A Scientific Software Verification Library Based on the Method of Manufactured Solutions

David D. Marshall

California Polytechnic State University, San Luis Obispo, CA 93407-0932, USA

A software library, still in development in the C++ programming language that applies the Method of Manufactured Solutions (MMS) to a variety of partial differential equation (PDE) problems. This library will allow researchers to employ MMS as a software verification process without developing significant amounts of testing code. The library is split into three components: solution classes which can be used as manufactured solutions to PDE problems; PDE problem classes which represent specific types of PDEs to be solved (such as linear convection-diffusion equation or Poisson's equation); and post-processing classes that collect the convergence information and can perform various analysis techniques to the convergence data. In use, any solution class can be used with any PDE problem class and vice versa. This creates a significant amount of flexibility in this architecture and allows the end users to customize their MMS testing process. In addition, end users are able to develop their own solution classes in one of three ways: inheriting from the solution base class and implementing their own class; providing the functions (as source code to be compiled or as a software library with exported functions) required to evaluate the solution and its derivatives; or providing the solution equation as a string to be parsed by the library into a function. This paper will demonstrate a number of features of this library, as well as demonstrate its application in a typical use case.

Nomenclature

\[ \delta \quad \text{Convective wave velocity in linear convection-diffusion problem} \]

\[ S \quad \text{Source term in partial differential equation} \]

\[ u \quad \text{Solution to partial differential equation} \]

\( \textbf{Subscripts} \)

\[ \text{CD} \quad \text{Convection-Diffusion Term} \]

\[ \text{mms} \quad \text{Manufactured Solution Term} \]

\( \textbf{Conventions} \)

\[ \text{GCI} \quad \text{Grid Convergence Index} \]

\[ \text{MMS} \quad \text{Method of Manufactured Solutions} \]

\[ \text{PDE} \quad \text{Partial Differential Equation} \]

\( \textbf{Symbols} \)

\[ \nu \quad \text{Diffusivity in linear convection-diffusion problem} \]

I. Introduction

Software verification is an important concept in software development, and it has begun to gain in significance in the area of software development for scientific applications. The distinction between software verification and software validation that is important to keep in mind. "The AIAA Guide for Verification and Validation of Computational Simulations" defines software verification as "the process of determining that a model implementation accurately represents the developer's conceptual description of the model and..."
the solution of the model." In other words, is the code solving the equations that it was intended to solve. For software validation, the AIAA guide says it is "the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model." In other words, is the code modeling the correct equations for the physics of the problem. This paper will focus on the aspect of software verification.

Roache has authored two books on the subject of software verification for scientific and engineering software, that provide an excellent summary of techniques that can be used to verify software. He also presents a thorough description of what information is important to the code developer, who is writing the software, and the experimentalist, who is generating the validation dataset. These two books are excellent references for anyone who is participating in scientific software verification and validation.

The book by Knupf and Salsi4 provides excellent examples of implementing software verification techniques on a wide variety of problems. They also cover implementation issues such as the process of order verification, mesh refinement guidelines for structured and unstructured meshes and the treatment of boundary conditions. They also provide excellent guidance on how to choose a manufactured solution so that it is similar to the expected solution.

Ega and Roekstra have also contributed a number of excellent papers in the field of software verification, They have focused on the use of the Method of Manufactured Solutions, MMS, to perform verification techniques on CFD codes. MMS is a technique where a solution to a differential equation (or set of differential equations) is assumed. These can be either ordinary or partial differential equations. This assumed solution is substituted into the differential equation(s) and terms are added to the original differential equation(s) in order to make the assumed solution an actual solution to the differential equation(s). Section II provides a brief introduction to MMS.

Oberkampf and Roy have recently produced a book16 that discusses a wide variety of verification and validation topics as they relate to scientific software development. They present a variety of techniques that can be used to generate the reference solution that will be used for comparison in the verification process. In addition to MMS, they also discuss a variety of ways of obtaining solutions that are more realistic to the underlying physics of the problem. This can be an important issue when validating differential equations that have a limited range of applicability, such as turbulence models in CFD.

II. Summary of the Method of Manufactured Solutions

The Method of Manufactured Solutions is a useful method of generating a reference solution to differential equations when an exact solution is not known or is too costly to obtain. As an example of how MMS is used, suppose the one-dimensional, linear wave equation with a source term is the differential equation that we are trying to perform software verification upon.

\[
\frac{\partial \phi}{\partial t} + \frac{\partial \phi}{\partial x} - e^{-t} = 0
\]  

(1)

where \( \phi \) is the unknown solution and \( a \) is the wave speed. Notice that the source term was moved to the left hand side so that the entire equation was equal to zero. This is so that the MMS source term is easier to identify.

Suppose that we do not know any exact solution to this partial differential equation, PDE, so we are going to assume that the following equation is the solution

\[
\phi_{\text{mm}} = \sin (\alpha x + \beta t)
\]  

(2)

where \( \alpha \) and \( \beta \) are arbitrary (non-zero) constants. While \( \phi_{\text{mm}} \) is not a solution to the original PDE, it is a solution to the following PDE

\[
\frac{\partial \phi}{\partial t} + \frac{\partial \phi}{\partial x} - e^{-t} = S_{\text{mm}}
\]  

(3)

\[
S_{\text{mm}} = (\alpha x + \beta) \cos (\alpha x + \beta t) - e^{-t}
\]  

(4)

where \( S_{\text{mm}} \) is the source term that needed to be added to the original PDE (1) in order to make (2) a solution. The source term is obtained by substituting the assumed solution into the original PDE (1). Any remaining terms on the left hand side (since we moved the original source term to the left hand side as mentioned above) are the MMS source term.
While this paper will not discuss the concept of boundary conditions as applied to MSS, it is worth noting that since we have the exact solution, then Dirichlet, Neumann or even more complicated boundary conditions can be applied exactly.

The use of MSS as a software verification technique is very promising since it allows the software developer to compare the computed solution to an exact solution. One of the difficulties in implementing this is that for each new differential equation solution that is being solved a new MSS source needs to be derived. In addition, it is believed that MSS can be too intrusive into the source code since the differential equation being solved needs to be modified in order to make the assumed solution an actual solution. This paper presents the preliminary work on a software library, swal, written in C++ that attempts to address these two issues and provide a convenient and easy to use tool for scientific software developers.

III. Software Architecture

A. Overview

There are three main components to swal: The Solution Component, the Equation Component and the Post-Processing Component. Each one is implemented as a separate namespace. The Solution Component implements a variety of standard manufactured solutions. Each class can calculate its value and its derivatives for a given input conditions and adheres to the interface. Since all solutions implement the same interface, they can be interchanged with little effort. The Equation Component utilizes the solution interface to define a variety of differential equations that can be used to perform MSS verification. Currently only a few interfaces have been developed, but more will be implemented in the future. The third component, the Post-Processing Component performs all of the standard post-processing that is typically performed for software verification. Currently, this component collects the solutions for the individual solutions that have been performed and calculates order of accuracy and other information that can be displayed to the user. It is important to remark that these three components only utilize interfaces from the other components as needed and are not tied to any specific instantiable class. Thus the library user is free to mix specific classes as appropriate to meet her needs.

B. Solution Component

The Solution Component is the most fundamental component in swal since it represents the implementation of the manufactured solutions and exists in the solutions namespace within the swal namespace. Figure 1 shows the inheritance hierarchy within this component. The most basic interface is the solution interface. This interface specifies how a solution interfaces with the user to provide the solution and the derivatives of the solution. For this interface there are two methods that represent the interface between the user and the classes. One is the evaluate method that takes a vector that represents the independent parameters and returns the evaluated solution. This is shown in Line 37 in the following code. The other method is the evaluate_partial which takes the same vector of independent parameters and also a vector representing the derivative order(s) desired. For instance, if \( \frac{\partial^n}{\partial x_1 \partial x_2} \) is desired for a solution of three independent variables \( (x_1, x_2, x_3) \), then the vector passed in would be \( [1, 1, 0] \). The method returns the evaluated partial derivative. This is shown in Line 40. Notice

![Figure 1. The Solution Component class inheritance hierarchy.](image)
that these methods are abstract since each solution implementation needs to define the functionality of these methods.

```cpp
namespace sbel1 {
    namespace solution {
        template<types... _value>
        class solution_interface {
            public:
                enum error_code {
                    no_error = 0,
                    invalid_parameter = 1,
                    invalid_variable_index = 2,
                    invalid_derivative_order = 3,
                    derivative_not_available = 4
                };
                typedef unsigned char order_type;
                typedef std::size_t variable_index_type;

                // NOTE: various protected and private methods and members removed

                public:

                solution_interface(const variable_index_type idx) : mvar(idx) {};
                solution_interface(const solution_interface<_value>& f4) : mvar(f1.mvar) {};
                virtual ~solution_interface() {};

                // return the number of variables that this function expects
                variable_index_type number_variables() const { return mvar; };

                // return pointer to copy of function
                virtual solution_interface<_value>* copy() const = 0;

                // evaluate the function for given values and return result
                virtual error_code evaluate(const value evalout, const value val[]) const = 0;

                // evaluate the partial derivative of the function for given values.
                virtual error_code evaluate_partial(const value evalout, const value val[],
                                                     const order_type ord[]) const = 0;
            }
        }
    }
}
```

Figure 1 shows an interface that extends the solution interface called `sbel1`. This interface is a specialization that handles the case where the manufactured solution is a function of one independent parameter. Such examples of this are trigonometric function, and they have been implemented in this library.

Since many manufactured solutions have an explicit time dependence that is fundamentally different that the other independent parameters, there is also an unsteady solution interface that is a specialization of the solution interface. Figure 1 shows this interface and the classes that implement this interface. The following code shows the unsteady solution interface with Lines 14 and 17 showing the unsteady evaluate and the unsteady evaluate partial derivative methods which are abstract methods that need to be implemented by the specific unsteady solution classes. Lines 23 and 27 show the steady interface versions of these methods. The unsteady solution interface implements these by passing the input vector to the steady versions with the first vector element being the time and the rest of the vector elements being the other independent parameters. Thus, no unsteady interface needs to re-implement those, however they can if the wish.

```cpp
namespace sbel1 {
    namespace solution {
        template<types... _value>
        class unsteady_interface : public solution_interface<_value>
```

---

American Institute of Aeronautics and Astronautics
An example implementation of a manufactured solution is one from Knupp and Salari. They present a manufactured solution of the form

$$w_{\text{true}}(t, \vec{x}) = u_0 \left[ \sin (\vec{x} \cdot \omega t) + \vec{d} \right]$$

(9)

that they used for the validation of the linear, convection-diffusion equation.

With the manufactured solution chosen, the solution class and the solution derivatives need to be developed. These derivatives are needed for tasks such as building the MMS source term. The *mms* library has implemented this manufactured solution in the *mms* class. The *mms* class inherits from the *unsteady_interface* class since it is a function of time as well as a function of position. It implements the *evaluate_unsteady* and the *evaluate_unsteady_partial* methods. Below shows an example usage of the *mms* class to evaluate the function, line 10, as well as the partial derivative \( \partial^2 w_{\text{true}} / \partial t^2 \), lines 13 through 15. Notice how few lines are needed to create a fully functional manufactured solution. Lines 1 through 4 are declarations, Line 7 is the initialization of the solution, Line 10 evaluates the solution, Lines 13 and 14 set the partial derivative order, and Line 15 evaluates the derivative.

C. Equation Component

The Equation Component represents a particular differential equation to be used in the MMS technique. It utilizes the solution and unsteady solution interfaces to define the manufactured solution source term. It exists in the *equation* namespace within the *mms* namespace. Figure 2 shows the class hierarchy for this component. The *pde_problem_base* class specifies the basic interface that all equations have. This common
functionality is specifying the MMS solution class and specifying the source term for the differential equation (if it has one). Since there are two types of solution (general and unsteady), there are also two types of differential equation interfaces: steady and unsteady. These classes define how the user interacts with the equation class, either with an explicit time term or not. The code below shows the pde_problem_base class.

```cpp
namespace mms {
    namespace equation {
        template <typename _value, size_t _space_dim>
        class pde_problem_base {
            public:
                enum error_code {
                    no_error = 0,
                    invalid_parameter = 1,
                    space_indices_out_of_range = 2,
                    invalid_solution_class = 3,
                    solution_not_specified = 4,
                    evaluation_not_supported = 5,
                    derivative_order_not_supported = 6,
                    unknown_error = 999,
                };

                // NOTE: various protected and private methods and members removed

                public:
                    pde_problem_base();
                    pde_problem_base(const pde_problem_base<_value, _space_dim> &pb);
                    pde_problem_base();

                    error_code set_solution(const solution<solution_interface<_value> &sol);
                    const solution<solution_interface<_value> * get_solution() const;

                    error_code set_jac(const _value *jac);
                    error_code set_jac(const solution<solution_interface<_value> *jac);
        }
    }
}
```

Figure 2. The Equation Component class inheritance hierarchy.
An example class implemented in this component is the linear convection-diffusion problem. The PDE that it represents is

\[
\frac{\partial u}{\partial t} + \bar{a} \cdot \nabla u - \nu \nabla^2 u - S_{CD} = 0
\]

where \( \bar{a} \) is the convective wave speed, \( \nu \) is the diffusivity, and \( S_{CD} \) is the source term for the convection-diffusion problem. Note that each of those terms can be function of space and time.

In order to accommodate the MMS usage, a modified version of the linear convection-diffusion PDE is actually solved. This modified equation is

\[
\frac{\partial u}{\partial t} + \bar{a} \cdot \nabla u - \nu \nabla^2 u - S_{CD} = S_{\text{mns}}
\]

where \( S_{\text{mns}} \) is the manufactured solution source term that is obtained from

\[
S_{\text{mns}} = \frac{\partial u_{\text{mns}}}{\partial t} + \bar{a} \cdot \nabla u_{\text{mns}} - \nu \nabla^2 u_{\text{mns}} - S_{CD}
\]

with \( u_{\text{mns}} \) representing the specified manufactured solution. Notice that if \( u_{\text{mns}} \) were an actual solution to Eq. 6, then the manufactured solution source term would be identically zero.

The class that represents this is `convection_diffusion_linear_pde`. The following code shows the class declaration. This class provides the user the ability to specify the problem parameters such as the diffusivity, the convective velocity, and the manufactured solution. These can all be specified as constants or as functions of time and space via lines 30 through 63. Lines 17 through 25 are convenience functions that allow the user to calculate specific terms in the convection-diffusion problem. Line 21 is what needs to be called to obtain the MMS source. Line 22 is what can be used to obtain the manufactured solution value. This interface allows the use of spatial and temporal variations for the diffusivity and wave speed by accepting a `function_interface` pointer in the set methods, lines 26 and 30.

```cpp
namespace avalia
{
  namespace equation
  {
    template <typename _value, size_t __space_dim>
    class convection_diffusion_linear_pde : public pde_problem_base<_value, __space_dim>
    
    
    // NOTE: various protected and private methods and members removed
    
    public:
      convection_diffusion_linear_pde();
      convection_diffusion_linear_pde(const convection_diffusion_linear_pde<_value, __space_dim> &dpl);
      "convection_diffusion_linear_pde"();
      
      typename pde_problem_base<_value, __space_dim>::error_code
calculate_diffusion_term(_value aval, const _value &t,
                               const _value x[__space_dim]) const;

      typename pde_problem_base<_value, __space_dim>::error_code
calculate_convection_term(_value aval, const _value &t,
                                const _value x[__space_dim]) const;

      typename pde_problem_base<_value, __space_dim>::error_code
calculate_unsteady_term(_value aval, const _value &t,
                              const _value x[__space_dim]) const;

      typename pde_problem_base<_value, __space_dim>::error_code
calculate_source_term(_value aval, const _value &t,
                           const _value x[__space_dim]) const;

      typenamse pde_problem_base<_value, __space_dim>::error_code
      set_diffusivity(const _value &avd);
      typenamse pde_problem_base<_value, __space_dim>::error_code
      set_velocity(const _value &avd);
  
```

7 of 13

American Institute of Aeronautics and Astronautics
D. Post-Processing Component

The Post-Processing Component is the least well developed component in swell. Currently it has a class to collect the results from one MMS run and a class that stores a collection of results. The results collection can then calculate the order of accuracy, the GCI,\(^2\) tracking the error as reported by the user and other information. These classes exist in the post namespace within the swell namespace.

```cpp
namespace swell {
    namespace post {
        template <typename _data, size_t _NNBRA>
        class case_error_result {
            //
            // NOTE: various protected and private methods and members removed
            //
            public:
                case_error_result();
                case_error_result(const case_error_result&);  
                case_error_result(const case_error_result& data, _NNBRA& ecr);

                _data get_spacing() const;
                void set_spacing(const _data &d);

                _data get_cpu_wall_time() const;
                _data get_wall_elapsed_time() const;
                void set_time(const real_type &); // set time

                size_t get_number_variables() const;
                void set_number_variables(const size_t &n);

        }
    }
}
```
### IV. Application of the Linear Convection-Diffusion Equation

To demonstrate the capabilities of **awell**, it has been used to develop and validate a one-dimensional, linear convection-diffusion equation. The use of this library in the linear convection-diffusion problem required eight lines of code plus another 10 lines for using the post component. The relevant lines of code are shown below. Lines 6 and 7 create the solution and the PDE problem classes. Lines 10, 11 and 12 initialize the PDE problem class (this could all have been done in the constructor in line 7, but was not for the sake of clarity). Lines 21 and 22 set the problem's Dirichlet boundary conditions. Line 23 calculates the RMS source term needed for the problem solution. In the future, the boundary condition interface will be added to the equation interface so that it is clearer to the user how the boundary conditions can be obtained.

```c
    // convection-diffusion variables
    double x(3.2), nu(0.6); // wave speed and diffusion coefficient

    // RMS prescribed solution information
    double cd(1.1), omg(0.0), apm(0.0); // bs fluid solution parameters
    double x1d(1.1), apm1(0.0); // prescribed solution for this case

    // initialize RMS
    cd = set_convection_flux(a);
    cd = set_diffusivity(nu);
    cd = set_solution(all);

    // removed code to set up solver and iterations

    for (int i = 0; i < NITER; ++i)
    {
        // removed pde solver code

        // set the boundary conditions using Dirichlet conditions
        cd = calculate_solution(u(0), t, x(0));
        cd = calculate_solution(u(MAX), t, x(1));

        // calculate the RMS source term to include into solver
        cd = calculate_mass_source_term(omega, t, x(1));

        // removed pde solver code
    }

    // post process data code here
```

This code was used to solve a linear convection-diffusion problem with $a = 3.2$, $\nu = 0.6$, $u_0 = 1.1$, $x = 0.0$ (i.e., a steady problem) and $\varepsilon = 0.01$. The problem domain was $x \in [-0.1, 1.2]$, and the problem was iterated until the change in the RMS error and $L_\infty$ error was less than $1 \times 10^{-6}$. A second order in space, first order in time, fully implicit scheme was used to solve the problem with the Courant number of 100. The calculations were performed on a MacBook Pro with a 3.06 GHz Intel Core 2 Duo CPU. The compiler used was gcc version 4.5.

Figure 3 shows the maximum and the RMS errors for the problem using float, double, and long double (extended precision) data as well as with the double-double and quad-double pseudo-primitive types by Hida et al.\textsuperscript{16,17} Since these classes are templated on the data type used for the calculations, changing the data type is as simple as changing the argument to the class instantiations. When the double double and quad double types were used, the only modification needed was to include the math functions implemented for the double double and quad double libraries since they should be in the std namespace and not the default namespace.\textsuperscript{18}
Note that the problem size is reduced beyond the point where roundoff error overwhelms the truncation error of the problem for float, double, and long double. Since the double double and quad double types are even higher precision, their minimums were not reached. This was done to demonstrate the robustness of the library and its ability to handle any datatype that mimics the primitive datatypes.

Figure 3. Maximum and RMS error for the linear convection-diffusion problem test case using a variety of data types.

Figure 4 shows the resulting CPU timings for the three different cases. This information, along with the information for Figure 3 was all stored and processed by the library. Notice that there is a slight performance penalty for using double (one to two times slower than float) or long double (1.5 to two times slower than double) for the majority of the test cases, however there is certainly a memory penalty. For double double and quad double, the performance penalty is rather severe. Compared to the long double calculations, the double double cases took between seven and eight times as long and the quad double took between 60 and 64 times as long. Note that some of the double and long double cases were not run since they solutions would take multiple days to complete.

Finally, Table 1 shows the calculated spacial order of accuracy for this problem for a variety of data types. Note that all times show very close to second order convergence behavior until the roundoff error becomes dominant. For the float datatype there are not many grid spacing cases before the roundoff error dominates. On the other extreme, the roundoff error does not influence the double double or quad double results for any of the grid spacings.

V. Extending the Applicability of the Library

There are two immediate ways that the user can extend this library. One is that the user can develop their own solution and use it with an existing PDE problem. They can accomplish this in one of three ways. They can create their own class that inherits from the solution interface class and implement the required methods. Another planned approach is for the user to use an existing Computer Algebra System (CAS) to differentiate the user's solution equation and have the CAS generate functions for those derivatives. This is the method that is most commonly used currently with MMS technique. It is envisioned that these functions would be passed to a class derived from the solution interface class and can use these functions to return...
Figure 4. Run-time information for the linear convection-diffusion problem test case using a variety of data types.

<table>
<thead>
<tr>
<th>Grid Spacing</th>
<th>Float Max</th>
<th>RMS</th>
<th>Double Max</th>
<th>RMS</th>
<th>Long Double Max</th>
<th>RMS</th>
<th>Double Double Max</th>
<th>RMS</th>
<th>Quad Double Max</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.125 x 10^{-2}</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
<td>2.01</td>
</tr>
<tr>
<td>4.062 x 10^{-2}</td>
<td>2.01</td>
<td>1.99</td>
<td>2.01</td>
<td>1.99</td>
<td>2.01</td>
<td>1.99</td>
<td>2.01</td>
<td>1.99</td>
<td>2.01</td>
<td>1.99</td>
</tr>
<tr>
<td>2.031 x 10^{-2}</td>
<td>2.01</td>
<td>2.00</td>
<td>2.00</td>
<td>1.99</td>
<td>2.00</td>
<td>1.99</td>
<td>2.00</td>
<td>1.99</td>
<td>2.00</td>
<td>1.99</td>
</tr>
<tr>
<td>1.010 x 10^{-2}</td>
<td>1.99</td>
<td>1.98</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>5.075 x 10^{-3}</td>
<td>2.32</td>
<td>2.50</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>2.530 x 10^{-3}</td>
<td>-4.50</td>
<td>-4.55</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>1.270 x 10^{-3}</td>
<td>-2.76</td>
<td>-2.96</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>6.348 x 10^{-4}</td>
<td>-2.20</td>
<td>-2.26</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>3.174 x 10^{-4}</td>
<td>-2.05</td>
<td>-2.06</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>1.587 x 10^{-4}</td>
<td>-1.80</td>
<td>-1.80</td>
<td>2.02</td>
<td>2.02</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>7.935 x 10^{-5}</td>
<td>-1.56</td>
<td>-1.55</td>
<td>2.45</td>
<td>2.50</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>3.967 x 10^{-5}</td>
<td>-1.00</td>
<td>-1.00</td>
<td>0.40</td>
<td>0.45</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>1.984 x 10^{-5}</td>
<td>-1.95</td>
<td>-1.95</td>
<td>-2.37</td>
<td>-2.35</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>9.918 x 10^{-6}</td>
<td>-1.97</td>
<td>-1.97</td>
<td>3.27</td>
<td>3.45</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>4.959 x 10^{-6}</td>
<td>-2.00</td>
<td>-2.00</td>
<td>-2.52</td>
<td>-2.85</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>2.480 x 10^{-6}</td>
<td>-2.15</td>
<td>-2.13</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>
the requested information. This is intended to be a transition option for those that have existing CAS functions that they do not want to rewrite to integrate into this architecture. The third planned option for users to specify their own solution function via character strings and have a built-in CAS convert that into the desired derivatives needed for the PDE problem. This limited functionality CAS is already capable of parsing equations such as: \(\cos(x^2+y^2+z^2+omega*t)+epsilon\) and creating an equation tree that represents that equation. Once derivatives can be calculated from this parsed equation then this functionality will be integrated into xill.

The second way that the user can extend this library by using the solutions without the PDE problem classes. This mode would be most likely used by users who have problems that cannot be represented by the existing PDE problem classes but want to take advantage of the automatic differentiation capability of the solution classes. This would require more coding on the end user's part, but much less than if they were to attempt the MMS technique without this library.

VI. Conclusions

In its current state xill shows a number of the desired features of an MMS library for software verification. The use of MMS to verify a PDE problem with very few lines of code has been shown with the results of the test also shown. The interfaces to the solution classes and the PDE problem classes have been well established, and the interface to the post-process functionality is evolving as needs are identified. The interfaces are the most important part to the library usefulness, and as time progresses more solution classes, PDE problem classes and post-processing functionality will be developed. In addition wrappers will be developed to allow the use of this library in other programming languages such as C, FORTRAN and MATLAB. Also, more differential equation classes will be developed in order to extend the usefulness of this library to a wider variety of users. While there is more to be done, it is believed that xill represents an excellent first step towards a simple scientific software verification library.

References