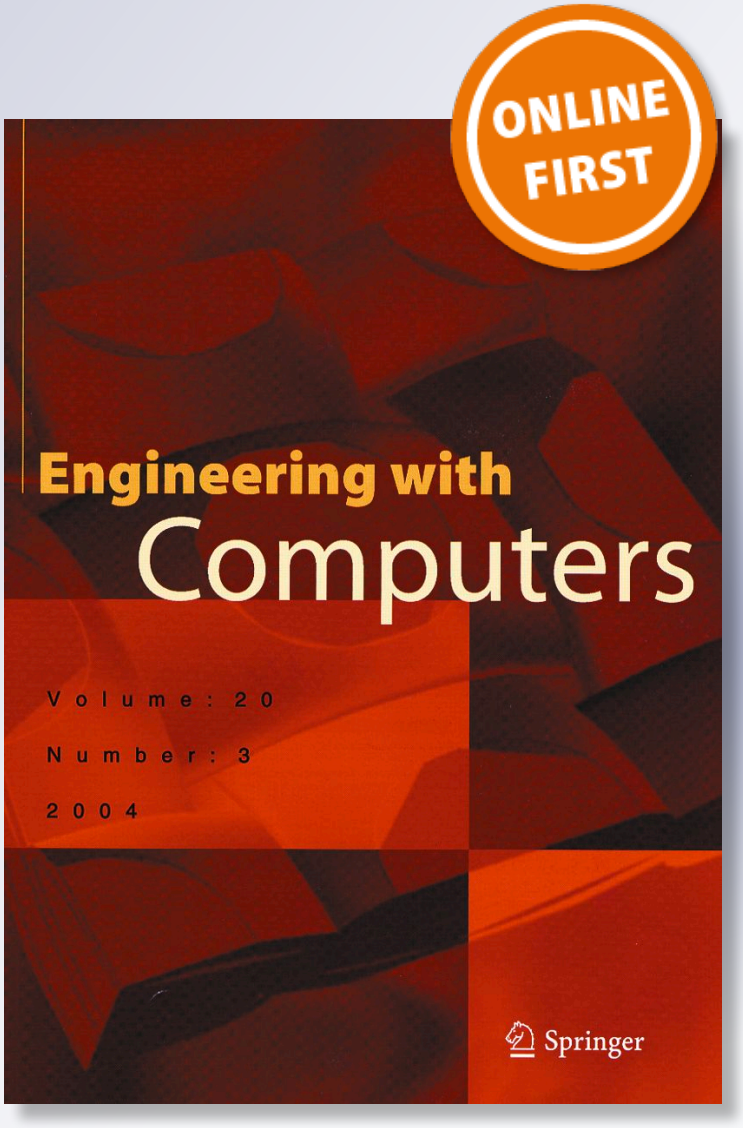
# MASA : *a library for verification using manufactured and analytical solutions*

**Nicholas Malaya, Kemelli C. Estacio-Hiroms, Roy H. Stogner, Karl W. Schulz, Paul T. Bauman & Graham F. Carey**

**Engineering with Computers**
An International Journal for Simulation-Based Engineering

ISSN 0177-0667

Engineering with Computers
DOI 10.1007/s00366-012-0267-9




Springer

Engineering with Computers
DOI 10.1007/s00366-012-0267-9

ORIGINAL ARTICLE

# MASA: a library for verification using manufactured and analytical solutions

**Nicholas Malaya · Kemelli C. Estacio-Hiroms · Roy H. Stogner · Karl W. Schulz · Paul T. Bauman · Graham F. Carey**

Received: 5 December 2011 / Accepted: 17 May 2012


**Abstract** In this paper we introduce the Manufactured Analytical Solution Abstraction (MASA) library for applying the method of manufactured solutions to the verification of software used for solving a large class of problems stemming from numerical methods in mathematical physics including nonlinear equations, systems of algebraic equations, and ordinary and partial differential equations. We discuss the process of scientific software verification, manufactured solution generation using symbolic manipulation with computer algebra systems such as Maple™ or SymPy, and automatic differentiation for forcing function evaluation. We discuss a hierarchic methodology that can be used to alleviate the combinatorial complexity in generating symbolic manufactured solutions for systems of equations based on complex physics. Finally, we detail the essential features and examples of the Application Programming Interface behind MASA, an open source library designed to act as a central repository for manufactured and analytical solutions over a diverse range of problems.

**Keywords** Verification · Manufactured solutions · Partial differential equations · Finite elements

## 1 Introduction

Over the last decade, the notions of verification and validation (V&V) have been recognized as crucial research areas in computational science [13, 17, 19] in order to enable the *predictability* of computer simulations. Verification refers to ensuring the correct solution of equations used in the mathematical model of interest, while validation refers to ensuring the correctness of the mathematical model in representing observed physical phenomena [3, 18].

Verification efforts fall into two broad categories: *code verification* and *solution verification*. Roache [25] differentiates between the two by noting that solution verification requires error *estimation* while code verification uses error *evaluation*. Code verification focuses on identifying failures of the code to correctly implement a desired numerical algorithm. Solution verification is the process of quantifying the numerical errors (e.g., round-off, iterative, and discretization errors) that can cause the numerical solution to be an inadequate approximation of the correct solution. When performing code verification, analytical solutions to mathematical equations are used to calculate *error* in a corresponding approximate solution. This is in contrast to solution verification where one simulates the phenomenon of interest and has no *a priori* knowledge of the solution; in such cases error can only be *estimated*.

There are many techniques common in the software engineering community [12, 31] that can assist code

The authors wish to dedicate this paper to the memory of Professor Graham Carey, who passed away during preparation of this work. Professor Carey was a mentor and advisor to the authors of this paper and will be greatly missed.

N. Malaya (✉) · K. C. Estacio-Hiroms · R. H. Stogner · K. W. Schulz · P. T. Bauman · G. F. Carey
Predictive Engineering and Computational Sciences Center, Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712, USA
e-mail: nick@ices.utexas.edu

K. C. Estacio-Hiroms
e-mail: kemelli@ices.utexas.edu

R. H. Stogner
e-mail: roystgnr@ices.utexas.edu

K. W. Schulz
e-mail: karl@ices.utexas.edu

P. T. Bauman
e-mail: pbauman@ices.utexas.edu

Published online: 07 June 2012

verification; for example, in the design and construction of unit and module tests that exercise-specific subsets of the software, regression tests that exercise fixes for previously discovered software errors, code coverage analysis, etc. In the realm of mathematical modeling, one should also exercise the software to ensure that exact solutions to the desired equations can be recovered when the code is given the corresponding inputs. This process is termed the "method of exact solutions" [26]. However, one is often limited by the availability of analytical solutions; there will almost certainly be no non-trivial analytical solutions available when the physics and/or geometry become complex.

An important tool that has emerged over the past decade to assist in the code verification process is the method of manufactured solutions (MMS) [13, 14, 16, 24, 33]. MMS, instead of relying upon the availability of an exact solution to the governing equations, specifies a solution. This artificial solution is then substituted into the equations. Naturally, there will be a residual term since the chosen function is unlikely to be an exact solution to the equations. This residual can then be added to the code as a source term; the MMS test then uses the code to solve the modified equations and checks that the chosen function is recovered. Although previous work has focused mainly on partial differential equations (PDE's), this idea applies to a broad range of systems in mathematical physics including nonlinear equations, systems of algebraic equations, and ordinary differential equations.

MMS has been applied in various applications [4–7, 15, 21, 28, 29, 32, 35], but previous works described neither instances when human intervention was required when using symbolic manipulation software nor the software development effort required to enable MMS implementations in a codebase. Furthermore, the computational science community lacks a standard set of tools that serve as a common, reusable repository of manufactured solutions. We address both of these important issues in the current work.

The remainder of this paper is organized as follows. Section 2 describes the MMS more precisely and introduces notation. Section 3 describes our experience in generating source terms using several different methods including computer algebra systems and automatic differentiation to evaluate PDE residuals. Section 4 details a strategy, which we term "hierarchic MMS", to overcome the combinatorial explosion that occurs in symbolic manipulations. In Sect. 5 we describe the Manufactured Analytical Solution Abstraction (`MASA`) library which encapsulates the MMS software efforts to date within the Center for Predictive Engineering and Computational Sciences (PECOS).[1] Finally, we conclude in Sect. 6.

[1] http://pecos.ices.utexas.edu.

## 2 Method of manufactured solutions

As discussed in Sect. 1, the majority of models stemming from mathematical physics lack analytical solutions for scenarios of engineering interest. In particular, there is often a lack of analytical solutions of sufficient complexity for the verification of a numerical scheme used to approximate the relevant mathematical model. The MMS is one strategy used to test scientific software in the way one would use analytical solutions if they were available. We follow and extend the main ideas in [25, 30, 33], as summarized below.

Suppose we wish to solve the following problem:

$$\begin{array}{l}\text{Find } u \in U \text{ such that:} \\ F(u) = 0\end{array}, \tag{1}$$

where $F : U \rightarrow V$ is an operator representing the model of interest, $U$ and $V$ are appropriately defined vector spaces, and $u$ is the solution. The idea is to *manufacture* a solution $\hat{u} \in U$, then produce a simple modified operator $\hat{F}$ for which $\hat{u}$ is a solution. In general one cannot expect and will not require $F(\hat{u}) = 0$. Thus, the original operator will have a potentially non-zero residual term, which we will refer to as $f = -F(\hat{u})$. If we use this residual as an additional forcing function, a new source term in the equations, we obtain the modified operator

$$\hat{F}(v) \equiv F(v) + f \tag{2}$$

which gives us $\hat{F}(\hat{u}) = 0$ by construction. Then, the verification problem we wish to solve is:

$$\begin{array}{l}\text{Find } \hat{u} \in U \text{ such that:} \\ \hat{F}(\hat{u}) = 0\end{array} \tag{3}$$

which is of the same root-finding form as the original problem. Typical decompositions of $F$ in the original problem, such as time stepping for transient problems, extend to similar decompositions of $\hat{F}$ in the verification problem. Because of this similarity, modifying a numerics code to optionally solve $\hat{F}$ rather than $F$ is minimally intrusive: residual evaluations are changed only by a constant, Jacobians are unchanged, and most physics and solver code can remain unmodified.

For complex problems, such as discretizations of PDEs, we typically search for an approximate numerical solution of the true problem, so that the discrete problem becomes:

$$\begin{array}{l}\text{Find } \hat{u}^h \in U^h \text{ such that:} \\ F(\hat{u}^h) = f\end{array} \tag{4}$$

If we construct a manufactured solution $\hat{u} \in U^h$, and if the code's numerical formulation is one which guarantees an approximation error which is bounded by a constant multiple of best-case interpolation error, then we can

compare $\hat{u}$ and $\hat{u}^h$ pointwise and expect a near-exact match (limited only by rounding and iterative errors). If $\hat{u} \notin U^h$, then we must compare in appropriate norms and check for proper convergence rates. If one has *a priori* bounds of the form

$$\|\hat{u} - \hat{u}^h\| \leq Ch^p, \tag{5}$$

where $C$ is a constant and $p$ is the expected order of convergence in the evaluated norm with respect to the mesh spacing parameter $h$, then a verification test might solve the manufactured problem at various levels of uniform $h$-refinement in the asymptotic regime and ensure that the numerical convergence rate is near $p$. Similar tests can be constructed using manufactured solutions to verify convergence with adaptive refinement or convergence of postprocessed quantities of interest.

Implementing the concepts outlined above is straightforward, assuming one has a manufactured solution and corresponding source term. We discuss challenging aspects of the process of generating these solutions and sources terms in the next section.

The remainder of this paper focuses on strong solutions, where the manufactured solution is sufficiently differentiable. Weak solutions whose forcing functions are properly integrated with standard quadrature can also be supplied via our current design. Weak solutions with more complex forcing functions may require design extensions, or in some cases further research.

## 3 Generating manufactured solutions

The starting point of MMS-based verification is to manufacture a suitable closed-form exact solution to a related problem form. To accomplish this, one must first select a non-trivial analytical solution.

The form of the solution is somewhat arbitrary, but it should be selected under a few acceptance criteria [16, 25, 29, 30]. The manufactured solution function for a PDE should typically be chosen to be smooth and sufficiently differentiable, so one may exercise all the terms of the governing equations without imposing special coordinate systems or particular boundary conditions. Although one is not necessarily concerned with the realism of the chosen function, it should still be physically consistent; e.g., solutions should be avoided which have negative densities, pressures, temperatures, etc. Manufactured solutions are commonly generated from trigonometric equations or polynomials [13, 29] although other forms are certainly possible. A useful manufactured solution should be of sufficient algebraic complexity to appropriately exercise the solver code undergoing the process of verification. However, symbolically evaluating a residual using a slightly complex solution as input often results in a much more complex forcing function as output [11, 24]. This evaluation can be done manually, but to limit the likelihood of human error, the process of manufactured forcing function generation is best undertaken using computer algebra systems (CAS) for symbolic manipulation.

In principle, any CAS can be employed: from commercial packages such as Maple™ and Mathematica™, to open-source libraries such as SymPy, Maxima, and Sage. The developers chose to use Maple™ and SymPy for the algebraic manipulations involved in this work. Although it is often suggested that the generation of sources terms with CAS software is almost entirely automated, our experience has been that this is not the case, particularly for producing reasonably sized source terms. We describe experiences in the following sections using representative CAS software.

For many of the manufactured solutions in the present work, we use a modified form of the function given in [27] to incorporate temporal variation:

$$\begin{aligned}\hat{u}(x,y,z,t) = \hat{u}_0 &+ \hat{u}_x f_s\left(\frac{a_{\hat{u}x}\pi x}{L}\right) + \hat{u}_y f_s\left(\frac{a_{\hat{u}y}\pi y}{L}\right) \\ &+ \hat{u}_z f_s\left(\frac{a_{\hat{u}z}\pi z}{L}\right) + \hat{u}_t f_s\left(\frac{a_{\hat{u}t}\pi t}{L}\right),\end{aligned} \tag{6}$$

where $f_s(\cdot)$ denotes either sine or cosine functions and $\hat{u}_x, \hat{u}_y, \hat{u}_z, \hat{u}_t, a_{\hat{u}x}, a_{\hat{u}y}, a_{\hat{u}z}, a_{\hat{u}t}, L \in \mathbb{R}$. Although only a scalar function is shown in (6), this is easily extended to vector-valued functions using the functional form for each component with corresponding coefficients. Furthermore, we note that we've considered other forms within the library focusing on compressible turbulence [20].

### 3.1 Maple™

Although the ideas presented previously are simple, several significant issues arise during the generation of forcing function source terms when using Maple™: the extensive length of the resulting terms, the choice of symbolic manipulations to be carried-out, the risk of human error when performing the manipulations, and software limitations. Simply evaluating the manufactured solution in the mathematical operator generally results in an extensive expression for the source term. Consequently, the source term complexity grows dramatically so further manipulations, such as collecting and factorizing terms, are usually required in order to facilitate the transition to usable source code. Consider the transient, two-dimensional, Euler momentum equation in the $x$ direction:

$$\mathcal{L}(u) = \frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2 + p)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} = 0. \tag{7}$$

When the manufactured solution (6) is substituted into (7), the resulting source term has over 5,300 characters. After simplification, the source term size is reduced to "only" 1,000 characters.

However, determining just which algebraic manipulations should be performed in order to reduce the size of the source term is not trivial. Ideally, these operations will enhance readability, while minimizing the likelihood of introducing errors. In many cases, the reduction in size can be substantial. For example, for the three-dimensional Navier–Stokes energy equation with constant viscosity, the source term presents over 338,000 characters prior to factorization, and has been reduced to 4,800 characters after symbolic manipulations, corresponding to only 1.4 % of its original size. In order to verify that the simplified function is consistent, one can compare (subtract) from the unsimplified version. Surprisingly, this seemingly simple operation may require care in order to succeed. In the case of Maple™ 13, a new worksheet had to be created to perform the comparison.

Of course, the complexity of the source term increases with the complexity of the mathematical model. Unfortunately, our experience has shown that it is common for the current Maple™ software to be unable to generate source terms for complicated PDE's in a timely manner, if at all. Applying commands in order to simplify extensive expressions (such as the 338,000 character three-dimensional Navier–Stokes energy equation discussed above) is challenging even with a high performance workstation.[2] Indeed, incorporating the Sutherland viscosity model [34] into the three-dimensional Navier–Stokes equations and attempting to generate a compact source term using the manufactured solution (6) causes Maple™ 13 to fail.

While this outcome was undesirable, it was also not surprising. The source term created by simply substituting the manufactured solutions (6) for each one of the flow variables into the three-dimensional total energy Navier–Stokes equation with Sutherland viscosity effects comprises over 1,612,000 characters.

### 3.2 SymPy

An alternative open source computer algebra system was sought in an effort to alleviate the manipulation issues encountered with Maple™. SymPy was chosen as it is under active development and particularly simple to use.

In the developer's experience, SymPy tends to generate smaller source terms than those output by Maple™. For instance, the SymPy-generated source term for the three-dimensional Navier–Stokes energy equation with the Sutherland viscosity model has under 41,000 characters. On the other hand, Maple™ tends to be more robust and includes a wider variety of commands for algebraic manipulation, and, therefore, the potential for reducing the size of the source terms is greater.

Unfortunately, SymPy is still in early stages of development and, therefore, more susceptible to the presence of errors (encountered, for example, in simply collecting a common factor in an expression). Additionally, SymPy requires specific commands in order to avoid some undesirable numerical evaluations, such as approximating $\sqrt[4]{2}$ to a real number; that is accomplished either explicitly declaring the exponent as a rational (e.g., 2**Rational(1,4)) or using the function `sympify()`.

The issues regarding the complexity of the source terms and the choice of the manipulations to be carried out remain unchanged, however.

### 3.3 Automatic differentiation

Although the majority of the manufactured solutions currently exposed by `MASA` were generated via CAS, the `MASA` Application Programming Interface (API) is independent of solution generation method. The same standard interface can be used to expose any manufactured solution whose evaluation has been encapsulated in C, C$^{++}$, or Fortran.

One promising method for alleviating combinatorial issues, useful for users who wish to write their own manufactured solutions in C$^{++}$, is automatic differentiation (AD). With an AD class using operator overloading, the same code which calculates a function value using a built-in floating-point scalar can also be used to calculate function gradients, second derivatives, etc. using the AD class as its scalar. `MASA` includes examples of Euler and Navier–Stokes manufactured solutions which use this method to directly evaluate PDE residual terms.

Although more sophisticated AD tools are available elsewhere, to avoid third-party dependencies `MASA` includes a simple forward AD class, `DualNumber`, templated around the underlying scalar type and derivative type. This design was inspired by the hyper-dual numbers of Fike & Alonzo [8], and using template recursion to provide a dual of a dual of a floating point value gives an object which is algebraically isomorphic to their hyper-duals. However, the recursive capabilities here enable arbitrary order derivative calculations. Using a similarly templated array class for the derivative type also enables easy simultaneous multidimensional derivative calculations with `NumberArray` objects representing gradient vectors and Hessian matrices. A once-differentiable scalar-valued function type `ADOne` on $\mathbb{R}^3$ can then be defined as

[2] The workstation used in this work was a 6-core 3.3 GHz i7 Intel processor with 12 GB RAM running Ubuntu Linux.

DualNumber<double,NumberArray<3,double>> and a twice-differentiable function type ADTwo as DualNumber<ADOne,NumberArray<3,ADOne>>. A vector-valued twice-differentiable function is then a NumberArray<3,ADTwo>.

Manufactured solution components such as RHO and U are declared as such types, then instantiated as in (6) (or some other manufactured form) from properly initialized geometric variables x, y, and z of the scalar-valued type. Evaluating the manufactured forcing function components is then as simple as writing the corresponding PDE terms in C$^{++}$. For example, the steady Euler momentum equation residual:

$$\mathcal{L}(\boldsymbol{u}) = \nabla \cdot (\rho \boldsymbol{u}\boldsymbol{u}) + \nabla p \tag{8}$$

is evaluated in C$^{++}$ as:

```
divergence(RHO * U.outerproduct(U))
  + P.derivatives()
```

For many problems, such PDE terms are simple enough to be inspected by hand. Some common user errors which produce invalid code (e.g. attempting to take second derivatives of a once-differentiable data type) are also caught at compile time.

### 3.4 Verifying manufactured solutions

Unfortunately, no method of generating manufactured solutions can guarantee the generation of *correct* manufactured solutions. Verification tools are themselves in need of verification. No single automatic way to detect any mistake in a manufactured solution exists, but various practices can reduce the opportunities for error.

When using computational algebra systems, one important step is to subtract condensed forcing function expressions from the original residual evaluation, and let the system confirm that the result is zero. This finds any errors in the simplification process. Pointwise evaluation of CAS results against floating-point C code output is also useful to confirm that rounding error in the latter is insignificant. When called from C$^{++}$ user code, MASA can even be instructed to simultaneously evaluate manufactured solutions at multiple precision levels to perform runtime detection of excessive rounding error at the user-specified evaluation points.

It is possible to generate equivalent manufactured solutions in multiple ways, then compare them against each other. MASA currently does this with one example each of Euler and Navier–Stokes solutions, comparing automatic versus symbolic differentiation output and verifying a relative error within an order of magnitude of the underlying floating point precision.

Finally, any manufactured solution is implicitly both a verification tool and a verification target when it is applied to a real simulation code. Early users of MASA have found many bugs in their own codes, but one MASA-based verification failure resulted in a correction of the manufactured forcing function itself, and a corresponding bug report sent upstream to the CAS system developers. Fortunately, such an error in a manufactured solution is far more likely to cause such a false negative rather than a false positive result; a false positive would require not just errors in both simulation code and manufactured solution, but matching errors.

## 4 Hierarchic MMS

One possible mechanism to dramatically reduce the combinatorial explosion encountered in generating source terms is to simplify each equation by decomposing it into a combination of sub-terms. Then, each of the terms may be applied to the manufactured solution individually and the resulting expressions can be combined to compactly represent the aggregate source term for the original equation.

For instance, consider the full three-dimensional Navier–Stokes energy equation:

$$\mathcal{L} = \frac{\partial(\rho e_t)}{\partial t} + \nabla \cdot (\rho \mathbf{u} e_t) + \nabla \cdot \mathbf{q} + \nabla \cdot (p\mathbf{u}) - \nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u}). \tag{9}$$

The right hand side terms in Eq. (9) can be decomposed using five separate operators:

$$\begin{aligned} \mathcal{L}_1 &= \frac{\partial(\rho e_t)}{\partial t} \\ \mathcal{L}_2 &= \nabla \cdot (\rho \mathbf{u} e_t) \\ \mathcal{L}_3 &= \nabla \cdot \mathbf{q} \\ \mathcal{L}_4 &= \nabla \cdot (p\mathbf{u}) \\ \mathcal{L}_5 &= -\nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u}) \end{aligned} \tag{10}$$

where

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3 + \mathcal{L}_4 + \mathcal{L}_5.$$

Terms 2 through 5 can even be split into three sub-terms each, by taking each component of the divergence operator separately. A nice feature of the decomposition approach is that physical meaning can be associated with each term. In (10), $\mathcal{L}_1$ denotes the rate of accumulation of inertial and kinetic energy, $\mathcal{L}_2$ is the net rate of internal and kinetic energy increase through convection, $\mathcal{L}_3$ is the rate of heat addition due to heat conduction, $\mathcal{L}_4$ is the rate of work done on the fluid by pressure forces, and $\mathcal{L}_5$ is the rate of work done on the fluid by shear forces.

Using this strategy, the source term for the energy equation can be calculated in a more tractable way: each subterm in the equation has its corresponding contribution to the residual calculated independently, then simplified independently, then summed back and re-simplified to compute the final residual. This requires less human effort, decreases the computational requirements, reduces the incidence of software failure, and increases the flexibility in the code verification procedure.

When constructing multiple manufactured solutions in this way for certain classes of physics, augmentation of the physical models becomes much simpler. We refer to this method as "Hierarchic MMS" because of its natural suitability for use with hierarchic sets of physical models. For example, construction of manufactured solutions to the Navier–Stokes equations can begin with compact manufactured solution components from the Euler equations and merely include additional terms. If one wishes to incorporate a different viscosity model in the Navier–Stokes equations, only one component needs recalculation instead of the entire expression. Likewise, manufactured solutions to higher dimensional problems can be bootstrapped from lower dimensional results, and solutions to transient problems can be constructed beginning with corresponding steady cases.

For instance, Table 1 illustrates reduction of the size of the sub-source terms generated by applying the Hierarchic MMS approach to the full three-dimensional Navier–Stokes energy equation (9) using operators defined in (10) and the manufactured solutions in (6).

Notice that manual algebraic simplification reduces the size of the combined generated source terms ($\mathcal{L}$) by 97.8 %. When using hierarchic MMS techniques, much of this simplification work does not need to be repeated. Modifying the viscosity terms to use the Sutherland model only requires recalculating $\mathcal{L}_5$. Although the new operator produces an expanded source subterm of the same multi-million-character complexity as the full Sutherland-model-based energy equation, it is sufficiently simpler that it can be symbolically simplified without inducing software failures from Maple™.

Finally, decomposing the solutions in this way assists in using the solutions for software debugging. In the event of a discrepancy between the manufactured solution and the users code, the user can verify each term individually in order to narrow down which piece of the equation is different.

This decomposition scheme can be useful for more than software debugging. The "pieces" from each operator can be reassembled or added to form additional manufactured solutions. For instance, adding viscosity to the automatic differentiation operator of the Euler equations would form the Navier–Stokes. It is possible to build a library of operators that could be assembled to span a wide class of equations.

**Table 1** Hierarchic MMS decomposition of the three-dimensional Navier–Stokes energy equation

| Term | Before | After |
|---|---|---|
| $\mathcal{L}_1$ | $70.1 \times 10^3$ | $1.1 \times 10^3$ |
| $\mathcal{L}_2$ | $292.8 \times 10^3$ | $4.0 \times 10^3$ |
| $\mathcal{L}_3$ | $11.3 \times 10^3$ | $1.2 \times 10^3$ |
| $\mathcal{L}_4$ | $1.5 \times 10^3$ | $5.8 \times 10^2$ |
| $\mathcal{L}_5$ | $3.1 \times 10^3$ | $1.3 \times 10^3$ |
| $\mathcal{L}$ | $378.9 \times 10^3$ | $8.3 \times 10^3$ |

Values represent number of characters in each source term before and after algebraic manipulations

## 5 Manufactured analytical solution abstraction

Previously we have focused on the main ideas of MMS and generating solutions and source terms using algebraic manipulation software. We now shift the focus to implementing these schemes in software of interest. Currently there appears to be no openly available software package that provides generated MMS source terms, solutions, etc. The work in Sect. 3 led to the conclusion that any effective tool will need to be completely application independent and capable of supporting diverse forms of solutions generated either analytically, with computer algebra systems, automatic differentiation, or other techniques.

Furthermore, given the extensive software development occurring within the PECOS Center, MMS is, and will remain, a critical component of verification. Therefore, it was decided to centralize the Center's MMS efforts into one library to enhance reusability and consistency across the various software packages. This library is called MASA and it is being made publicly available for others to use in their own verification efforts.

### 5.1 Summary of the library

The MASA library is written entirely in C$^{++}$, with both object-oriented and procedural C$^{++}$ interfaces, as well as interfaces providing native procedure bindings for programs written in Fortran 2003 or C. The library has been tested on various systems, and supports Intel, GNU and Portland Group compilers.

This open source library currently provides a generalized interface for manufactured solution and forcing function evaluations, as well as instantiations implementing manufactured solutions for a diverse set of problems in fluid dynamics, heat transfer, etc. Documentation is

generated using Doxygen [2] so that the documentation automatically updates alongside modifications to the source code. MASA is publicly available[3] under the terms of the 2.1 GNU Lesser General Public License. The website hosts software and documentation download links, the MASA email list, as well as a basic bug reporting ticket system.

The C++ interface to MASA supports arbitrary precision. This is accomplished by templating objects and function calls around a "Scalar" argument, where "Scalar" could be float, double, long double, or any numeric class which implements the necessary overloaded math operators and standard trigonometric functions. For example, in a PDE including a conservation equation for momentum component $\rho u$, `masa_eval_source_rho_u<double>(x,y)` provides the manufactured source term in double precision, while `masa_eval_source_rho_u<float>(x,y)` calculates the same value in single precision.

### 5.2 Example of using MASA for MMS

Figure 1 details a typical verification process using the Euler equations as an example. It also provides examples of the MASA API.

Applications are required to `#include <masa.h>` in C/C++ or `use masa` in Fortran. The initialization of a particular manufactured solution can be accomplished with `masa_init(string s1, string s2)`. The first string (e.g. "`my_sol`") provides a unique identifier for the manufactured solution for use in later procedural APIs. The second string (e.g. "`euler_2d`") selects the particular manufactured solution class to load. In this way, MASA supports sequential or simultaneous use of multiple manufactured solutions, including multiple differently-parameterized instantiations of the same solution type. The procedural interface for this and all subsequent function calls are similar between C, C++, and Fortran.

MASA provides functions that assign default values of parameters required for manufactured solutions (these values are set by default, but can be reset by calling `masa_set_param()`). Default parameters are selected in such a way as to produce solutions that are physically realizable.

Finally, with the problem initialized, the user can evaluate source and manufactured solution terms. For example, the source term for the u-component of the two-dimensional momentum equation is evaluated at a point $(x, y)$ by calling `masa_eval_source_rho_u(x,y)`. Likewise, the manufactured analytical solution for the u-component of velocity at $(x, y)$ would be `masa_eval_exact_u(x,y)`.

[3] https://red.ices.utexas.edu/projects/software. Please note that the server certification fingerprint requires a modern web browser.

Evaluating the analytical solution terms in this way is one of the principal methods of software verification available to MASA users. This is accomplished by choosing an appropriate error norm ($H^1, L_\infty$, etc.) and comparing the evaluations of the manufactured analytical solution over the entire domain with evaluations of the numerical approximation produced by the users' code. Repeating this process while increasing the code discretization fidelity produces a plot of the error in the numerical solution, which should converge at a rate predicted by the order of the discretization scheme and smoothness of the solution. An example of such a convergence plot is shown in Fig. 2, which uses manufactured solutions from MASA for the steady, one-dimensional, nonlinear heat equation to verify numerical solutions from uniform refinement of a low order finite element method code. Similar plots can be used to verify convergence of adaptive discretization heuristics, quantity-of-interest functionals, transient schemes using unsteady manufactured solutions, etc.

The process by which a convergence study can be performed is detailed in Fig. 1, and is easily performed using the MASA library. The user should begin by choosing an appropriate norm and, then, calculate the error between the solution computed by the software to be verified and with the manufactured solution provided from MASA. This process should then be iterated over a variety of mesh sizes. Finally, the nonlinear fit of the error as a function of the mesh size should return a exponent that matches the expected order of accuracy for the spatial discretization scheme. One can often visually verify a codebase by comparing the slope of the error as a function of mesh size on a log–log plot against a function that converges at the same order as the numerical scheme.

Other functions are provided through MASA to supplement the core of the MASA API depicted in Fig. 1. For example, analytic gradient evaluations of the manufactured analytical solutions are available, allowing users to apply Neumann and/or mixed boundary conditions or to verify $H^1$ solution norms. The gradient is returned from calls to functions such as `masa_eval_grad_rho(Scalar, Scalar, Scalar, int)` (for a three-dimensional density field, $\rho$), where the integer argument specifies the component direction of the gradient, and the preceding three scalar arguments specify the spatial location where the source term should be evaluated.

### 5.3 Currently available manufactured solutions

Table 2 lists all the currently available manufactured solutions available in MASA, as of version 0.40. All of these solutions can be called from programs written in C, C++, and Fortran 2003.

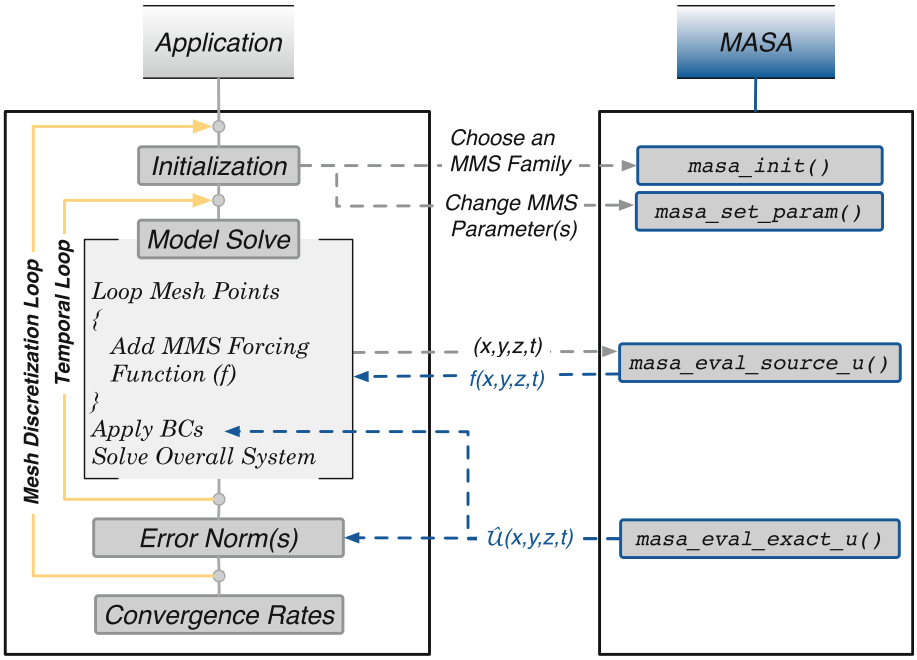



**Fig. 1** Example application flowchart using manufactured solutions from the MASA library to perform a convergence analysis

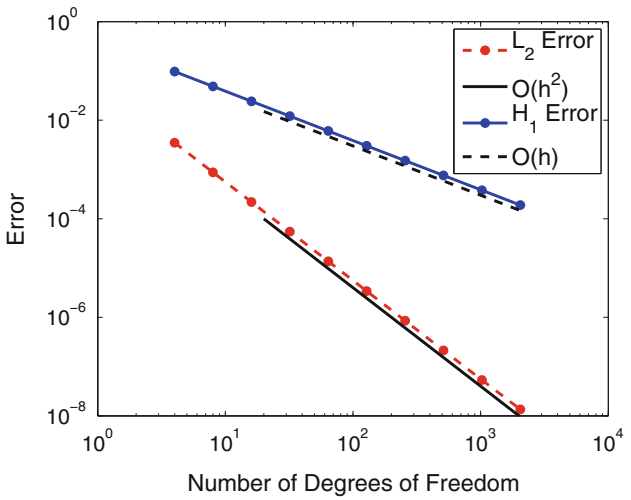



**Fig. 2** Convergence plot for first order uniform refinement test, generated using the steady, nonlinear, one-dimensional heat equation in MASA

**Table 2** Summary of currently available MMS solutions in MASA

| Equations | Dimensions | Time |
|---|---|---|
| Euler | 1,2,3, axi | Transient, Steady |
| Nonlinear heat conduction | 1,2,3 | Transient, Steady |
| Navier–Stokes | 1,2,3, axi | Transient, Steady |
| N–S + Sutherland | 3 | Transient, Steady |
| N–S + ablation | 1 | Transient, Steady |
| Burgers | 2 | Transient, Steady |
| Sod shock tube | 1 | Transient |
| Euler + chemistry | 1 | Steady |
| RANS: Spalart–Allmaras | 1 | Steady |
| Favre averaged N–S: S–A | 2 | Steady |
| Radiation: spectral line | 1 | Steady |
| Conjugate prior: Gaussian | 1 | Steady |

Manufactured solutions for the heat equation are available in steady or unsteady forms, and with variable or constant coefficients. The current solutions available for Navier–Stokes, and Spalart–Allmaras Reynolds-averaged Navier–Stokes (RANS) are all steady. For the Euler equations, MASA has MMS solutions for steady and transient perfect gas flows, as well as a solution for chemical non-equilibrium flow modeling nitrogen dissociation.

The PECOS Center staff will continue to add solutions as needed for their own verification efforts. As a result, it is expected that future solution sets incorporated into MASA will include:

- Additional RANS models ($v^2 - f$, $k - \epsilon$, etc.)
- Oblique shock Euler
- One-dimensional Navier–Stokes with shocks.

In the longer term, the MASA library is designed so that the API is sufficiently general for a broad range of problems, and is certainly not constrained to the solutions described here. In principle, manufactured solutions could be included in MASA for any physical domain, such as Einstein's field equations in general relativity or the Schrödinger equation in quantum mechanics.

It is the hope of the authors that additional solutions will be developed and incorporated into MASA by the computational science community at large. With this in mind, a generalized interface has been developed allowing users to

provide C-code files for source terms and register their own manufactured solutions. This will provide users all the tools built into MASA while ideally streamlining the eventual reintegration and distribution of these solutions.

### 5.4 Importing new solutions

MASA permits users to import additional solutions directly into the codebase. It is the hope of the developers that this import mechanism will streamline the process of adding and testing manufactured solutions, which in turn will encourage users to contribute patch files with additional manufactured solutions to MASA, expanding the scope of the available solutions and broadening its appeal as a community tool.

To import a solution into MASA, the user is required to have:

- C/C$^{++}$ file with your source terms.
- C/C$^{++}$ file containing the analytical solution functions.
- Text file listing the manufactured solution parameters, as well as any default values (if any).

at which point executing a simple script will complete the process of integrating the solution into the MASA codebase. Future work will expand this functionality to permit the user to include analytically calculated gradient functions, import unit tests for source terms, and directly import LaTeX documents into the MASA documentation.

### 5.5 Library verification

We conclude our discussion of the MASA library with a discussion of the verification of the library itself. A library purporting to provide methods of software verification should itself be held to extremely strict software standards. With this in mind, tremendous care has been put into testing the MASA library. The focus of testing of the library is comparing output of expressions (source terms, solutions, etc.) generated using computer algebra systems to the expressions instantiated in the MASA library itself. We enforce strict precision requirements on the source term evaluations (including greater than double precision using templated arguments for arbitrary precision): on local machines, the tests use an **absolute** error less than $1 \times 10^{-15}$, while on other machines, the tolerance is provided up to a relative error of $1 \times 10^{-15}$ (for double precision).

These strict requirements are difficult or impossible to achieve with standard compiler optimization levels. For example, using the Intel compiler, with all compiler optimizations disabled (`-O0`), the floating-point behavior needed additional constraints, namely the `-fp-model precise` flag. Similar flags were required for GNU and Portland Group compilers, and as a result these flags are and are now enabled in MASA by default.

These, and other tests, are encapsulated into over 60 unit tests provided through a "make check" target within the Autotools [9] build system that MASA uses. These tests were found to provide an estimated 98 % + line coverage using the `gcov` [10] tool. Using `Buildbot` [1], the test suite is run, automatically, upon each revision to the codebase, comparing the current library output against the generated manufactured solution.

## 6 Concluding remarks

MMS is an important tool in the suite of tools for code verification of software used for mathematical modeling. Although the use of computer algebra systems for symbolic manipulation, such as Maple$^{TM}$ or SymPy, can greatly assist in the generation of source terms, gradient evaluations, etc., it is not yet an "automatic" process. For complicated models, such as the three-dimensional Navier–Stokes equations, exceedingly complex source terms are generated. Adding the equations from even a simple viscosity model can be enough to cause industry standard symbolic algebra code to fail, presumably from combinatorial explosion. However, by decomposing model equations into individual operators and using the algebraic manipulators on individual terms, the process becomes much more tractable, although it still requires significant human effort.

In order to supply the results of that effort in the form of a reusable tool to the computational science community at large, we have developed the MASA library to ease the burden of incorporating MMS into scientific software. The library is open-source and designed for easy extension to incorporate additional physical models beyond those currently in the library. Both using and extending the library are straightforward tasks, and examples of accomplishing this are provided in the MASA documentation.

We feel that this work represents a strong first step in the direction of encouraging a wider adoption of the MMS techniques in the computational science community, but there are still many interesting avenues to explore. Multiphysics problems coupled between subdomains present a significant challenge, both for generating usable source terms and exercising all numerical components of the mathematical operators. Also, many compressible flow applications have solutions with sharp internal layers or discontinuities (shocks) that require special numerical methods. As a result, smooth manufactured solutions will not fully exercise all the terms of standard numerical formulations. There is some work for inviscid reacting flows with shocks [22], but, to the best knowledge of the authors,

the problem of generating viscous shock-containing solutions remains an open question using the standard MMS techniques; further research will be required.

Another interesting direction, somewhat different than those explored in this paper, is in the realm of statistical algorithms. A major focus of the PECOS Center is validation and uncertainty quantification of mathematical models and quantities of interest. The Bayesian paradigm is central to the development of uncertainties in the mathematical models and the QUESO (Quantification of Uncertainty for Estimation, Simulation and Optimization) library [23], developed within the PECOS Center, is heavily leveraged for these studies. However, to the knowledge of the authors, there has been no MMS infrastructure developed to exercise *statistical* mathematical software in the way described in this work. This will be the focus of forthcoming work.

**Acknowledgments** This material is based in part upon work supported by the Department of Energy [National Nuclear Security Administration] under Award Number [DE-FC52-08NA28615]. Algebraic manipulations for generating source terms in this paper were performed using Maple™ and SymPy. We appreciate the many interesting discussions with the other staff and faculty of the PECOS project. We are particularly grateful to Onkar Sahni for the many helpful comments during the development of MASA.

## References

1. (2010) Buildbot 0.8.1. http://trac.buildbot.net/
2. (2010) Doxygen 1.6.3. http://www.doxygen.org/index.html
3. Babuska I, Oden JT (2004) Verification and validation in computational engineering and science: basic concepts. Comput Methods Appl Mech Eng 193(36–38):4057–4066
4. Bond RB, Ober CC, Knupp PM, Bova SW (2007) Manufactured solution for computational fluid dynamics boundary condition verification. AIAA J 45(9):2224–2236
5. Eça L, Hoekstra M, Hay A, Pelletier D (2007) A manufactured solution for a two-dimensional steady wall-bounded incompressible turbulent flow. Int J Comput Fluid Dyn 21(3–4):175–188
6. Eça L, Hoekstra M, Hay A, Pelletier D (2007) Verification of RANS solvers with manufactured solutions. Eng Comput 23(4):253–270
7. Ellis JR, Hall CD (2009) Model development and code verification for simulation of electrodynamic tether system. J Guid Control Dyn 32(6):1713–1722
8. Fike JA, Alonso JJ (2011) The development of hyper-dual numbers for exact second-derivative calculations. In: AIAA paper 2011-886, 49th AIAA Aerospace Sciences Meeting
9. GNU (2009) Autoconf 2.65. http://www.gnu.org/software/autoconf/
10. GNU (2010) Gcov 4.4.3. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
11. Griewank A (1989) On automatic differentiation. In: In mathematical programming: recent developments and applications. Kluwer Academic Publishers, The Nethjerlands, pp 83–108
12. Kernighan BW, Pike R (1999) The Practice of Programming (Addison-Wesley Professional Computing Series). Addison-Wesley Professional, Boston
13. Knupp P, Salari K (2003) Verification of computer codes in computational science and engineering. Discrete mathematics and its applications. Chapman & Hall/CRC, London
14. Long K, Kirby R, Waanders BVB (2010) Unified embedded parallel finite element computations via software-based frechet differentiation. SIAM J Sci Comput 32(6):3323–3351
15. McClarren RG, Lowrie RB (2008) Manufactured solutions for the P-1 radiation-hydrodynamics equations. J Quant Spectrosc Radiat Transf 109(15):2590–2602
16. Oberkampf WL, Roy CJ (2010) Verification and validation in scientific computing. Cambridge University Press, Cambridge
17. Oberkampf WL, Trucano TG (2002) Verification and validation in computational fluid dynamics. Prog Aerosp Sci 38(3):209–272
18. Oberkampf WL, Trucano TG (2008) Verification and validation benchmarks. Nucl Eng Des 238(3):716–743
19. Oden JT, Belytschko T, Fish J, Hughes TJ, Johnson C, Keyes D, Laub A, Petzold L, Srolovitz D, Yip S (2006) Revolutionizing engineering science through simulation. Tech rep, National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science (SBES)
20. Oliver TA, Estacio-Hiroms KC, Malaya N, Carey GF (2012) Manufactured solutions for the Favre-averaged Navier–Stokes equations with eddy-viscosity turbulence models. In: 50th AIAA Aerospace Sciences Meeting, AIAA 2012-0080
21. Pelletier D, Turgeon E, Tremblay D (2004) Verification and validation of impinging round jet simulations using an adaptive FEM. Int J Numer Methods Fluids 44(7):737–763
22. Powers JM, Aslam TD (2006) Exact solution for multidimensional compressible reactive flow for verifying numerical algorithms. AIAA J 45:337–344
23. Prudencio E, Schulz KW (2012) The parallel C++ statistical library QUESO: Quantification of uncertainty for estimation, simulation and optimization. In: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol 7155. Springer, Berlin, Heidelberg, pp 398–407. http://dx.doi.org/10.1007/978-3-642-29737-3_44
24. Roache PJ (1998) Verification and validation in computational science and engineering. Hermosa Publishers, Socorro
25. Roache PJ (2002) Code verification by the method of manufactured solutions. J Fluids Eng Transact ASME 124(1):4–10
26. Roy CJ (2005) Review of code and solution verification procedures for computational simulation. J Comput Phys 205(1):131–156
27. Roy CJ, Smith TM, Ober CC (2002) Verification of a compressible CFD code using the method of manufactured solutions. In: 32nd AIAA Fluid Dynamics Conference and Exhibit, AIAA 2002–3110
28. Roy CJ, Tendean E, Veluri SP, Rifki R, Hebert S, Luke EA (2007) Verification of rans turbulence models in Loci-CHEM using the method of manufactured solutions. In: 18th AIAA Computational Fluid Dynamics Conference, AIAA 2007–4203
30. Roy CJ, Nelson CC, Smith TM, Ober CC (2004) Verification of Euler/Navier–Stokes codes using the method of manufactured solutions. Int J Numer Meth Fluids 44(6):599–620
30. Salari K, Knupp P (2000) Code verification by the method of manufactured solutions. Tech Rep SAND2000 - 1444, Sandia National Laboratories
31. Shih TM (1985) A procedure to debug computer-programs. Int J Numer Meth Fluids 21(6):1027–1037
32. Silva HG, Souza LF, Medeiros MAF (2009) Verification of a mixed high-order accurate DNS code for laminar turbulent transition by the method of manufactured solutions. Int J Numer Meth Fluids 64(3):336–354
33. Steinberg S, Roache PJ (1985) Symbolic manipulation and computational fluid dynamics. J Comput Phys 57(2):251–284
34. Sutherland W (1893) The viscosity of gases and molecular force. Phil Mag 5(36):507–531
35. Tremblay D, Etienne S, Pelletier D (2006) Code Verification and the Method of Manufactured Solutions for Fluid-Structure Interaction Problems. In: 36th AIAA Fluid Dynamics Confernce, AIAA 2006–3218, pp 882–892