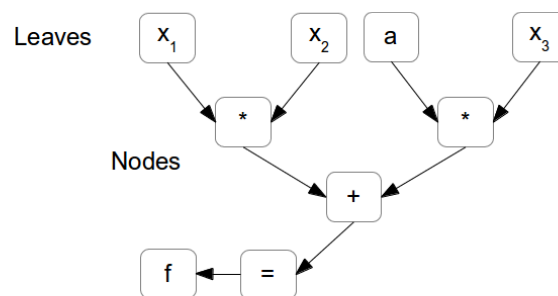


Figure 10: Binary Expression Tree for the Function $f(x_1, x_2, x_3) = x_1x_2 + ax_3$

### A. Traditional C++ Operator Overloading

Traditional operator overloading follows a greedy algorithm. That is, the results of binary operations are stored in temporary variables at the nodes of the expression tree. Hence, the data type that results from each nodal calculation is always a *SurrealS* type as depicted in Fig. 11.

The *SurrealS* class is shown in Fig. 12. The methods `value` and `deriv` respectively return the value and derivatives stored in the class, and the size of the derivative vector is given by the template argument $N$. This assumes that the number of derivatives is known at compile time. The two assignment operators are simple copy operations.

The implementation of the two multiplication operators in the expression tree is shown in Fig. 13. Note that the multiplication of two *SurrealS* data types is a simple application of the Chain rule. Both functions create a temporary variable, $c$, that stores the result of the calculation. Similarly, the addition operator is given in Fig. 14 where the result of the calculation is stored in the temporary variable, $c$.

The expression tree in Fig. 11 is parsed by first computing two temporary *SurrealS* variables from the two multiplication operators. These temporary variables are the arguments for the addition operator, where a third temporary variable is computed. This final temporary variable is the argument for the assignment operator in the *SurrealS* class. The pseudo-code for parsing the expression tree shown in Fig. 10 with traditional operator overloading is shown in Algorithm 1. Each binary operator creates temporary variables and induces a separate for-loop over the derivative computations. The use of these temporary variables to evaluate the expression tree can incur a significant computational overhead, particularly if the size of the temporary variables is large. The use of expression templates serves to eliminate these temporary variables from the expression tree.

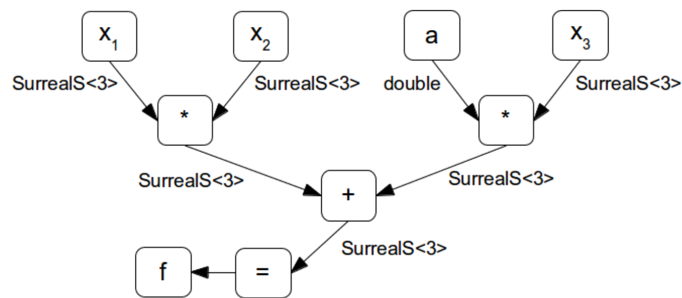


Figure 11: Traditional Operator Overloading Data Types from Eq. 8

```
1  template<int N>
2  class SurrealS
3  {
4  public:
5    // Default constructor intentionally omitted. This means Surreal is
6    // not initialized when declared (consistent with regular numbers).
7    // This also improves performance.
8
9    // value accessor operators
10         double& value()       { return v_; }
11   const double& value() const { return v_; }
12
13   // derivative accessor operators
14         double& deriv( const int i )       { return d_[i]; }
15   const double& deriv( const int i ) const { return d_[i]; }
16
17   // scalar assignment operator
18   SurrealS& operator=( const double& r )
19   {
20     for(int i = 0; i < N; i++)
21       d_[i] = 0;
22
23     v_ = r;
24   }
25
26   // surreal assignment operator
27   SurrealS& operator=( const SurrealS& r )
28   {
29     // copy all derivatives
30     for (int i = 0; i < N; ++i)
31       d_[i] = r.deriv(i);
32
33     // copy the value
34     v_ = r.value();
35
36     return *this;
37   }
38
39 protected:
40   double v_;      // value
41   double d_[N];   // derivative array
42 };
```

Figure 12: *SurrealS* Class with Traditional Operator Overloading

```
1  template<int N>
2  inline SurrealS<N> operator*( const SurrealS<N>& a, const SurrealS<N>& b )
3  {
4    // create temporary variable
5    SurrealS<N> c;
6
7    // compute all derivatives using the chain rule
8    for (int i = 0; i < N; i++)
9      c.deriv(i) = a.value()*b.deriv(i) + a.deriv(i)*b.value();
10
11   // compute the value
12   c.value() = a.value()*b.value();
13
14   return c;
15 }
16
17 template<int N>
18 inline SurrealS<N> operator*( const double& a, const SurrealS<N>& b )
19 {
20   // create temporary variable
21   SurrealS<N> c;
22
23   // compute all derivatives
24   for (int i = 0; i < N; i++)
25     c.deriv(i) = a*b.deriv(i);
26
27   // compute the value
28   c.value() = a*b.value();
29
30   return c;
31 }
```

Figure 13: *SurrealS* Class Multiplication using Traditional Operator Overloading

```
1  template<int N>
2  inline SurrealS<N> operator+( const SurrealS<N>& a, const SurrealS<N>& b )
3  {
4    // create temporary variable
5    SurrealS<N> c;
6
7    // compute all derivatives
8    for (int i = 0; i < N; i++)
9      c.deriv(i) = a.deriv(i) + b.deriv(i);
10
11   // compute the value
12   c.value() = a.value() + b.value();
13
14   return c;
15 }
```

Figure 14: *SurrealS* Class Addition using Traditional Operator Overloading

**Algorithm 1** pseudo-code for Traditional Operator Overloading

```
for i to N
  t1.deriv[i] = x1.value*x2.deriv[i] + x1.deriv[i]*x2.value
t1.value = x1.value*x2.value

for i to N
  t2.deriv[i] = a*x3.deriv[i]
t2.value = a*x3.value

for i to N
  t3.deriv[i] = t1.deriv[i] + t2.deriv[i]
t3.value = t1.value + t2.value

for i to N
  f.deriv[i] = t3.deriv[i]
f.value = t3.value
```

**Algorithm 2** pseudo-code for Lazy Template Expression Operator Overloading

```
for i to N
  f.deriv[i] = x1.value*x2.deriv[i] + x1.deriv[i]*x2.value + a*x3.deriv[i]
f.value = x1.value*x2.value + a*x3.value
```

### B. Expression Templates Operator Overloading

Expression templates is a lazy evaluation algorithm where all binary operators are evaluated together at the point when the assignment operator is parsed. The resulting code eliminates all temporary variables and contains a single for-loop over the derivative computations as shown in Algorithm 2. The elimination of temporary variables and coalescence of for-loops are the key differences between expression templates and traditional operator overloading. These code constructs reduce both computational cost and memory storage.

Expression templates are implemented by creating overloaded binary operators that instantiate a class that represents the binary operation, rather than directly evaluating the binary operation as is done with traditional operator overloading. The overloaded expression template multiplication operators and the classes that represent the multiplication operation, *SurrealSOpMul*, is shown in Fig. 15. A specialization of *SurrealSOpMul*, shown in Fig. 16, is used to represent the multiplication between a *SurrealS* and a scalar quantity. The constructor of the class *SurrealSOpMul* only stores references to the two expressions that are multiplied, and it does not compute the multiplication. Similarly, the addition operation shown in Fig. 17 creates an instance of the class *SurrealSOpAdd* that stores references to the two expression that are summed. The evaluation of the expression tree occurs in the expression assignment operator of the *SurrealS* class shown in Fig. 18. The *SurrealS* class shown in Fig. 18 uses a template argument to static size the array of derivatives.

Note that all of expression template classes inherit from the base class *SurrealSType*, shown in Fig. 19, and that the arguments to the overloaded operators are *SurrealSType*. This is a technique to ensure that the compiler is only able to instantiate the overloaded operators for data types that inherit from the *SurrealSType* class. Using the classes *SurrealS*, *SurrealSOpMul*, and *SurrealSOpAdd* as arguments is not feasible as different custom data types are generated dependent on the complexity of expression. For example, consider the expression tree in Fig. 10 with three partial derivatives. The overloaded multiplication operator is used twice to create instances of *SurrealSOpMul,* shown in Fig. 20, to represent the two different multiplication operations. These instances of *SurrealSOpMul* are the arguments to the overloaded addition operator which creates an instance of the class *SurrealSOpAdd* shown in Fig. 20. The custom *SurrealSOpAdd* instance is the argument for the assignment operator in the *SurrealS* class.

Unlike the traditional operator overloading where copies of the *SurrealS* class are created at each node of the expression tree, the for-loop of the assignment operator repeatedly calls the `deriv` method of the expression tree, which is the `deriv` method of the *SurrealSOpAdd* class. The `deriv` method of the *SurrealSOpAdd* class in turn calls the `deriv` method of the two *SurrealSOpMul* instances, which return the value of the derivative according to the chain rule. Hence, all the derivatives are computed with the single for-loop in the assignment operator. After all derivatives are computed, a similar call graph is parsed to compute the value. *Note that the value must be computed last in case the variable on the left of the assignment operator is also used in the expression tree*. This occurs, for example, when computing a sum. The expression tree is transformed into the pseudo-code in Algorithm 2 after the

compiler optimization inlines all the `deriv` and `value` methods. An example of using the *SurrealS* class to evaluate Eq. 8 and its partial derivatives in Eq. 9 is shown in Fig. 21.

The template expressions used here are only suitable for binary operations that can be efficiently elevated element-wise. A recursive function expression strategy is suitable for binary operations which cannot be efficiently evaluated element-wise, such as linear algebra matrix multiplication operations[24,25].

```
1  template<class ExprL, class ExprR>
2  class SurrealSOpMul :
3    public SurrealSType< SurrealSOpMul<ExprL, ExprR> >
4  {
5  public:
6    SurrealSOpMul(const ExprL& a, const ExprR& b) :
7      a_(a), b_(b), a_val_(a.value()), b_val_(b.value()) {}
8
9    // compute the value
10   double value() const { return a_val_*b_val_; }
11   double deriv(const int& i) const
12   {
13     // compute the derivative with the chain rule
14     return a_val_*b_.deriv(i) + a_.deriv(i)*b_val_;
15   }
16
17 protected:
18   const ExprL& a_;
19   const ExprR& b_;
20   const double a_val_, b_val_;
21 };
22
23 template<class ExprL, class ExprR>
24 inline SurrealSOpMul<ExprL, ExprR>
25 operator*(const SurrealSType<ExprL>& a, const SurrealSType<ExprR>& b)
26 {
27   // create an instance to represent the multiplication
28   return SurrealSOpMul<ExprL, ExprR>( a.cast(), b.cast() );
29 }
```

Figure 15: *SurrealS* Class Multiplication using Template Expression Operator Overloading

```
1  template<class Expr>
2  class SurrealSOpMul<Expr, double> :
3    public SurrealSType< SurrealSOpMul<Expr, double> >
4  {
5  public:
6    SurrealSOpMul(const Expr& e, const double s) : e_(e), s_(s) {}
7
8    // compute the value
9    double value() const { return e_.value()*s_; }
10
11   // compute the derivative
12   double deriv(const int& i) const { return e_.deriv(i)*s_; }
13
14 protected:
15   const Expr& e_;
16   const double s_;
17 };
18
19 template<class Expr>
20 inline SurrealSOpMul<Expr, double>
21 operator*(const double& s, const SurrealSType<Expr>& e)
22 {
23   // create an instance to represent the multiplication
24   return SurrealSOpMul<Expr, double>( e.cast(), s );
25 }
```

Figure 16: *SurrealS* Class Multiplication with a Scalar using Template Expression Operator Overloading

```
1  template<class ExprL, class ExprR>
2  class SurrealSOpAdd : public SurrealSType< SurrealSOpAdd<ExprL,ExprR> >
3  {
4  public:
5    SurrealSOpAdd(const ExprL& a, const ExprR& b) : a_(a), b_(b) {}
6
7    // compute the value
8    double value() const { return a_.value() + b_.value(); }
9
10   // compute the derivative
11   double deriv(const int& i) const { return a_.deriv(i) + b_.deriv(i); }
12
13 private:
14   const ExprL& a_;
15   const ExprR& b_;
16 };
17
18 template<class ExprL, class ExprR>
19 inline SurrealSOpAdd<ExprL,ExprR>
20 operator+(const SurrealSType<ExprL>& a, const SurrealSType<ExprR>& b)
21 {
22   // create an instance to represent the addition
23   return SurrealSOpAdd<ExprL,ExprR>( a.cast(), b.cast() );
24 }
```

Figure 17: *SurrealS* Class Addition using Expression Template Operator Overloading

```
1  template<int N>
2  class SurrealS : public SurrealSType< SurrealS<N> >
3  {
4  public:
5    // Default constructor intentionally omitted. This means Surreal is
6    // not initialized when declared (consistent with regular numbers).
7    // This also improves performance.
8
9    // value accessor operators
10         double& value()       { return v_; }
11   const double& value() const { return v_; }
12
13   // derivative accessor operators
14         double& deriv( const int i )       { return d_[i]; }
15   const double& deriv( const int i ) const { return d_[i]; }
16
17   // scalar assignment operator
18   SurrealS& operator=( const double& r )
19   {
20     for(int i = 0; i < N; i++)
21       d_[i] = 0;
22
23     v_ = r;
24   }
25
26   // expression assignment operator
27   template<class Expr>
28   SurrealS& operator=( const SurrealSType<Expr>& r )
29   {
30     const Expr& ExpressionTree = r.cast();
31
32     //Compute all derivatives
33     for (int i = 0; i < N; ++i)
34       d_[i] = ExpressionTree.deriv(i);
35
36     //Compute value last in case "this" is in the expression tree
37     v_ = ExpressionTree.value();
38
39     return *this;
40   }
41 protected:
42   double v_;      // value
43   double d_[N];   // derivative array
44 };
```

Figure 18: *SurrealS* Class using Expression Template Operator Overloading

```
template< class Derived >
struct SurrealSType
{
  // a convenient method for casting to the derived type
  const Derived& cast() const
  {
    return static_cast<const Derived&>(*this);
  }
};
```

Figure 19: *SurrealS* Base Class for Lazy Template Expression Operator Overloading

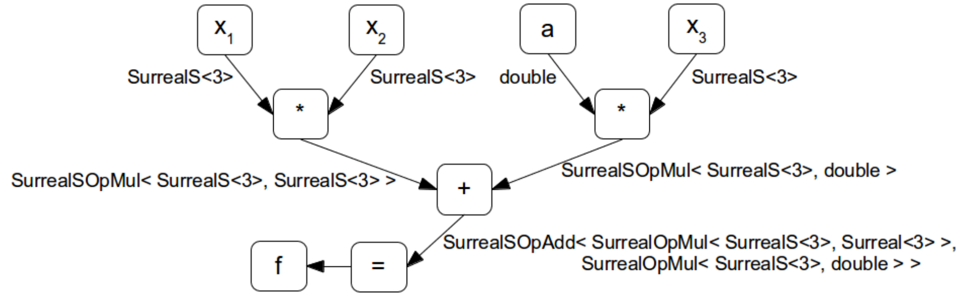


Figure 20: Template Expression Data Types Generated from Eq. 8

```
 1 #include "SurrealSType.h"
 2 #include "SurrealS_LAZY.h"
 3 #include "SurrealS_LAZY_Mul.h"
 4 #include "SurrealS_LAZY_MulScalar.h"
 5 #include "SurrealS_LAZY_Add.h"
 6 #include <iostream>
 7
 8 int main()
 9 {
10   double a = 4;
11   SurrealS<3> f, x1, x2, x3;
12
13   x1 = 3; x1.deriv(0) = 1;
14   x2 = 2; x2.deriv(1) = 1;
15   x3 = 5; x3.deriv(2) = 1;
16
17   f = x1*x2 + a*x3;
18
19   std::cout << "f      = " << f.value()  << std::endl;
20   std::cout << "df/dx1 = " << f.deriv(0) << std::endl;
21   std::cout << "df/dx2 = " << f.deriv(1) << std::endl;
22   std::cout << "df/dx3 = " << f.deriv(2) << std::endl;
23 }
24
25 /* OUTPUT:
26  * f      = 26
27  * df/dx1 = 2
28  * df/dx2 = 3
29  * df/dx3 = 4
30  */
```

Figure 21: Example Computing Eq. 8 and the Partial Derivatives in Eq 9 using Expression Template Operator Overloading

### C. Automatic Differentiation Practice and Timing

Automatic differentiation has been criticized for incurring an unacceptable computational overhead. Execution times, obtained with multiple compilers, to compute the derivatives of the two-dimensional van Leer flux and Roe flux using static and dynamic array size *Surreal* types are compared with manually differentiated code written both in C and Fortran in Tables 2 and 3. Tables 2 and 3 also show the execution time using traditional and expression templates operator overloading as well as the ratio of the *Surreal* execution time relative to the two manually coded implementations. The *Surreal* datatype uses a size four array to compute the derivatives of the van Leer flux, and a size eight array to compute the derivatives of the Roe flux. All the calculations use double precisions arithmetic, and the codes are compiled with the -O3 optimization flag.

The expression template operator overloading is approximately an order of magnitude faster than the traditional operator overloading for the *Surreal* datatype with a dynamic sized array. The difference between the *Surreal* datatype with static and dynamic sized array are more significant for the Roe flux calculation. Here, the *Surreal* datatype with expression templates is approximately one order of magnitude slower than the static sized array datatype. However, the expression template implementation is approximately one order of magnitude faster than the traditional operator overloading. Most notably, the *Surreal* datatype with static sized array has an execution time that is comparable to the manually coded differentiated code using both traditional and expression template overloaded operators. Hence, the static sized *Surreal* datatype is able to compute derivatives of fluxes with execution times comparable to manually coded derivatives, but only requires developers to implement the primary function. Any changes to the primary function usually requires complex changes in the manually differentiated code, which is clearly unnecessary using automatic differentiation. This leads to a significant reduction in the amount of software code that needs to be verified and maintained.

| | Manually Coded | | Automatic | | Ratio | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | C (s) | Fortran (s) | Static (s) | Dynamic (s) | Static/C | Dynamic/C | Static/F | Dynamic/F | |
| GNU 4.8 | 8.35 | 10.55 | 16.89 | 295.59 | 2.02 | 35.38 | 1.60 | 28.02 | Traditional |
| | | | 15.04 | 42.62 | 1.80 | 5.11 | 1.43 | 4.04 | Lazy |
| Clang 3.4 | 11.61 | - | 10.98 | 308.60 | 0.95 | 26.58 | - | - | Traditional |
| | | | 11.72 | 39.57 | 1.02 | 3.43 | - | - | Lazy |
| Intel 14 | 10.72 | 10.46 | 6.73 | 304.93 | 0.63 | 28.45 | 0.64 | 29.17 | Traditional |
| | | | 7.65 | 36.86 | 0.71 | 3.44 | 0.73 | 3.52 | Lazy |

Table 2: Execution Time in Seconds to Compute $100 \times 10^6$ van Leer Flux Evaluations; Averaged over 50 Calls

| | Manually Coded | | Automatic | | Ratio | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | C (s) | Fortran (s) | Static (s) | Dynamic (s) | Static/C | Dynamic/C | Static/F | Dynamic/F | |
| GNU 4.8 | 5.18 | 5.15 | 7.70 | 270.58 | 1.49 | 52.21 | 1.50 | 52.54 | Traditional |
| | | | 5.99 | 49.81 | 1.16 | 9.61 | 1.16 | 9.66 | Lazy |
| Clang 3.4 | 5.10 | - | 5.85 | 292.60 | 1.15 | 57.36 | - | - | Traditional |
| | | | 7.83 | 55.08 | 1.53 | 10.80 | - | - | Lazy |
| Intel 14 | 4.45 | 5.18 | 7.10 | 381.40 | 1.60 | 85.74 | 1.37 | 73.65 | Traditional |
| | | | 6.06 | 56.91 | 1.36 | 12.78 | 1.17 | 11.00 | Lazy |

Table 3: Execution Time in Seconds to Compute $25 \times 10^6$ Roe Flux Evaluations; Averaged over 50 Calls

## V. Conclusion

The verification driven development process presented is designed to accelerate the development of a new software framework for a high-order output-based mesh adaptation method. The process is based on the principles of code Verification and Validation. The software development environment is designed to encourage developers to utilize unit testing as the primary means of correctness verification, and simplifies the process of creating and executing the unit tests as much as possible. Having a comprehensive suite of tests means that developers can focus on developing new features and answering research questions rather than spending extraordinary amounts of time debugging software code. Code coverage utilities are employed to ensure that the verification test suite remains comprehensive, and new features are not added to the software without also developing corresponding unit tests. The use of regression testing is restricted to cases where analytical solutions to partial differential equations exist, and the spectrum of analytical solutions is expanded through the use of the Method of Manufactured Solutions. The implementation of the manufactured solutions is simplified through the use of the strong form of the partial differential equation rather than symbolic manipulation software. Code quality is enhanced by compiling the source code with multiple compilers with all warning flags enabled as part of the verification process, and by augmenting the compiler with a static code analyzer which is automatically executed with the build process. Collective ownership and collaboration is promoted amongst developers through the use of automatic coding standards verification that is executed as part of the build and a flexible continuous integration procedure. The continuous integration procedure is designed to promote frequent commits, promote collaboration, and retain a copy of the source code that passes all verification tests. In particular, promoting collaboration amongst developers further accelerates the development process. Finally, the use of automatic differentiation significantly simplifies the process of developing CFD software as it eliminates the need to derive, implement, and verify any linearizations. Automatic differentiation with the use of operator overloading is demonstrated to exhibit the same efficiency as manually coded linearizations when static array sizes are employed, and the use of the expression templates improves the computational efficiency by an order of magnitude for dynamic array sizes.

## Acknowledgments

The authors would like to thank Dr. Michael List and Robert Haimes for their insights that lead to the presented software development environment. This work was supported by funding from: The Boeing Company with technical monitor Dr. Mori Mani; NASA with technical monitor Dr. Harold Atkins (NASA Cooperative Agreement NNX12AJ75A); and Aramco Services Company with technical monitor Dr. Ali Dogru.

## References

[1]Darmofal, D. L., Allmaras, S. R., Yano, M., and Kudo, J., "An Adaptive, Higher-order Discontinuous Galerkin Finite Element Method for Aerodynamics," AIAA CFD Conference, June 2013.

[2]Cockburn, B., Gopalakrishnan, J., and Lazarov, R., "Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems," Vol. 47, No. 2, 2009, pp. 1319–1365.

[3]Nguyen, N., Peraire, J., and Cockburn, B., "A Hybridizable Discontinuous Galerkin Method for the Incompressible Navier-Stokes Equations," AIAA 2010-362, 2010.

[4]Peraire, J., Nguyen, N., and Cockburn, B., "A Hybridizable Discontinuous Galerkin Method for the Compressible Euler and Navier-Stokes Equations," AIAA 2010-363, 2010.

[5]Peraire, J., Nguyen, N., and Cockburn, B., "An embedded discontinuous Galerkin method for the compressible Euler and Navier-Stokes equations," AIAA 2011–3228, 2011.

[6]Shakib, F., Hughes, T. J. R., and Johan, Z., "A new finite element formulation for computational fluid dynamics: X. The compressible Euler and Navier-Stokes equations," *Comput. Methods Appl. Mesh. Eng.*, Vol. 89, 1991, pp. 141–219.

[7]Hughes, T. J. R., Franca, L. P., and Hulbert, G. M., "A new finite element formulation for computational fluid dynamics: VIII. The Galerkin/least-squares method for advective-diffusive equations," Vol. 73, 1989, pp. 173–189.

[8]AIAA (1998), *Guide for Verication and Validation of Computational Fuild Dynamics Simulations*, AIAA-G-077-1998e, Reston, VA, 1998.

[9]Oberkampf, W. L. and Roy, C. J., *Verifcation and Validation in Schientific Computing*, Cambridge University Press, New York, 2010.

[10]Kleb, W. L., Nielsen, E. J., Gnoffo, P. A., Park, M. A., and Wood, W. A., "Collaborative Software Development in Support of Fast Adaptive AeroSpace Tools (FAAST)," AIAA 2003-3978, 2003.

[11]Newman, J. C., Anderson, W. K., and Whitfield, D. L., "Multidisciplinary Sensitivity Derivatives using Complex Variables," Engineering Research Center Report MSSU-COE-ERC-98-08, Mississippi State University, 1998.

[12]Anderson, W. K., Newman, J. C., Whitfield, D. L., and Nielsen, E. J., "Sensitivity Analysis for the Navier-Stokes Equations on Unstructured Meshes using Complex Variables," AIAA 1999-3294, 1999.

[13]Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The Complex-Step Derivative Approximation," *ACM Transactions on Mathematical Software*, Vol. 29, No. 3, Sept. 2003, pp. 245–262.

[14]Veldhuizen, T. L., "Scientific Computing: C++ Versus Fortran: C++ has more than caught up," *Dr. Dobb's Journal of Software Tools*, Vol. 22, No. 11, 1997, pp. 34–91.

[15]Abrahams, D., *C++ template metaprogramming : concepts, tools, and techniques from Boost and beyond*, Pearson Education, Inc., 2004.

[16]Davis, T. A., "Algorithm 832: UMFPACK V4.3 - An unsymmetric-pattern multifrontal method," *ACM Transactions on mathematical software*, Vol. 30, No. 2, 2004, pp. 196–199.

[17]Ghia, U., Bayyuk, S., Habchi, S., Roy, C., Shih, T., Conlisk, T., Hirsch, C., and Powers, J. M., "The AIAA Code Verification Project - Test cases for CFD Code Verification," AIAA Paper AIAA 2010-125, 2010.

[18]Wang, Z., Fidkowski, K., Abgrall, R., Bassi, F., Caraeni, D., Cary, A., Deconinck, H., Hartmann, R., Hillewaert, K., Huynh, H., Kroll, N., May, G., Persson, P.-O., van Leer, B., and Visbal, M., "High-order CFD methods: current status and perspective," *International Journal for Numerical Methods in Fluids*, Vol. 72, No. 8, 2013, pp. 811–845.

[19]Roache, P. J. and Steinberg, S., "Symbolic Manipulation and Computational Fluid Dynamics," *AIAA Journal*, Vol. 22, No. 10, 1984, pp. 1390–1394.

[20]Oberkampf, W. L. and Aeschliman, D. P., "Methodology for Computational Fluid Dynamics Code Verfication/Validation," AIAA Paper AIAA 1995-2226, 1995.

[21]Roache, P. J. and Steinberg, S., "Code Vertification by the Method of Manufactured Solutions," *Journal of Fluids Engineering*, Vol. 124, No. 1, 2002, pp. 4–10.

[22]Brooks, A. N. and Hughes, T. J. R., "Streamline upwind / Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations," Vol. 32, 1982, pp. 199–259.

[23]Phipps, E. and Pawlowski, R., "Efficient Expression Templates for Operator Overloading-based Automatic Differentiation," *ArXiv e-prints*, May 2012.

[24]Myrnyy, V., "Recursive Function Templates as a Solution of Linear Algebra Expressions in C++," *CoRR*, Vol. cs.MS/0302026, 2003.

[25]Galbraith, M. C., *A Discontinuous Galerkin Chimera Overset Solver*, PhD thesis, University of Cincinnati, School of Aerospace Systems, Dec. 2013.