10th World Congress on Computational Mechanics
8-13 July 2012 • São Paulo • Brazil

A TRANSIENT MANUFACTURED SOLUTION FOR THE COMPRESSIBLE NAVIER–STOKES EQUATIONS WITH A POWER LAW VISCOSITY

Rhys Ulerich[13], Kemelli C. Estacio-Hiroms[1], Nicholas Malaya[1], Robert D. Moser[12]

[1] Institute for Computational Engineering and Sciences, The University of Texas at Austin

[2] Department of Mechanical Engineering, University of Texas at Austin

[3] Corresponding author (rhys@ices.utexas.edu)

Abstract. *A time-varying manufactured solution is presented for the compressible Navier–Stokes equations under the assumption of a constant Prandtl number, Newtonian perfect gas obeying a power law viscosity. The solution is built from waveforms with adjustable phase offsets and mixed partial derivatives to thoroughly exercise all the terms in the equations. Temperature, rather than pressure, is selected to have a simple analytic form to aid verifying codes having temperature-based boundary conditions. In order to alleviate the combinatorial complexity of finding a symbolic expression for the complete forcing terms, a hybrid approach combining the open source symbolic manipulation library SymPy with floating point computations is employed. A C++ implementation of the resulting manufactured solution and the forcing terms are provided. Tests ensure the floating point implementation matches the solution to relative errors near machine epsilon. The manufactured solution was used to verify a new three-dimensional, pseudo-spectral channel code. A verification test for the flat plate geometry is also included. This hybrid manufactured solution generation approach can be extended to either more complicated constitutive relations or multi-species flows.*

Keywords: *Verification, Manufactured solution, Navier–Stokes, Channel flow, Flat plate.*

1. INTRODUCTION

Modeling and simulation find a variety of scientific and engineering uses in applications ranging from industry to finance to governmental planning. Every time computational results are used to inform a decision making process, the credibility of those results becomes crucial [12]. Given this importance, assuming that a valid mathematical model has been chosen to represent the desired phenomena, the verification of the software used to perform these simulations becomes essential.

"Code verification" is a process to determine if a computer program is a faithful representation of the desired mathematical model [17]. Often these models take the form of a set of partial differential (and/or integral) equations along with auxiliary relationships (constitutive laws, boundary/initial conditions). By both employing appropriate software engineering practices and by practicing code verification, it is possible to build a high degree of confidence that there are no inconsistencies in the selected equation discretization algorithms or mistakes in their implementations [11].

Though other approaches may provide insightful results, the most rigorous and widely-accepted criteria for verifying a partial differential equation-based computer program is performing order of accuracy studies [18]. To study the order of accuracy, one investigates whether the program's observed order of accuracy matches the formal order of accuracy, i.e. whether or not the discretization error can be reduced at the expected rate. For this, an exact solution for the underlying problem is required. Unfortunately, analytical solutions are known for only relatively simple problems. For problems of engineering interest, analytical solutions often cannot be found because the relevant physics and/or geometry is too complex.

One can, however, synthesize exact solutions for complex mathematical models using the method of manufactured solutions (MMS). The MMS modifies governing equations by the addition of source terms such that the exact – manufactured – solution is known *a priori* [15, 22]. Order of accuracy studies then may be conducted using the constructed solution. The MMS is a powerful tool for performing order of accuracy studies on coupled nonlinear partial differential equations and its use has become a broadly-accepted methodology for code verification [27, 18]. Indeed, MMS-based code verification has been performed in many simulation areas including turbulence modeling [3], turbulent reacting flows [28], radiation [10], and fluid-structure interaction [26].

In this work, we first briefly review the MMS. Second, we set forth the complete partial differential equations comprising our mathematical model, namely the compressible Navier–Stokes equations for a perfect gas obeying a power-law viscosity. Next a new, time-varying manufactured solution designed to thoroughly exercise all model terms is presented. We then demonstrate a novel way to compute the associated manufactured forcing that circumvents common problems arising from using computer algebra systems for that purpose. We quickly discuss a publicly available solution reference implementation and detail test problems suitable for flow solvers simulating isothermal channel flows and/or flat plates. Afterwards, we share some experiences from using the solution to debug and then verify a new pseudo-spectral code called "Suzerain". Finally, suggestions are made for how our solution and manufactured forcing computation technique could be extended to other cases.

2. THE METHOD OF MANUFACTURED SOLUTIONS (MMS)

The MMS modifies a system of governing equations to construct analytical solutions known *a priori*. The construction of new analytical solution requires two ingredients: a complete description of an equation-based mathematical model and a set of quasi-arbitrary functions describing the desired "manufactured solution". The solution functions, which do not exactly satisfy the model, are substituted into the model equations producing a non-zero residual. By adding the residual, often called the "manufactured source terms" or "manufactured forcing", back into the equations one makes the manufactured solution exactly satisfy the model. The manufactured source terms are the product of the MMS recipe. The modified model, manufactured solution, and source terms are often referred to collectively as a manufactured solution of the original mathematical model. To be useful, all three facets must be correctly implemented in a computer program so a user may perform order of accuracy studies.

Manufactured solutions need to satisfy a few, modest requirements and so their selection is not entirely arbitrary. They must be from the same space of functions as solutions of the unmodified mathematical model. Often, simply being continuously differentiable up to the order required by governing equations and adhering to the relevant boundary conditions is sufficient though more regularity can be advantageous when testing higher order numerics. Ideally, they should also exercise all terms in the original model. Roache [16], followed by Knupp and Salari [6], and Oberkampf and Roy [11] provide well-documented guidelines on verification of scientific computer codes, including construction of manufactured solutions, the application of MMS, and analysis of the results.

3. MATHEMATICAL MODEL

Our mathematical model is the compressible Navier–Stokes equations which may be written as

$$\frac{\partial}{\partial t}\rho = -\nabla\cdot\rho\vec{u} + Q_\rho \tag{1a}$$

$$\frac{\partial}{\partial t}\rho\vec{u} = -\nabla\cdot(\vec{u}\otimes\rho\vec{u}) - \nabla p + \nabla\cdot\overleftrightarrow{\tau} + \vec{Q}_{\rho u} \tag{1b}$$

$$\frac{\partial}{\partial t}\rho e = -\nabla\cdot\rho e\vec{u} - \nabla\cdot p\vec{u} - \nabla\cdot\vec{q} + \nabla\cdot\overleftrightarrow{\tau}\vec{u} + Q_{\rho e} \tag{1c}$$

with the auxiliary relations

$$p = (\gamma-1)\left(\rho e - \rho\frac{\vec{u}\cdot\vec{u}}{2}\right) \tag{1d}$$

$$T = \frac{p}{\rho R} \tag{1e}$$

$$\mu = \mu_r\left(\frac{T}{T_r}\right)^\beta \tag{1f}$$

$$\lambda = \frac{\lambda_r}{\mu_r}\mu \tag{1g}$$

$$\overleftrightarrow{\tau} = \mu\left(\nabla\vec{u} + \nabla\vec{u}^{\mathsf{T}}\right) + \lambda\left(\nabla\cdot\vec{u}\right)\overleftrightarrow{I} \tag{1h}$$

$$\vec{q} = -\frac{\kappa_r}{\mu_r}\mu\nabla T \tag{1i}$$

where all symbols have their customary interpretations. Note e denotes the specific total energy and that the components of velocity $\vec{u}$ will be referred to as the scalars u, v, and w. These equations arise from applying the conservation of mass, momentum, and energy to a Newtonian perfect gas. The model assumes the gas' first viscosity μ obeys a power law in temperature T, the other viscosity λ is a constant multiple of μ, heat conduction through the gas obeys Fourier's law, and momentum and thermal diffusivity are related by a constant

Prandtl number. The arbitrary terms Q_ρ, $\vec{Q}_{\rho u}$, and $Q_{\rho e}$ will be used to force the desired manufactured solution.

The free constants in the model are the ratio of specific heats γ, the gas constant R, the viscosity power law exponent β, and the reference properties μ_r, T_r, κ_r, and λ_r. One fixes κ_r when choosing the Prandtl number Pr because $\kappa_r = \frac{\gamma R \mu_r}{(\gamma-1)\mathrm{Pr}}$. Selecting $\lambda_r = -2\mu_r/3$ recovers Stokes' hypothesis that the bulk viscosity is negligible. The thermal conductivity κ does not appear in the above equations as our constant Prandtl number assumption and the observation that κ increases with μ implies $\mu/\mu_r = \kappa/\kappa_r$.

4. MANUFACTURED SOLUTION

The set of functions $\phi \in \{\rho, u, v, w, T\}$ selected as analytical solution for density, velocity, and temperature are of the form

$$
\begin{aligned}
\phi(x,y,z,t) = {} & a_{\phi 0} && \cos\left(f_{\phi 0}\, t + g_{\phi 0}\right) \\
+ {} & a_{\phi x} \cos\left(b_{\phi x}\, 2\pi x L_x^{-1} + c_{\phi x}\right) && \cos\left(f_{\phi x}\, t + g_{\phi x}\right) \\
+ {} & a_{\phi xy} \cos\left(b_{\phi xy} 2\pi x L_x^{-1} + c_{\phi xy}\right) \cos\left(d_{\phi xy} 2\pi y L_y^{-1} + e_{\phi xy}\right) && \cos\left(f_{\phi xy} t + g_{\phi xy}\right) \\
+ {} & a_{\phi xz} \cos\left(b_{\phi xz} 2\pi x L_x^{-1} + c_{\phi xz}\right) \cos\left(d_{\phi xz} 2\pi z L_z^{-1} + e_{\phi xz}\right) && \cos\left(f_{\phi xz} t + g_{\phi xz}\right) \\
+ {} & a_{\phi y} \cos\left(b_{\phi y}\, 2\pi y L_y^{-1} + c_{\phi y}\right) && \cos\left(f_{\phi y}\, t + g_{\phi y}\right) \\
+ {} & a_{\phi yz} \cos\left(b_{\phi yz} 2\pi y L_y^{-1} + c_{\phi yz}\right) \cos\left(d_{\phi yz} 2\pi z L_z^{-1} + e_{\phi yz}\right) && \cos\left(f_{\phi yz} t + g_{\phi yz}\right) \\
+ {} & a_{\phi z} \cos\left(b_{\phi z}\, 2\pi z L_z^{-1} + c_{\phi z}\right) && \cos\left(f_{\phi z}\, t + g_{\phi z}\right)
\end{aligned}
\tag{2}
$$

where a, b, c, d, e, f, and g are constant coefficient collections indexed by ϕ and one or more directions. To aid in providing reusable, physically realizable coefficients for Cartesian domains of arbitrary size, domain extents L_x, L_y, L_z have been introduced. Each term has an adjustable amplitude, frequency, and phase for all spatial dimensions. Cosines were chosen so all terms can be "turned off" by employing zero coefficients. It is suggested that users gradually "turn on" the more complicated features of the solution (i.e. use non-zero coefficients) after ensuring simpler usage has been successful.

Mixed partial spatial derivatives are included to improve code coverage. Nontrivial mixed spatial velocity derivatives are essential for testing implementations of $\nabla \cdot \overleftrightarrow{\tau}$ and $\nabla \cdot \overleftrightarrow{\tau} \vec{u}$. The addition of these terms therefore represents a marked improvement over earlier solutions by Roy and coworkers for verifying viscous flow solvers [19, 18]. Our approach to computing manufactured forcing, to be discussed in Section 5, trivially affords us the additional complexity these new terms introduce. Others, e.g. Silva et al. [20], have included mixed partial derivative coverage though not for this particular Navier–Stokes formulation.

The manufactured forcing will later require the partial derivatives ϕ_t, ϕ_x, ϕ_y, ϕ_z, ϕ_{xx}, ϕ_{xy}, ϕ_{xz}, ϕ_{yy}, ϕ_{yz}, and ϕ_{zz}. These may be computed by hand and implemented directly from Expression (2). More conveniently, the Python-based computer algebra system SymPy [24] can calculate the derivatives:

```
from sympy import *

# Coordinates
var('x y z t', real=True)

# Solution parameters used in the form of the analytical solution
var(""" a_0  a_x  a_xy  a_xz  a_y  a_yz  a_z
             b_x  b_xy  b_xz  b_y  b_yz  b_z
             c_x  c_xy  c_xz  c_y  c_yz  c_z
                  d_xy  d_xz       d_yz
                  e_xy  e_xz       e_yz
        f_0  f_x  f_xy  f_xz  f_y  f_yz  f_z
        g_0  g_x  g_xy  g_xz  g_y  g_yz  g_z """, real=True)

# Explicitly keep (2 * pi / L) terms together as indivisible tokens
var('twopi_invLx  twopi_invLy  twopi_invLz', real=True)

# Form the analytical solution and its derivatives
phi = (
    a_0                                                                   *cos(f_0 *t + g_0 )
  + a_x  * cos(b_x *twopi_invLx*x + c_x )                                 *cos(f_x *t + g_x )
  + a_xy * cos(b_xy*twopi_invLx*x + c_xy)*cos(d_xy*twopi_invLy*y + e_xy)*cos(f_xy*t + g_xy)
  + a_xz * cos(b_xz*twopi_invLx*x + c_xz)*cos(d_xz*twopi_invLz*z + e_xz)*cos(f_xz*t + g_xz)
  + a_y  * cos(b_y *twopi_invLy*y + c_y )                                 *cos(f_y *t + g_y )
  + a_yz * cos(b_yz*twopi_invLy*y + c_yz)*cos(d_yz*twopi_invLz*z + e_yz)*cos(f_yz*t + g_yz)
  + a_z  * cos(b_z *twopi_invLz*z + c_z )                                 *cos(f_z *t + g_z )
)
phi_t  = phi.diff(t)
phi_x  = phi.diff(x)
phi_y  = phi.diff(y)
phi_z  = phi.diff(z)
phi_xx = phi_x.diff(x)
phi_xy = phi_x.diff(y)
phi_xz = phi_x.diff(z)
phi_yy = phi_y.diff(y)
phi_yz = phi_y.diff(z)
phi_zz = phi_z.diff(z)
```

SymPy will also generate C code for computing these quantities at a given x, y, z, and t:

```
from sympy.utilities.codegen import codegen
codegen((
            ("phi",    phi   ),
            ("phi_t",  phi_t ),
            ("phi_x",  phi_x ),
            ("phi_xx", phi_xx),
            ("phi_xy", phi_xy),
            ("phi_xz", phi_xz),
            ("phi_y",  phi_y ),
            ("phi_yy", phi_yy),
            ("phi_yz", phi_yz),
            ("phi_z",  phi_z ),
            ("phi_zz", phi_zz),
        ), "C", "soln", header=False, to_files=True)
```

5. MANUFACTURED FORCING

The solutions given by Equation (2) may be substituted into Equation (1) and solved for the forcing terms Q_ρ, $\vec{Q}_{\rho u}$, and $Q_{\rho e}$. Doing so by hand is impractical. Doing so in a straightforward manner using a computer algebra system causes an unwieldy explosion of terms. For our solution, even aggressive simplification by SymPy, Mathematica$^{\mathrm{TM}}$, or Maple$^{\mathrm{TM}}$ does not render results usable in any non-mechanical way. Those results will not be shown here.

Malaya et al. [8] suggested mitigating vexing irreducible terms using an approach they called the “hierarchic MMS”. If we employed their hierarchic MMS approach here, we would express contributions to our forcing functions based on terms in the original equations. For example, one might define $Q_{\rho e}^{\nabla\cdot\overset{\leftrightarrow}{\tau}\vec{u}}$ which would represent the contribution to $Q_{\rho e}$ coming from substituting Equation (2) into only the $\nabla\cdot\overset{\leftrightarrow}{\tau}\vec{u}$ term from Equation (1). Still, our hypothetical $Q_{\rho e}^{\nabla\cdot\overset{\leftrightarrow}{\tau}\vec{u}}$ would be too lengthy for casual human consumption and it would require painstaking effort to manually check the results returned by a computer algebra system.

Moreover, in both the traditional and the hierarchic MMS approaches, the manufactured forcing computation depends strongly on the chosen solution form. Positing any local change to the solution entirely changes these global symbolic results. As the pointwise system evolution is governed only by local state and its derivatives, one should be able to write the manufactured forcing using only pointwise information from the solution.

Wishing to avoid these deficiencies, we start from ρ, ρ_t, ρ_x, etc. and use the chain rule and algebra to obtain a sequence of expressions for computing the forcing using SymPy:

```
# Assuming that we are given
#   rho, rho_t, rho_x, rho_xx, rho_xy, rho_xz, rho_y, rho_yy, rho_yz, rho_z, rho_zz
#   u,   u_t,   u_x,   u_xx,   u_xy,   u_xz,   u_y,   u_yy,   u_yz,   u_z,   u_zz
#   v,   v_t,   v_x,   v_xx,   v_xy,   v_xz,   v_y,   v_yy,   v_yz,   v_z,   v_zz
#   w,   w_t,   w_x,   w_xx,   w_xy,   w_xz,   w_y,   w_yy,   w_yz,   w_z,   w_zz
#   T,   T_t,   T_x,   T_xx,   T_xy,   T_xz,   T_y,   T_yy,   T_yz,   T_z,   T_zz
# and the coefficients
#   gamma, R, beta, mu_r, T_r, kappa_r, lambda_r
# compute the source terms
#   Q_rho, Q_rhou, Q_rhov, Q_rhow, Q_rhoe
# necessary to force the solution rho, u, v, w, and T.
```

First come computations from the auxiliary relations (1d)–(1i):

```
e        = R * T   / (gamma - 1) + (u*u   + v*v   + w*w  ) / 2
e_x      = R * T_x / (gamma - 1) + (u*u_x + v*v_x + w*w_x)
e_y      = R * T_y / (gamma - 1) + (u*u_y + v*v_y + w*w_y)
e_z      = R * T_z / (gamma - 1) + (u*u_z + v*v_z + w*w_z)
e_t      = R * T_t / (gamma - 1) + (u*u_t + v*v_t + w*w_t)
p        = rho * R * T
p_x      = rho_x * R * T + rho * R * T_x
p_y      = rho_y * R * T + rho * R * T_y
p_z      = rho_z * R * T + rho * R * T_z
mu       = mu_r * pow(T / T_r, beta)
mu_x     = beta * mu_r / T_r * pow(T / T_r, beta - 1) * T_x
mu_y     = beta * mu_r / T_r * pow(T / T_r, beta - 1) * T_y
mu_z     = beta * mu_r / T_r * pow(T / T_r, beta - 1) * T_z
lambda_  = lambda_r / mu_r * mu   # "lambda" is a Python keyword
lambda_x = lambda_r / mu_r * mu_x
lambda_y = lambda_r / mu_r * mu_y
lambda_z = lambda_r / mu_r * mu_z
qx       = - kappa_r / mu_r *  mu   * T_x
qy       = - kappa_r / mu_r *  mu   * T_y
qz       = - kappa_r / mu_r *  mu   * T_z
qx_x     = - kappa_r / mu_r * (mu_x * T_x + mu * T_xx)
qy_y     = - kappa_r / mu_r * (mu_y * T_y + mu * T_yy)
qz_z     = - kappa_r / mu_r * (mu_z * T_z + mu * T_zz)
```

Next, we obtain all the terms from the partial differential Equations (1a)–(1c):

```
rhou    = rho * u
rhov    = rho * v
rhow    = rho * w
rhoe    = rho * e
rhou_x  = rho_x * u + rho * u_x
rhov_y  = rho_y * v + rho * v_y
rhow_z  = rho_z * w + rho * w_z
rhou_t  = rho_t * u + rho * u_t
rhov_t  = rho_t * v + rho * v_t
rhow_t  = rho_t * w + rho * w_t
```

```
rhoe_t  = rho_t * e + rho * e_t

rhouu_x = (rho_x * u * u) + (rho * u_x * u) + (rho * u * u_x)
rhouv_y = (rho_y * u * v) + (rho * u_y * v) + (rho * u * v_y)
rhouw_z = (rho_z * u * w) + (rho * u_z * w) + (rho * u * w_z)
rhouv_x = (rho_x * u * v) + (rho * u_x * v) + (rho * u * v_x)
rhovv_y = (rho_y * v * v) + (rho * v_y * v) + (rho * v * v_y)
rhovw_z = (rho_z * v * w) + (rho * v_z * w) + (rho * v * w_z)
rhouw_x = (rho_x * u * w) + (rho * u_x * w) + (rho * u * w_x)
rhovw_y = (rho_y * v * w) + (rho * v_y * w) + (rho * v * w_y)
rhoww_z = (rho_z * w * w) + (rho * w_z * w) + (rho * w * w_z)
rhoue_x = (rho_x * u * e) + (rho * u_x * e) + (rho * u * e_x)
rhove_y = (rho_y * v * e) + (rho * v_y * e) + (rho * v * e_y)
rhowe_z = (rho_z * w * e) + (rho * w_z * e) + (rho * w * e_z)

tauxx = mu * (u_x + u_x) + lambda_ * (u_x + v_y + w_z)
tauyy = mu * (v_y + v_y) + lambda_ * (u_x + v_y + w_z)
tauzz = mu * (w_z + w_z) + lambda_ * (u_x + v_y + w_z)
tauxy = mu * (u_y + v_x)
tauxz = mu * (u_z + w_x)
tauyz = mu * (v_z + w_y)

tauxx_x = (   mu_x * (u_x  + u_x ) + lambda_x * (u_x  + v_y  + w_z )
            + mu   * (u_xx + u_xx) + lambda_  * (u_xx + v_xy + w_xz) )
tauyy_y = (   mu_y * (v_y  + v_y ) + lambda_y * (u_x  + v_y  + w_z )
            + mu   * (v_yy + v_yy) + lambda_  * (u_xy + v_yy + w_yz) )
tauzz_z = (   mu_z * (w_z  + w_z ) + lambda_z * (u_x  + v_y  + w_z )
            + mu   * (w_zz + w_zz) + lambda_  * (u_xz + v_yz + w_zz) )

tauxy_x = mu_x * (u_y + v_x) + mu * (u_xy + v_xx)
tauxy_y = mu_y * (u_y + v_x) + mu * (u_yy + v_xy)
tauxz_x = mu_x * (u_z + w_x) + mu * (u_xz + w_xx)
tauxz_z = mu_z * (u_z + w_x) + mu * (u_zz + w_xz)
tauyz_y = mu_y * (v_z + w_y) + mu * (v_yz + w_yy)
tauyz_z = mu_z * (v_z + w_y) + mu * (v_zz + w_yz)

pu_x = p_x * u + p * u_x
pv_y = p_y * v + p * v_y
pw_z = p_z * w + p * w_z
utauxx_x = u_x * tauxx + u * tauxx_x
vtauxy_x = v_x * tauxy + v * tauxy_x
wtauxz_x = w_x * tauxz + w * tauxz_x
utauxy_y = u_y * tauxy + u * tauxy_y
vtauyy_y = v_y * tauyy + v * tauyy_y
wtauyz_y = w_y * tauyz + w * tauyz_y
utauxz_z = u_z * tauxz + u * tauxz_z
vtauyz_z = v_z * tauyz + v * tauyz_z
wtauzz_z = w_z * tauzz + w * tauzz_z
```

Finally, we directly compute Q_ρ, the three components of $\vec{Q}_{\rho u}$, and $Q_{\rho e}$:

```
Q_rho  = rho_t  + rhou_x + rhov_y + rhow_z
Q_rhou = ( rhou_t + rhouu_x + rhouv_y + rhouw_z + p_x - tauxx_x - tauxy_y - tauxz_z )
Q_rhov = ( rhov_t + rhouv_x + rhovv_y + rhovw_z + p_y - tauxy_x - tauyy_y - tauyz_z )
Q_rhow = ( rhow_t + rhouw_x + rhovw_y + rhoww_z + p_z - tauxz_x - tauyz_y - tauzz_z )
Q_rhoe = ( rhoe_t + rhoue_x + rhove_y + rhowe_z
                  + pu_x + pv_y + pw_z + qx_x + qy_y + qz_z
                  - utauxx_x - vtauxy_x - wtauxz_x
                  - utauxy_y - vtauyy_y - wtauyz_y
                  - utauxz_z - vtauyz_z - wtauzz_z )
```

As desired, all of these calculations are ignorant of Section 4 save for requiring the model constants and that the solution is given in terms of ρ, u, v, w, and T. Given sufficiently smooth, exact solution information it is now possible to produce the exact manufactured forcing required to enforce it.

Computing both solution information and forcing using SymPy can be slow. We propose translating the expressions into an imperative programming language and *evaluating them using floating point operations at runtime*. The errors arising from doing so behave like standard floating point truncations [4]. Notice that the current practice of using a computer algebra system to generate thousands of exact, irreducible trigonometric terms only to then evaluate the result in floating point suffers from the same collection of floating point truncation concerns. Sections 6 and 8 will comment further on the matter.

Conceptually, this process is nothing but hierarchic MMS where all forcing terms have been recursively decomposed into small, human-decipherable, solution-agnostic building blocks. However, unlike traditional MMS and hierarchic MMS as presented by Malaya et al. [8], this approach regains the ability to manually check the forcing with minimal effort.

6. REFERENCE IMPLEMENTATION

A C++ implementation for evaluating our manufactured solution is distributed with MASA [9]. The MASA library provides a suite of manufactured solutions for the verification of partial differential equation solvers [8]. Through MASA, one can also use the reference implementation directly in C- and Fortran-based codes without a working knowledge of C++. For C++-savvy users, templates permit computing the forcing at any desired floating point precision. Templates also permit substituting arbitrary solution forms (which must possess adequate smoothness) into the solution-agnostic manufactured forcing. A more complete package, including solution visualization capabilities to assist in coefficient selection, is available by contacting the corresponding author.

The complete package contains high precision tests for ensuring the reference routines compute what this paper's listings describe. The tests use the SymPy listings which generate this document to compute "exact" values in 40-digit arithmetic. The C++ floating point implementation matches those values to within a relative error of 4.9 times machine epsilon ϵ when working in double precision. For comparison, the regression tests in MASA, which are used to verify floating point computations from the large quantities of code generated by computer algebra systems, require relative errors of no more than 4.5ϵ. Working in our test platform's 12-byte `long double` type reduced relative error to less than double precision ϵ.

Because of our automated paper/implementation consistency tests, a reader concerned with the correctness of the reference implementation need only check the basic calculus performed by SymPy in Section 4, check the simple algebraic expansions from the forcing listing in Section 5, and then check Table 1 to be certain the reference implementation source files match exactly what this paper documents.

Table 1. Checksums for the solution, forcing, and reference implementation files.

Description	Filename	MD5 checksum
SymPy solution form	`soln.py`	`84749af63c5de16ebf2b24a46231f326`
SymPy forcing	`forcing.py`	`5034166058fbe31c6193810e4b141785`
C++ declarations	`nsctpl_fwd.hpp`	`27d0766f99d8d84154e12bde8484569c`
C++ implementation	`nsctpl.hpp`	`61931064049455715db7d0d46d5c2cb7`

7. TEST COEFFICIENTS FOR ISOTHERMAL CHANNELS AND FLAT PLATES

Employing the manufactured solution requires fixing the more than two hundred coefficients appearing in Equations (1) and (2). Selecting coefficients giving physically realizable fields (i.e. satisfying $\rho > 0$ and $T > 0$ everywhere in space over some time duration) is not difficult but it is time consuming. The C++ reference implementation can be used to expedite the selection process. Here we present reasonable coefficient choices for testing flow solvers on the channel and flat plate geometries pictured in Figure 1.

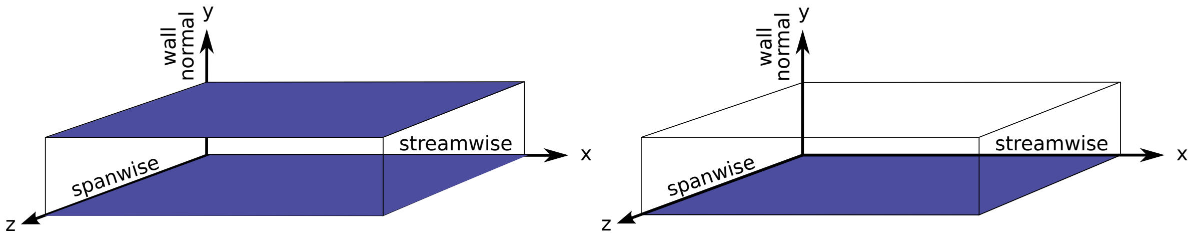

Figure 1. The channel (left) and flat plate (right) geometries.

In both geometries the streamwise, wall-normal, and spanwise directions are labeled x, y, and z respectively. Both x and z are periodic while $y \in \{0, L_y\}$ is not. Transient tests should likely take place within the duration $0 \leq t \leq 1/10$ seconds as the time phase offsets (e.g. g_{Tyz}) have been chosen for appreciable transients to occur throughout this time window.

For isothermal channel flow code verification we recommend testing using

$$b_{\rho y} = b_{uy} = b_{vy} = b_{wy} = b_{Ty} = \frac{1}{2}$$

and the coefficients given in Tables 2 and 3. With these choices the manufactured solution satisfies isothermal, no-slip conditions at $y = 0, L_y$. The resulting density and pressure fields are depicted in Figure 2. For isothermal flat plate code verification we suggest using

$$b_{\rho y} = b_{uy} = b_{vy} = b_{wy} = b_{Ty} = \frac{1}{4}$$

and the coefficients given in the same tables. With these choices the manufactured solution satisfies an isothermal, no-slip condition at $y = 0$. Both coefficients collections are easily accessible from the reference implementation.

8. EXPERIENCES VERIFYING A PSEUDO-SPECTRAL COMPRESSIBLE CODE

We used the solution reference implementation with the suggested isothermal channel coefficients to verify a new pseudo-spectral Navier–Stokes flow solver called “Suzerain”. Suzerain uses a mixed Fourier/B-spline spatial discretization [1, 7, 5] and a hybrid implicit/explicit temporal scheme [21]. The B-spline piecewise polynomial order can be set as high as desired. The temporal scheme is globally second-order on linear implicit terms and globally third-order on nonlinear explicit terms. The code has been developed as part of the first author’s thesis research.

An initial attempt to verify Suzerain against the fully three-dimensional, transient solution immediately encountered a bug. The bug, and all others mentioned in this section,

Table 2. Constant recommendations from Section 7. Standard MKS units are implied with each value; e.g. R is given in $\mathrm{J\,kg^{-1}\,K^{-1}}$ and μ_r is given in $\mathrm{Pa \cdot s}$.

Constant	Value	Constant	Value	Constant	Value
γ	1.4	T_r	300	R	287
Pr	0.7	β	$\frac{2}{3}$	κ_r	$\frac{\gamma R \mu_r}{(\gamma-1)\mathrm{Pr}}$
μ_r	1.852×10^{-5}	λ_r	$-\frac{2}{3}\mu_r$		

Table 3. Parameter recommendations from Section 7. Unlisted values should be set to zero.

Param.	Value	Param.	Value	Param.	Value	Param.	Value	Param.	Value
L_x	4π	a_{uxy}	$\frac{53}{37}$	a_{vxy}	3	a_{wxy}	11	a_{T0}	300
L_y	2	b_{uxy}	3	b_{vxy}	3	b_{wxy}	3	a_{Txy}	$\frac{300}{17}$
L_z	$\frac{4\pi}{3}$	c_{uxy}	$-\frac{\pi}{2}$	c_{vxy}	$-\frac{\pi}{2}$	c_{wxy}	$-\frac{\pi}{2}$	b_{Txy}	3
$a_{\rho 0}$	1	d_{uxy}	3	d_{vxy}	3	d_{wxy}	3	c_{Txy}	$-\frac{\pi}{2}$
$a_{\rho xy}$	$\frac{1}{11}$	e_{uxy}	$-\frac{\pi}{2}$	e_{vxy}	$-\frac{\pi}{2}$	e_{wxy}	$-\frac{\pi}{2}$	d_{Txy}	3
$b_{\rho xy}$	3	f_{uxy}	3	f_{vxy}	3	f_{wxy}	3	e_{Txy}	$-\frac{\pi}{2}$
$d_{\rho xy}$	3	g_{uxy}	$\frac{\pi}{4}$	g_{vxy}	$\frac{\pi}{4}$	g_{wxy}	$\frac{\pi}{4}$	f_{Txy}	3
$f_{\rho xy}$	3	a_{uy}	53	a_{vy}	2	a_{wy}	7	g_{Txy}	$\frac{\pi}{4}$
$g_{\rho xy}$	$\frac{\pi}{4}$	b_{uy}	see §7	b_{vy}	see §7	b_{wy}	see §7	a_{Ty}	$\frac{300}{13}$
$a_{\rho y}$	$\frac{1}{7}$	c_{uy}	$-\frac{\pi}{2}$	c_{vy}	$-\frac{\pi}{2}$	c_{wy}	$-\frac{\pi}{2}$	b_{Ty}	see §7
$b_{\rho y}$	see §7	f_{uy}	1	f_{vy}	1	f_{wy}	1	c_{Ty}	$-\frac{\pi}{2}$
$f_{\rho y}$	1	g_{uy}	$\frac{\pi}{4} - \frac{1}{20}$	g_{vy}	$\frac{\pi}{4} - \frac{1}{20}$	g_{wy}	$\frac{\pi}{4} - \frac{1}{20}$	f_{Ty}	1
$g_{\rho y}$	$\frac{\pi}{4} - \frac{1}{20}$	a_{uyz}	$\frac{53}{41}$	a_{vyz}	5	a_{wyz}	13	g_{Ty}	$\frac{\pi}{4} - \frac{1}{20}$
$a_{\rho yz}$	$\frac{1}{31}$	b_{uyz}	2	b_{vyz}	2	b_{wyz}	2	a_{Tyz}	$\frac{300}{37}$
$b_{\rho yz}$	2	c_{uyz}	$-\frac{\pi}{2}$	c_{vyz}	$-\frac{\pi}{2}$	c_{wyz}	$-\frac{\pi}{2}$	b_{Tyz}	2
$d_{\rho yz}$	2	d_{uyz}	2	d_{vyz}	2	d_{wyz}	2	c_{Tyz}	$-\frac{\pi}{2}$
$f_{\rho yz}$	2	e_{uyz}	$-\frac{\pi}{2}$	e_{vyz}	$-\frac{\pi}{2}$	e_{wyz}	$-\frac{\pi}{2}$	d_{Tyz}	2
$g_{\rho yz}$	$\frac{\pi}{4} + \frac{1}{20}$	f_{uyz}	2	f_{vyz}	2	f_{wyz}	2	e_{Tyz}	$-\frac{\pi}{2}$
		g_{uyz}	$\frac{\pi}{4} + \frac{1}{20}$	g_{vyz}	$\frac{\pi}{4} + \frac{1}{20}$	g_{wyz}	$\frac{\pi}{4} + \frac{1}{20}$	f_{Tyz}	2
								g_{Tyz}	$\frac{\pi}{4} + \frac{1}{20}$

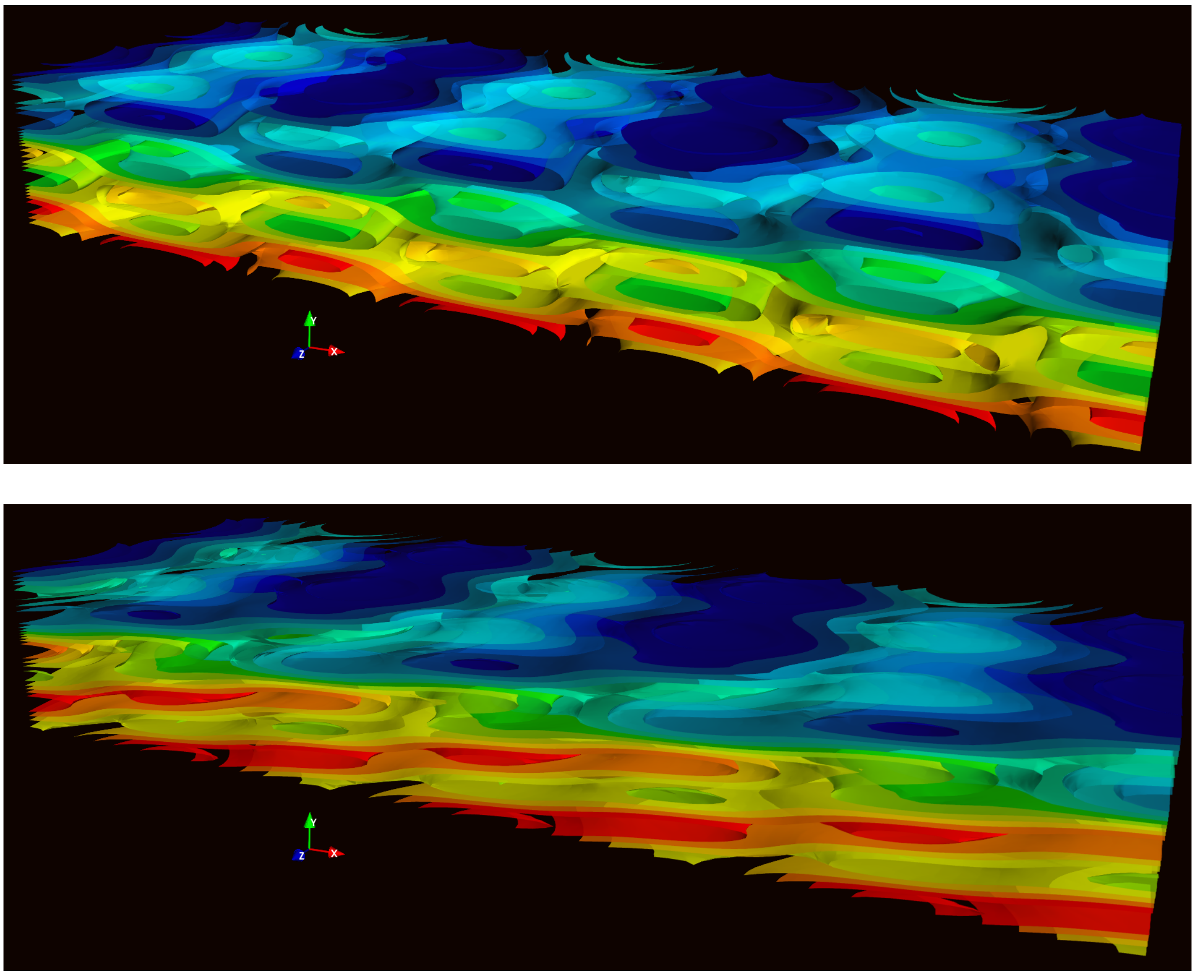

Figure 2. Isocontours of the density (above) and pressure (below) from the suggested isothermal channel test problem. The density is fixed by the solution (2) while the pressure results from substituting the density and temperature solutions into the model (1).

manifested itself as a failure to converge to zero error at the rate appropriate for the chosen numerics. This bug was isolated by “turning off” all x, z, and t-related partial derivatives using zero coefficients in the solution (2). We verified the steady, y-only solver behavior to be correct and then “turned on” the remaining two spatial directions. This led to the discovery and correction of a typo in the continuity equation implementation. While the steady-state behavior of the code was now correct, the transient behavior did not converge at the expected rate of the time discretization scheme. We then identified a formulation issue stemming from incorrectly accounting for fast Fourier transform normalization constants. The issue caused the fields to evolve too slowly in time by a constant factor related to the Fourier transform sizes. Correcting this final issue, we became confident that Suzerain was solving the desired equations correctly because the code produced convergence rates matching the numerics’ formal order of accuracy.

Convergence rates were assessed using the three-sample observed order of accuracy technique detailed by Roy [18] which he references from Roache [16]. To review, assuming an approximation $A(h)$ shows an h-dependent truncation error compared to an exact value A, *viz.* $A - A(h) = a_0 h^{k_0} + a_1 h^{k_1} + \cdots$, gives rise to the classical Richardson extrapolation procedure. Neglecting $O(h^{k_1})$ contributions, one can estimate the leading error order k^0 by numerically solving

$$A = \frac{t^{k_0} A\left(\frac{h}{t}\right) - A(h)}{t^{k_0} - 1} + O(h^{k_1}) = \frac{s^{k_0} A\left(\frac{h}{s}\right) - A(h)}{s^{k_0} - 1} + O(h^{k_1}) \tag{3}$$

given three approximations $A(h)$, $A(h/s)$, and $A(h/t)$ to A. In our case, $A(h)$ was the maximum "coefficient-wise" absolute error taken over all errors present in the real and imaginary parts of Fourier/B-spline coefficients when compared against the exact solution projected onto the same grid. This admittedly odd error metric came from using `h5diff`, a "real-only" utility distributed with the HDF5 library [25], to compute differences between simulation restart files storing complex-valued expansion coefficients. By using the MMS, it is known *a priori* that A should be zero which allows assessing the impact of neglecting higher order terms when computing the three-sample estimate of k_0.

An example of the observed convergence for a time-invariant subset of the solution is shown on the left side of Figure 3. Below 4×10^5 degrees of freedom (DOF) per field (i.e. the number of Fourier/B-spline expansion coefficients employed for each of ρ, ρu, etc.), the coefficient-wise error reduction is consistent with the piecewise septic B-spline basis used in the y direction for this steady computation. Above 4×10^5 DOF, floating point truncation errors in the manufactured forcing of roughly 45 times machine epsilon prevent ρ from converging further. This stall becomes slightly visible in ρe above 1.3×10^6 DOF due to coupling between the continuity and total energy equations. Another forcing-limited convergence stall appears in ρu at 3×10^6 DOF due to $Q_{\rho u}$ error.

Convergence on the full, transient solution is depicted in the right side of Figure 3. Below 5×10^4 DOF per field, convergence approaches the piecewise quartic B-spline order used in this unsteady calculation. The temporal error has not yet come into play because such coarse grids can compute the selected duration in a single time step. Above 5×10^4 DOF, multiple time steps are required and we observe fourth-order coefficient-wise convergence consistent with Suzerain's hybrid timestepping scheme when the nonlinear, globally third-order behavior dominates. Notice that, since local error in coefficient-space is shown, one observes spatial and temporal rates one order higher than the selected numerics' global orders.

The solution flexibility from Equation (2) greatly aided our verification efforts because we could isolate specific aspects of the solution merely by changing configuration parameters (e.g. temporal dependence, dependence on particular spatial derivatives). The one-to-one correspondence between our C++ reference implementation and the presented SymPy forcing permitted investigating individual terms during the usual edit-compile-run process by simply commenting or uncommenting their contributions. Having available only a large, monolithic symbolic expression or even a relatively granular hierarchic MMS would have made debugging more difficult.

We caution that manufactured-solution-based verification is a necessary but not sufficient condition for code correctness. Our confidence in Suzerain's correctness also relies upon a large, automated test suite including more than just such convergence tests.

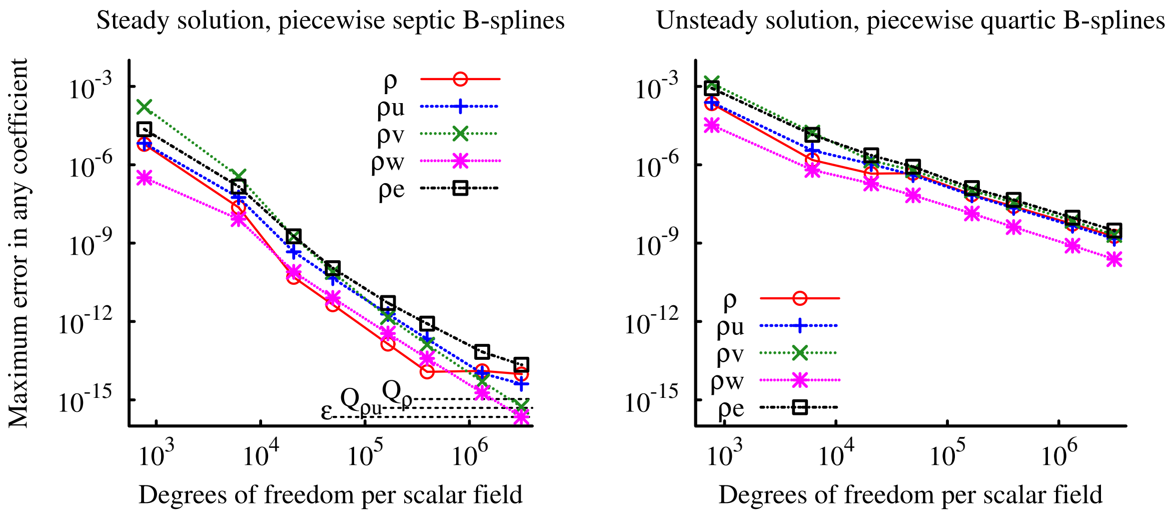

Figure 3. Field-by-field convergence for Suzerain on a steady (left) and transient (right) problem at two different B-spline orders. Labels Q_ρ and $Q_{\rho u}$ show measured relative error in the associated floating point manufactured forcing computations. Label ϵ shows machine epsilon.

9. POTENTIAL REUSE AND EXTENSION OF THE MANUFACTURED SOLUTION

Because our approach decouples the manufactured forcing computations from the manufactured solution form, this solution and its reference implementation could be easily reused and extended in a number of ways. As alluded to in the previous section, by specifying zero coefficients for terms causing particular partial derivatives, it is trivial to reduce the solution to either a lower dimensionality, to steady state, or to both. We obtained periodic, isothermal problems with no-slip walls through careful coefficient selection. Other boundary conditions can be built with the same solution form. One could, for example, change the y-related coefficients to create a fully periodic test problem suitable for a homogeneous, isotropic code. Different density or pressure conditions could be set at the walls. However, pressure boundary conditions would require adopting pressure instead of temperature as a specified solution followed by modifying a small number of constitutive relation computations.

Beyond geometry changes or boundary condition modifications, one could adjust the solution forms to better represent expected physics. Physically-inspired MMS behavior has appeared in the context of Favre-averaged modeling [13]. It would be possible to make the present solution's near-wall velocity behavior consistent with that expected in a wall-bounded flow [14] by adjusting the solution implementation for u, v, and w but without modifying the forcing computations. Small formulation changes are also feasible. One could employ the solution in a code using a Sutherland's law viscosity [23] by changing only the expressions for `mu`, `mu_x`, `mu_y`, and `mu_z` within Section 5 and fixing the reference Sutherland temperature. The solution form would remain identical. The reference implementation would similarly require only changes to how μ and its spatial derivatives are computed.

Finally, extending our hybrid symbolic manipulation/floating point MMS approach to basic multi-species flows should be both straightforward and yield human-verifiable results.

Multi-species formulations are necessary for solving mixing problems and investigating hydrodynamic instabilities. These formulations, e.g. as described by Cook [2], add one-or-more species concentration evolution equations and obtain local thermodynamic properties through a weighted mixture of local species contributions. Pushing a symbolic solution through such mixture relations and their associated energy terms would exacerbate the symbolic term explosion inherent in the traditional MMS. Though floating-point-related convergence stalls have appeared in the late stages of driving towards machine epsilon, we suspect our hybrid MMS can successfully verify codes employing multi-species formulations when forcing is computed in double precision. If not, higher precision floating point is available. In either case, truncation-monitoring approaches, e.g. interval arithmetic, could be brought to bear quickly using C++ templates.

10. CONCLUSIONS

We have constructed a new manufactured solution for the compressible Navier–Stokes equations for a perfect gas with a power-law viscosity. We have made our C++ reference implementation easily accessible to the computational fluid dynamics community by releasing it through the MASA library [9]. This manufactured solution was invaluable for debugging and verifying our pseudo-spectral compressible flow solver. Finally, we provided many examples of how this manufactured solution easily might be adapted for other use cases and extended for other formulations.

Acknowledgements

This material is based in part upon work supported by the United States Department of Energy [National Nuclear Security Administration] under Award Number [DE-FC52-08NA28615].

REFERENCES

[1] John P. Boyd, *Chebyshev and Fourier spectral methods*, Dover, 2001.

[2] A. W. Cook, *Enthalpy diffusion in multicomponent flows*, Physics of Fluids **21** (2009), no. 5, 055109.

[3] L. Eça, M. Hoekstra, A. Hay, and D. Pelletier, *Verification of RANS solvers with manufactured solutions*, Engineering with Computers **23** (2007), no. 4, 253–270.

[4] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comput. Surv. **23** (1991), no. 1, 5–48.

[5] S. E. Guarini, R. D. Moser, K. Shariff, and A. Wray, *Direct numerical simulation of a supersonic turbulent boundary layer at Mach 2.5*, Journal of Fluid Mechanics **414** (2000), 1–33.

[6] Patrick M. Knupp and Kambiz Salari, *Verification of computer codes in computational science and engineering*, Discrete Mathematics and its Applications, Chapman & Hall/CRC, 2003.

[7] W. Y. Kwok, R. D. Moser, and J. Jiménez, *A critical evaluation of the resolution properties of B-spline and compact finite difference methods*, Journal of Computational Physics **174** (2001), no. 2, 510–551.

[8] N. Malaya, K. C. Estacio-Hiroms, R. H. Stogner, K. W. Schulz, P. T. Bauman, and G. F. Carey, *MASA: A library for verification using manufactured and analytical solutions*, submitted to Engineering with Computers, 2012.

[9] MASA Development Team, *MASA: A Library for Verification Using Manufactured and Analytical Solutions*, `https://red.ices.utexas.edu/projects/software/wiki/MASA`, 2011.

[10] R. G. McClarren and R. B. Lowrie, *Manufactured solutions for the P-1 radiation-hydrodynamics equations*, Journal of Quantitative Spectroscopy & Radiative Transfer **109** (2008), no. 15, 2590–2602.

[11] William L. Oberkampf and Christopher J. Roy, *Verification and validation in scientific computing*, Cambridge University Press, 2010.

[12] J. T. Oden, T. Belytschko, J. Fish, T. J. R. Hughes, C. Johnson, D. Keyes, A. Laub, L. Petzold, D. Srolovitz, and S. Yip, *Revolutionizing engineering science through simulation*, Tech. report, National Science Foundation Blue Ribbon Panel on Simulation-Based Engineering Science (SBES), 2006.

[13] T. A. Oliver, K. C. Estacio-Hiroms, N. Malaya, and G. F. Carey, *Manufactured solutions for the Favre-averaged Navier–Stokes equations with eddy-viscosity turbulence models*, 50th AIAA Aerospace Sciences Meeting, no. AIAA 2012-0080, 2012.

[14] Stephen B. Pope, *Turbulent flows*, Cambridge University Press, October 2000.

[15] P. J. Roache and S. Steinberg, *Symbolic manipulation and computational fluid-dynamics*, AIAA Journal **22** (1984), no. 10, 1390–1394.

[16] Patrick J. Roache, *Verification and validation in computational science and engineering*, Hermosa Publishers, 1998.

[17] ______, *Code verification by the method of manufactured solutions*, Journal of Fluids Engineering **124** (2002), no. 1, 4–10.

[18] C. J. Roy, *Review of code and solution verification procedures for computational simulation*, Journal of Computational Physics **205** (2005), no. 1, 131–156.

[19] C. J. Roy, C. C. Nelson, T. M. Smith, and C. C. Ober, *Verification of Euler/Navier–Stokes codes using the method of manufactured solutions*, International Journal for Numerical Methods in Fluids **44** (2004), no. 6, 599–620.

[20] H. G. Silva, L. F. Souza, and M. A. F. Medeiros, *Verification of a mixed high-order accurate DNS code for laminar turbulent transition by the method of manufactured solutions*, International Journal for Numerical Methods in Fluids **64** (2010), no. 3, 336–354.

[21] P. R. Spalart, R. D. Moser, and M. M. Rogers, *Spectral methods for the Navier–Stokes equations with one infinite and two periodic directions*, Journal of Computational Physics **96** (1991), no. 2, 297–324.

[22] S. Steinberg and P. J. Roache, *Symbolic manipulation and computational fluid dynamics*, Journal of Computational Physics **57** (1985), no. 2, 251–284.

[23] W. Sutherland, *The viscosity of gases and molecular force*, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **36** (1893), no. 223, 507–531.

[24] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, `http://www.sympy.org`, 2011.

[25] The HDF Group, *Hierarchical data format version 5*, `http://www.hdfgroup.org/HDF5`, 2000–2010.

[26] D. Tremblay, S. Etienne, and D. Pelletier D., *Code verification and the method of manufactured solutions for fluid-structure interaction problems*, 36th AIAA Fluid Dynamics Confernce, no. AIAA 2006-3218, 2006, pp. 882–892.

[27] J. M. Vedovoto, A. Silveira Neto, A. Mura, and L. F. F. Silva, *Application of the method of manufactured solutions to the verification of a pressure-based finite-volume numerical scheme*, Computers & Fluids **51** (2011), 85–99.

[28] S. Viswanathan, H. Wang, and S. B. Pope, *Numerical implementation of mixing and molecular transport in LES/PDF studies of turbulent reacting flows*, Journal of Computational Physics **230** (2011), 6916–6957.