49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2011, Orlando, Florida

AIAA 2011-615

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

David D. Marshall [*]

California Polytechnic State University, San Luis Obispo, CA 93407-0352, USA

A software library, `svell`, is being developed 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 utilize 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

$\vec{a}$	Convective wave velocity in linear convection-diffusion problem
S	Source term in partial differential equation
u	Solution to partial differential equation
Subscripts	
CD	Convection-Diffusion Term
mms	Manufactured Solution Term
Conventions	
GCI	Grid Convergence Index
MMS	Method of Manufactured Solutions
PDE	Partial Differential Equation
Symbols	
ν	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[1] software verification as "the process of determining that a model implementation accurately represents the developer's conceptual description of the model and

[*]Associate Professor, Aerospace Engineering Department, Senior Member AIAA.

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[2,3] 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 Knupp and Salari[4] 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.

Eça and Hoekstra have also contributed a number of excellent papers in the field of software verification.[5-14] 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 the 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 book[15] 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 know 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.

$$\frac{\partial \phi}{\partial t} + c\frac{\partial \phi}{\partial x} - e^{-t} = 0 \qquad (1)$$

where ϕ 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

$$\phi_{mms} = \sin(\alpha x + \beta t) \qquad (2)$$

where α and β are arbitrary (non-zero) constants. While ϕ_{mms} is not a solution to the original PDE, it is a solution to the following PDE

$$\frac{\partial \phi}{\partial t} + c\frac{\partial \phi}{\partial x} - e^{-t} = S_{mms} \qquad (3)$$

$$S_{mms} = (c\alpha + \beta)\cos(\alpha x + \beta t) - e^{-t} \qquad (4)$$

where S_{mms} 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 MMS 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 MMS source needs to be derived. In addition, it is believed that MMS 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, `svell`, 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 `svell`: 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 solution 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 MMS verification. Currently only a two 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 `svell` since it represents the implementation of the manufactured solutions and exists in the `solution` namespace within the `svell` 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 $\partial/\partial(x_1 x_2)$ 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

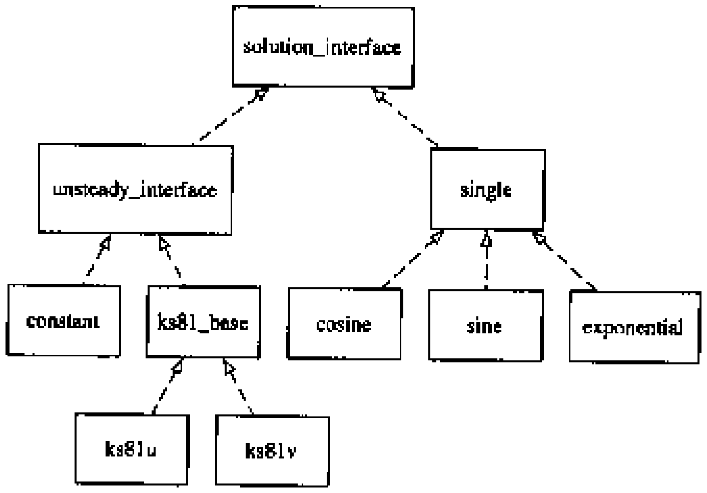

Figure 1. The Solution Component class inheritance hierarchy.

that these methods are abstract since each solution implementation needs to define the functionality of these methods.

```
namespace sveli
{
  namespace solution
  {
    template <typename __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 &nv): nvar(nv) {}
        solution_interface(const solution_interface<__value> &fi) : nvar(fi.nvar) {}
        virtual ~solution_interface() {}

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

        // 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(__value &valout, const __value val[]) const = 0;

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

Figure 1 shows an interface that extends the solution interface called `single`. This interface is a specialization that handles the case where the manufactured solution is a function of one independent parameters. 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 parsing 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 these, however they can if the wish.

```
1  namespace sveli
2  {
3    namespace solution
4    {
5      template <typename __value>
6      class unsteady_interface : public solution_interface<__value>
```

```
      {
        public:
          unsteady_interface(const typename solution_interface<__value>::variable_index_type &nv);
          unsteady_interface(const unsteady_interface<__value> &fi);
          virtual ~unsteady_interface();

          // evaluate the function for given values and return result
          virtual typename solution_interface<__value>::error_code
                evaluate_unsteady(__value &valout, const __value &t, const __value val[]) const = 0;
          // evaluate the partial derivative of the function for given values.
          virtual typename solution_interface<__value>::error_code
                evaluate_unsteady_partial(__value &valout, const __value &t, const __value val[],
                                          const typename solution_interface<__value>::order_type tord,
                                          const typename solution_interface<__value>::order_type ord[]) const = 0;

          // evaluate the function for given values and return result
          virtual typename solution_interface<__value>::error_code
                evaluate(__value &valout, const __value val[]) const;

          // evaluate the partial derivative of the function for given values.
          virtual typename solution_interface<__value>::error_code
                evaluate_partial(__value &valout, const __value val[],
                                 const typename solution_interface<__value>::order_type ord[]) const;
      };
    }
  }
```

An example implementation of a manufactured solution is one from Knupp and Salari.[4] They present a manufactured solution of the form

$$u_{mms}\left(t,\vec{x}\right)=u_0\left[\sin\left(\vec{x}\cdot\vec{x}+\omega t\right)+\varepsilon\right] \tag{5}$$

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 `eveli` library has implemented this manufactured solution in the `ks81u` class. The `ks81u` 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 `ks81u` class to evaluate the function, line 10, as well as the partial derivative $\frac{\partial^5 u_{mms}}{\partial t^2 \partial x^3}$, 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.

```
1   double vout, vpout, x[2] = {1.2, 3.4};
2   double u0(3.2), omega(0.2), epsilon(-0.6);
3   eveli::ks81u<double, 1> k1d;
4   typename eveli::solution::solution_interface<double>::order_type ord[2];
5
6   // set value
7   k1d.set_values(u0, omega, epsilon);
8
9   // evaluate the function at x with return value vout
10  k1d.evaluate(vout, x);
11
12  // evaluate the partial derivative u_{ttxxx} at x with return value vpout
13  ord[0]=2;
14  ord[1]=3;
15  k1d.evaluate_partial(vpout, x, ord);
```

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 `eveli` 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 of 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.

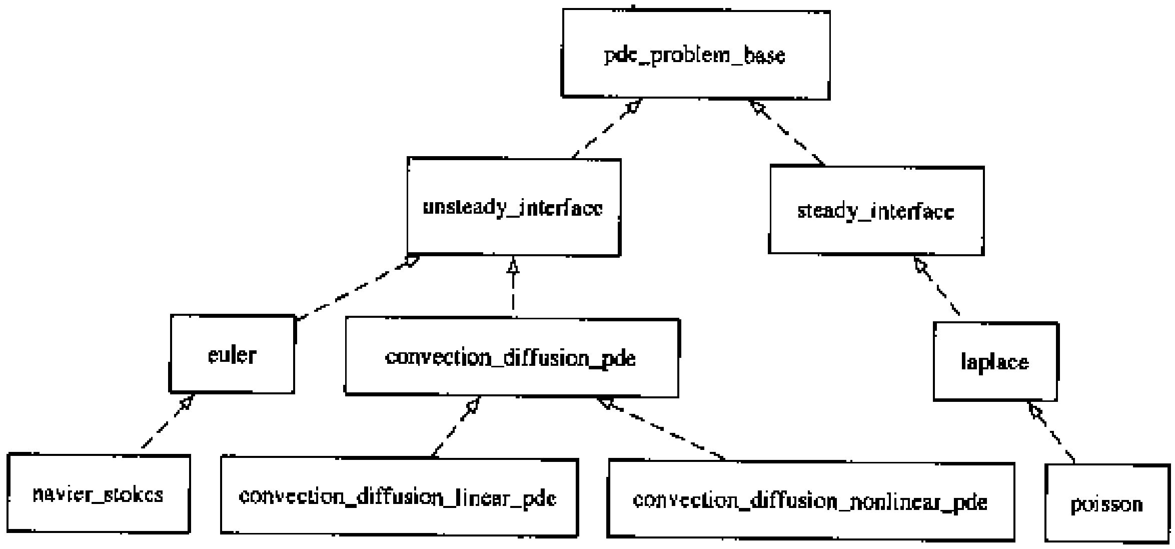

Figure 2. The Equation Component class inheritance hierarchy.

```
namespace evali
{
  namespace equation
  {
    template <typename __value, size_t __space_dim>
    class pde_problem_base
    {
      public:
        enum error_code
        {
          no_error                       = 0,
          invalid_parameter              = 1,
          space_index_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> &ppb);
        ~pde_problem_base();

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

        error_code set_source(const __value &vs);
        error_code set_source(const solution::solution_interface<__value> &src);
```

```
          const solution::solution_interface<__value> * get_source() const;
      };
    }
  }
```

An example class implemented in this component is the linear convection-diffusion problem. The PDE that it represents is

$$\frac{\partial u}{\partial t} + \vec{a} \cdot \vec{\nabla} u - \nu \nabla^2 u - S_{CD} = 0 \tag{6}$$

where $\vec{a}$ is the convective wave speed, ν 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

$$\frac{\partial u}{\partial t} + \vec{a} \cdot \vec{\nabla} u - \nu \nabla^2 u - S_{CD} = S_{mms} \tag{7}$$

where S_{mms} is the manufactured solution source term that is obtained from

$$S_{mms} = \frac{\partial u_{mms}}{\partial t} + \vec{a} \cdot \vec{\nabla} u_{mms} - \nu \nabla^2 u_{mms} - S_{CD} \tag{8}$$

with u_{mms} representing the specified manufactured solution. Notice that if u_{mms} 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, eq. 8. 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.

```
 1  namespace eveli
 2  {
 3    namespace equation
 4    {
 5      template <typename __value, size_t __space_dim>
 6      class convection_diffusion_linear_pde : public pde_problem_base<__value, __space_dim>
 7      {
 8          //
 9          // NOTE: various protected and private methods and members removed
10          //
11
12        public:
13          convection_diffusion_linear_pde();
14          convection_diffusion_linear_pde(const convection_diffusion_linear_pde<__value, __space_dim> &cdlp);
15          ~convection_diffusion_linear_pde();
16
17          typename pde_problem_base<__value, __space_dim>::error_code
18                calculate_diffusion_term(__value &val, const __value &t,
19                                         const __value x[__space_dim]) const;
20          typename pde_problem_base<__value, __space_dim>::error_code
21                calculate_convection_term(__value &val, const __value &t,
22                                          const __value x[__space_dim]) const;
23          typename pde_problem_base<__value, __space_dim>::error_code
24                calculate_unsteady_term(__value &val, const __value &t,
25                                        const __value x[__space_dim]) const;
26          typename pde_problem_base<__value, __space_dim>::error_code
27                calculate_mms_source_term(__value &val, const __value &t,
28                                          const __value x[__space_dim]) const;
29
30          typename pde_problem_base<__value, __space_dim>::error_code
31                set_diffusivity(const __value &vd);
32          typename pde_problem_base<__value, __space_dim>::error_code
```

```
              set_diffusivity(const solution::unsteady_interface<__value> &dif);
        const solution::unsteady_interface<__value> * get_diffusivity() const;

        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_velocity(const __value &vd, const size_t &vindex);
        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_velocity(const solution::unsteady_interface<__value> &vel,
                                      const size_t &vindex);
        const solution::unsteady_interface<__value> *
              get_convective_velocity(const size_t &vindex) const;

        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_x_velocity(const __value &vel);
        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_x_velocity(const solution::unsteady_interface<__value> &vel);
        const solution::unsteady_interface<__value> *
              get_convective_x_velocity() const;

        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_y_velocity(const __value &vel);
        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_y_velocity(const solution::unsteady_interface<__value> &vel);
        const solution::unsteady_interface<__value> *
              get_convective_y_velocity() const;

        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_z_velocity(const __value &vel);
        typename pde_problem_base<__value, __space_dim>::error_code
              set_convective_z_velocity(const solution::unsteady_interface<__value> &vel);
        const solution::unsteady_interface<__value> *
              get_convective_z_velocity() const;
    };
  }
}
```

D. Post-Processing Component

The Post-Processing Component is the least well developed component in evali. 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 evali namespace.

```
namespace evali
{
  namespace post
  {
    template <typename __data, size_t __NERROR>
    class case_error_result
    {
        //
        // NOTE: various protected and private methods and members removed
        //

      public:
        case_error_result();
        case_error_result(const size_t &cid);
        case_error_result(const case_error_result<__data, __NERROR> &cer);

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

        __data get_cpu_elapsed_time() const;
        __data get_wall_elapsed_time() const;
        void set_timer(const gutil::stop_watch &st);

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

        size_t get_case_id() const;
        void set_case_id(const size_t &cid);

        __data get_error(size_t i) const;
        void set_error(const __data &er, size_t i);
    };
  }
}
```

IV. Application of the Linear Convection-Diffusion Equation

To demonstrate the capabilities of sveli, it has been used to develop and validate a one-dimensional, linear convection-diffusion equation. The use of this library in the linear convective-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 25 calculates the MMS 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.

```
 1  // convection-diffusion variables
 2  double a(3.2), nu(0.6); // wave speed and diffusion coefficient
 3
 4  // MMS prescribed solution information
 5  double u0(1.1), omega(0.0), epsilon(0.01);              // ks6luld solution parameters
 6  sveli::ks6lu<double, 1> sol(u0, omega, epsilon);        // prescribed solution for this case
 7  sveli::convection_diffusion_linear_pde<double, 1> cdl; // 1-d, unsteady, linear C-D manufactured solution
 8
 9  // initialize MMS
10  cdl.set_convective_x_velocity(a);
11  cdl.set_diffusivity(nu);
12  cdl.set_solution(sol);
13
14  // removed code to set up solver and iterations
15
16  for (i=0; i<NITER; ++i)
17  {
18    // removed pde solver code
19
20    // set the boundary conditions using Dirichlet conditions
21    cdl.calculate_solution(u[0], t, &(x[0]));
22    cdl.calculate_solution(u[MAX], t, &(x[MAX]));
23
24    // calculate the MMS source to include into solver
25    cdl.calculate_mms_source_term(sterm, t, &(x[i]));
26
27    // removed pde solver code
28  }
29
30  // post process data code here
```

This code was used to solve a linear convection-diffusion problem with $a = 3.2$, $\nu = 0.6$, $u_0 = 1.1$, $\omega = 0.0$ (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_∞ error was less than 1×10^{-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.[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.[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.

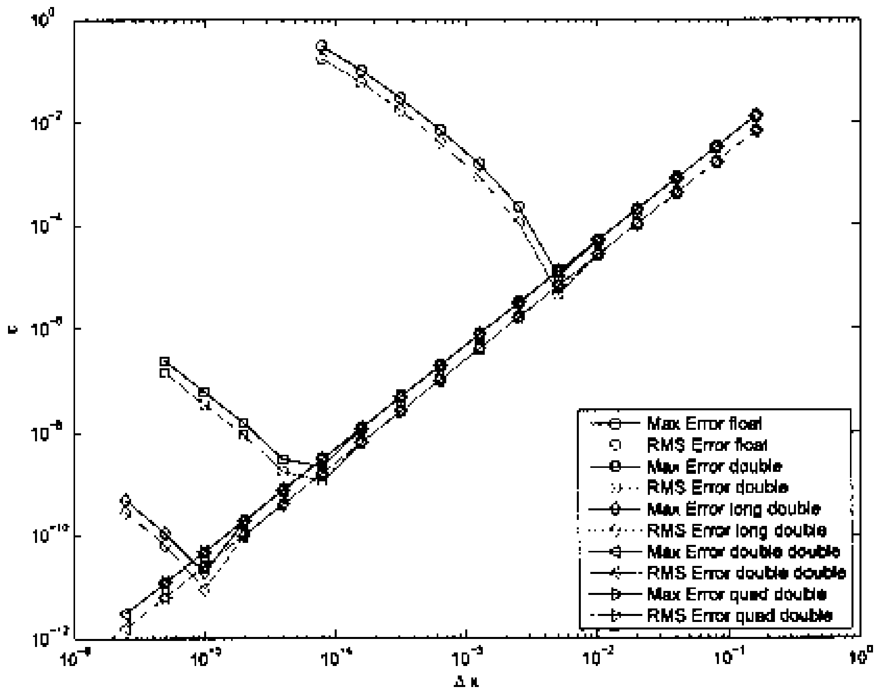

Figure 3. Maximum and RMS error for the linear convection-diffusion problem test case using a variety of data types.

Figure 4 shows the resulting CPU timings for the three difference 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

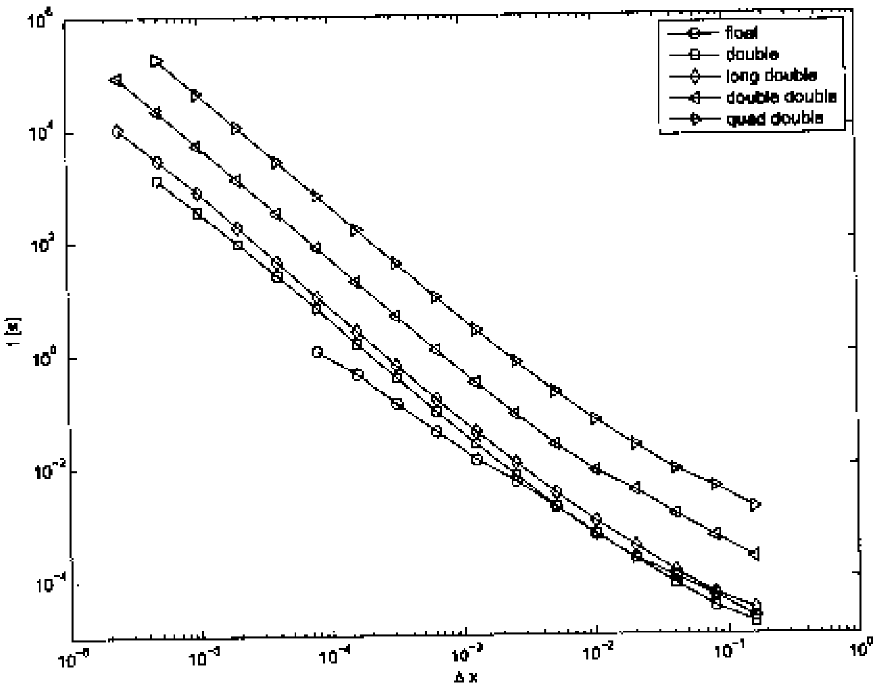

Figure 4. Run-time information for the linear convection-diffusion problem test case using a variety of data types.

Table 1. The order of accuracy for the linear convection-diffusion problem test case using a variety of data types.

	Float		Double		Long Double		Double Double		Quad Double	
Grid Spacing	Max	RMS	Max	RMS	Max	RMS	Max	RMS	Max	RMS
8.125×10^{-2}	2.03	2.01	2.03	2.01	2.03	2.01	2.03	2.01	2.03	2.01
4.063×10^{-2}	2.01	1.99	2.01	1.99	2.01	1.99	2.01	1.99	2.01	1.99
2.031×10^{-2}	2.01	2.00	2.00	1.99	2.00	1.99	2.00	1.99	2.00	1.99
1.016×10^{-2}	1.99	1.98	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
5.078×10^{-3}	2.32	2.50	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
2.539×10^{-3}	-4.50	-4.55	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
1.270×10^{-3}	-2.76	-2.96	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
6.348×10^{-4}	-2.20	-2.26	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
3.174×10^{-4}	-2.05	-2.06	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
1.587×10^{-4}	-1.80	-1.80	2.02	2.02	2.00	2.00	2.00	2.00	2.00	2.00
7.935×10^{-5}	-1.56	-1.55	2.45	2.50	2.00	2.00	2.00	2.00	2.00	2.00
3.967×10^{-5}	—	—	-0.46	-0.65	2.00	2.00	2.00	2.00	2.00	2.00
1.984×10^{-5}	—	—	-2.37	-2.35	2.05	2.06	2.00	2.00	2.00	2.00
9.918×10^{-6}	—	—	-1.97	-1.97	3.27	3.45	2.00	2.00	2.00	2.00
4.959×10^{-6}	—	—	-2.00	-2.00	-2.52	-2.85	2.00	2.00	2.00	2.00
2.480×10^{-6}	—	—	—	—	-2.15	-2.13	2.00	2.00	—	—

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: `u0*(sin(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 `aveli`.

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 `aveli` 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, FORTAN 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 `aveli` represents an excellent first step towards a simple scientific software verification library.

References

[1]American Institute of Aeronautics and Astronautics Staff, "Guide for the Verificaiton and Validation of Computational Fluid Dynamics Simulations," Guide G-077-1998, American Institute for Aeronautics & Astronautics, Reston, VA, 1998.

[2]Roache, P. J., *Verification and Validation in Computational Science and Engineering*, Hermosa Publishers, Albuquerque, NM, 1998.

[3]Roache, P. J., *Fundamentals of Verification and Validation*, Hermosa Publishers, Albuquerque, NM, 2009.

[4]Knupp, P. M. and Salari, K., *Verification of Computer Codes in Computational Science and Engineering*, Discrete Mathematics and Its Applications, Chapman & Hall, 2002.

[5]Eça, L. and Hoekstra, M., "An Evaluation of Verification Procedures for CFD Algorithms," *24th Symposium on Naval Hydrodynamics*, Fukuoka, Japan, July 2002.

[6]Eça, L. and Hoekstra, M., "On the Grid Sensitivity of the Wall Boundary Condition of the $k-\omega$ Turbulence Model," *Journal of Fluids Engineering*, Vol. 126, No. 6, November 2004, pp. 900–910.

[7]Eça, L. and Hoekstra, M., "On the Influence of the Iterative Error in the Numerical Uncertainty of Ship Viscous Flow Calculations," *26th Symposium on Naval Hydrodynamics*, Rome, Italy, September 2006.

[8]Eça, L. and Hoekstra, M., "Verification of Turbulence Models with a Manufactured Solution," *4th European Conference on Computational Fluid Dynamics*, edited by P. Wesseling, E. Oñate, and J. Périaux, ECCOMAS, Egmond aan Zee, The Netherlands, September 2006.

[9]Eça, L. and Hoekstra, M., "An Introduction to CFD Code Verification Including Eddy-Viscosity Models," *4th European Conference on Computational Fluid Dynamics*, edited by P. Wesseling, E. Oñate, and J. Périaux, ECCOMAS, Egmond aan Zee, The Netherlands, September 2006.

[10]Eça, L., Hoekstra, M., Hay, A., and Pelletier, D., "A Manufactured Solution for a Two-Dimensional Steady Wall-Bounded Incompressible Turbulent Flow," *International Journal of Computational Fluid Dynamics*, Vol. 21, No. 3-4, March–April 2007, pp. 175–188.

[11]Eça, L., Hoekstra, M., Hay, A., and Pelletier, D., "On the Construction of Manufactured Solutions for One and Two-Equation Eddy-Viscosity Models," *International Journal for Numerical Methods in Fluids*, Vol. 54, 2007, pp. 119–154.

[12]Eça, L., Hoekstra, M., Hay, A., and Pelletier, D., "Verification of RANS Solvers with Manufactured Solutions," *Engineering with Computers*, Vol. 23, 2007, pp. 253-270.

[13]Eça, L. and Hoekstra, M., "Code Verification of Unsteady Flow Solvers with the Method of Manufactured Solutions," *17th International Offshore and Polar Engineering Conference*, ISOPE, Lisbon, Portugal, July 2007, pp. 2012–2019.

[14]Eça, L. and Hoekstra, M., "Evaluation of Numerical Error Estimation Based on Grid Refinement Studies with the Method of the Manufactured Solutions," *Computers & Fluids*, Vol. 38, No. 8, September 2009, pp. 1580-1591.

[15]Oberkampf, W. L. and Roy, C. J., *Verification and Validation in Scientific Computing*, Cambridge University Press, New York, NY, 2010.

[16]Hida, Y., Li, X. S., and Bailey, D. H., "Quad-Double Arithmetic: Algorithms, Implementation, and Application," Tech. Rep. LBNL-46597, Lawrence Berkeley National Laboratory, October 2000.

[17]Hida, Y., Li, X. S., and Bailey, D. H., "Algorithms for Quad-Double Precision Floating Point Arithmetic," *15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 2001, pp. 155–162, LBNL-46597.

[18]Josuttis, N. M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, New York, NY, 1999.