SANDIA REPORT
SAND2020-XXXX
Printed September 2020

Sandia National Laboratories

An Overview of Gemma FY20 Verification Activities

Aaron Krueger, Jack Hamel, Neil Matula, Brian Freno

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

ABSTRACT

Gemma verification activities for FY20 can be divided into three categories: the development of specialized quadrature rules, initial progress towards the development of manufactured solutions for code verification, and automated code-verification testing.

In the method-of-moments implementation of the electric-field integral equation, the presence of a Green's function in the four-dimensional integrals yields singularities in the integrand when two elements are nearby. To address these challenges, we have developed quadrature rules to integrate the functions through which the singularities can be characterized.

Code verification is necessary to develop confidence in the implementation of the numerical methods in Gemma. Therefore, we have begun investigating the use of manufactured solutions to more thoroughly verify Gemma. Manufactured solutions provide greater flexibility for testing aspects of the code; however, the aforementioned singularities provide challenges, and existing work is limited in rigor and quantity.

Finally, we have implemented automated code-verification testing using the VVTest framework to automate the mesh refinement and execution of a Gemma simulation to generate mesh convergence data. This infrastructure computes the observed order of accuracy from these data and compares it with the theoretical order of accuracy to either develop confidence in the implementation of the numerical methods or detect coding errors.

CONTENTS

1. INTRODUCTION

Gemma is a boundary element method code that solves the method-of-moments implementations of the electric-field integral equation (EFIE), the magnetic-field integral equation (MFIE), and the combined-field integral equation (CFIE). To improve the numerical methods and verify their implementation in Gemma, we have undertaken multiple activities in FY20. These activities included the development of more suitable quadrature rules to integrate some of the singular integrands that arise in Gemma, as well as more traditional code-verification activities.

In the method-of-moments implementation of the EFIE, the presence of a Green's function in the four-dimensional integrals yields singularities in the integrand when two elements are nearby. To address these challenges, we published a journal article describing how to compute symmetric triangle quadrature rules that can integrate arbitrary function sequences [1], and we characterized the singularities in terms of function sequences [2].

To verify the implementation of the numerical methods in Gemma, we undertook multiple code-verification activities. When numerically solving the underlying equations, the equations must be discretized. Due to the finite nature of the discretization, the equations incur a truncation error and, consequently, their solutions introduce a discretization error. As the discretization is refined, the discretization error should decrease. More rigorously, the code should achieve an expected order of accuracy: as the discretization is refined by a factor, the error should decrease at a rate that is an expected power of that factor, provided the discretization is in the asymptotic region. In practice, since the exact solution is generally unavailable, manufactured solutions are frequently employed [3].

Code verification has been performed on computational physics codes associated with several physics disciplines, including fluid dynamics [4, 5, 6, 7, 8], solid mechanics [9], fluid–structure interaction [10], heat transfer in fluid–solid interaction [11], multiphase flows [12], radiation hydrodynamics [13], and electrodynamics [14]. However, very little work has been done on verification for integral equations, due to the challenges of analytically integrating arbitrary functions, especially when they are singular.

Nonetheless, whether using manufactured or exact solutions to perform code verification, the process can be facilitated through automation to run numerous tests frequently. Therefore, we have developed an automated testing framework to verify the numerical methods. This framework employs VVTtest and additional postprocessing tools to automate test execution and the determination of whether the tests pass or fail.

This report is organized as follows. In Chapter 2, we describe the quadrature work. In Chapter 3, we discuss the importance of manufactured solutions in code verification, as well as the challenges of implementing them in a code like Gemma, and our ongoing work. In Chapter 4, we describe the automated verification testing infrastructure in detail. Finally, in Chapter 5, we provide an outlook for our future work.

2. GEOMETRICALLY SYMMETRIC QUADRATURE RULES FOR THE EFIE

The method of moments is generally employed to solve the EFIE, MFIE, and CFIE, upon discretizing surfaces using planar or curvilinear mesh elements. Through this method, the integration over source and test elements leads to the evaluation of four-dimensional integrals. However, the presence of a Green's function in these equations yields scalar and vector potential terms with singularities (in their higher-order derivatives) when the test and source elements share one or more edges or vertices and near-singularities when they are otherwise close.

Many approaches have been developed to address the singularity and near-singularity for the inner, source-element integral; fewer approaches have been developed to address the singularity in the outer, test-element integral. We have developed geometrically symmetric quadrature rules better suited for evaluating the logarithmic singularities in the test integral [1]. Symmetric rules that can efficiently handle singularities are desirable because their mapping to the integration domain is straightforward and points are not heavily concentrated near some vertices. Asymmetric rules, on the other hand, which are generally employed to integrate singularities, require the the determination of vertex mapping, and points may be concentrated nonuniformly at the vertices.

We have demonstrated the effectiveness of these rules for several examples encountered in both the scalar and vector potentials of the EFIE (singular, near-singular, and far interactions), and we have compared their performance to existing rules [2]. These rules exhibit better convergence properties than quadrature rules for polynomials and, in general, lead to better accuracy with a lower number of quadrature points.

3. MANUFACTURED SOLUTIONS

In general, codes that approximately solve systems of differential, integral, or integro-differential equations can only be verified by using them to solve problems with known solutions [3]. The discretization of the governing equations (e.g., finite differences, volumes, or elements) will necessarily incur some truncation error, and thus the approximate solutions produced from the discretized equations will incur some associated discretization error. If the solution to the problem is known, a measure of the discretization error (typically a discrete norm thereof) may be evaluated directly from the approximate solution. In the most basic sense of verification, if the discretization error tends to zero as the discretization is refined, the consistency of the code is verified [15]. This may be taken a step further by examining not only consistency, but the rate at which the error decreases as the discretization is refined, thereby verifying the order of accuracy of the discretization scheme. The correctness of the numerical-method implementation may then be verified by comparing the expected and observed orders of accuracy obtained from numerous test cases with known solutions.

Unfortunately, exact solutions to systems of engineering interest are rare, and those that do exist often require dramatic simplifications to both the domain geometry and the equations themselves in order to obtain a tractable problem. Individually, these simple problems do not exercise enough of the code to constitute convincing evidence of its correctness. However, a body of evidence supporting the correctness of the code may be constructed by amassing a large suite of test cases that collectively cover all terms of the governing equations, all possible boundary conditions, and a wide range of domain geometries [16]. The tests documented in Chapter 4 are a significant step toward such a body of evidence. Nevertheless, the most convincing code-verification tests come from more complex cases that simultaneously exercise many terms of the governing equations on nontrivial domain geometries, and exact solutions for problems with this level of complexity are scarce.

The method of manufactured solutions (MMS) is a general technique for constructing problems of arbitrary complexity with known solutions. One begins this process in reverse by manufacturing the desired solution. In principle, this manufactured solution (MS) may be any function, but several properties are desirable [16]:

1. The MS should be consist of combinations of elementary functions, such as polynomial, trigonometric, and exponential functions. This not only simplifies derivations and implementation, but ensures that the MS (and its derivatives) will be representable to sufficient precision within the tested code. Series representations are undesirable, as they create an additional concern regarding whether they have been carried to a sufficient number of terms to obtain the required precision [3].

2. The MS should be sufficiently smooth, such that the error incurred by the discretization is small on relatively coarse meshes. This ensures that the order of accuracy may be estimated with minimal computational expense.

3. The MS should be general enough that all terms of the governing equations are exercised.

4. The MS should have a sufficient number of nontrivial derivatives, such that the expected order of accuracy of the discretization can be observed. In the most ideal case, the solution will have an infinite number of nontrivial derivatives.

5. Since the robustness of the code is not the primary concern, the MS should not have any features that inhibit the solution of the discretized equations.

Once a solution is manufactured, it is substituted directly into the governing equations. If the MS is composed of elementary functions (as suggested in Item 1 of the above list), each term of the governing equations may easily be evaluated exactly, either by hand or using symbolic manipulation software. In general, the MS is not expected to satisfy the governing equations. Instead, a residual term will appear, which quantifies the deviation from the satisfaction of the equations. Since the MS is a prescribed function of space and time, the residual term will, likewise, be a known function of space and time. If this residual is added to the governing equations as a source term, the resulting modified equations will be exactly satisfied by the MS. Concerns are immediately raised regarding uniqueness of the solution to the manufactured problem, but this has rarely been found to cause difficulties in practice [3]. The result of this process is a problem, of arbitrary complexity, for which an exact solution is known. At this point, it should be noted that we have not restricted ourselves to any particular domain geometry or boundary conditions. An arbitrary domain boundary may be selected, with portions allocated for each type of boundary condition we wish to test. The selected boundary conditions are modified to support the MS in the same manner as the governing equations; that is, by substituting in the MS to obtain a residual term. Finally, the code to be tested is modified to support the additional source term and boundary conditions. The code may then be verified by comparing the simulation result for the manufactured problem against the known solution. Several benefits of this framework bear repeating:

1. Exact solutions may be created for problems that simultaneously exercise every term of the governing equations, thus providing more convincing evidence of code correctness than several simpler tests that collectively exercise those same terms.

2. MMS is purely a mathematical process, i.e., the physics of the problem need not be considered. This enables the user to avoid difficulties that would normally complicate the solution, such as the singularities that would commonly appear at reentrant corners [17, p. 76].

3. Because the domain boundary choice is decoupled from the solution, the user has full control over the complexity of the geometry to be considered. In practice, a range of boundary complexities is employed in a test suite created with MMS. In some test cases, unrealistically complicated geometries can be selected that fully stress the boundary treatments of the code. In other cases, simple geometries that can be exactly represented

with the mesh elements supported by the tested code can be used to avoid the faceting errors that commonly complicate grid convergence studies.

In the form outlined above, MMS has been used extensively in the verification of software for the solution of systems of differential equations [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19]. However, the literature contains few instances of MMS being used in the verification of software for integral equations. This is due to the simple fact that, while analytical differentiation is a straightforward exercise, analytical integration is not always possible. Hence, the residual source term arising from the manufactured solution may not be representable in closed form, and its implementation may be accompanied by numerical techniques that carry their own uncertainties. Furthermore, in many applications, such as the boundary element method (BEM) employed by Gemma, singular integrals appear, which can further complicate the numerical evaluation of the source term. Hence, much of the elegance, simplicity, and instilled confidence of MMS is lost when applied to integral equations in a straightforward manner, and as a result, effective implementation of MMS in the context of BEM codes is an open subject of research. The most substantial effort thus far in the computational electromagnetics (CEM) community has been the work of Marchand [20], in which the author calculated the MMS source terms for the EFIE using numerical techniques. While the quadrature error can be driven low enough that grid-independence studies become feasible, the presence of this additional error often places a lower bound on the discretization error that can be obtained, and therefore limits the scope of the grid convergence study, in addition to being undesirable for the aforementioned reasons.

In FY20, the authors of this report have pursued an implementation of MMS for Gemma that avoids the complications discussed above. Two options have been proposed and pursued in parallel:

1. Manufactured solutions composed of sums of point sources enable direct analytical evaluation of all terms required by Gemma without relying on numerical integration.

2. Conventional manufactured solutions (with certain restrictions) may be carefully supplied to Maxwell's equations and associated boundary conditions to analytically build the incident field and required source terms.

A simple code for solving the EFIE on a rectangular plate was developed in MATLAB as a testbed for these ideas. If successful, these proposed methods will represent a significant step forward in the verification of CEM codes using boundary elements by removing reliance on numerical integration to implement the manufactured solution, thereby better demonstrating the credibility of the software.

4. AUTOMATED CODE-VERIFICATION TESTING

To automate the verification tests for Gemma, the VVTest infrastructure is employed with a custom postprocessing script, which determines whether the tests pass or fail. This chapter describes the VVTest infrastructure and the pass/fail criteria, with examples.

4.1. VVTEST

To perform code-verification tests, Gemma employs VVTest, which has been used to perform code-verification testing for other Sandia codes. VVTest is a simulation manager that executes multiple simulations either sequentially, or in parallel. For a given simulation, VVTest executes the simulation for differently sized discretizations to assess how the discretization impacts the solution. The solutions from the different discretizations are compared with an exact solution to compute their error. This error can be a quantity-of-interest (QoI) error, or a norm of the solution error. As the discretization is refined, the discretization error should decrease. More rigorously, the code should achieve an expected order of accuracy or rate of convergence: as the mesh is refined by a factor, the error should decrease at a rate that is an expected power of that factor, provided the mesh is in the asymptotic region. Comparing the theoretical convergence rate to the observed convergence rate helps determine whether the numerical scheme is implemented correctly. Additional information about VVTest can be found at `https://gitlab.sandia.gov/rrdrake/scidev/-/wikis/home`. Information about the Gemma-specific VVTest implementation can be found at at `https://cee-gitlab.sandia.gov/EMR_codes/gemma/-/wikis/vvtest`.

4.2. POSTPROCESSING TOOLS

The simulation data generated by VVTest are postprocessed using additional tools, which compute the errors and determine whether the test passes or fails.

4.2.1. Error

The Gemma implementation of VVTest employs a variety of options for computing errors via various functions, which are documented here. These functions are contained in `gemma/gemma_vvtest_config/gemma/error_tools.py`.

Discrete L^p-Norms

The first function is `computeDiscreteNorms(p, A)`, which computes a discrete norm. `p` indicates the L^p-norm (e.g., $p = 1$, $p = 2$), and `A` is an $M \times N$ array of M row vectors $\mathbf{a}_i \in \mathbb{R}^N$, for $i = 1, \ldots, M$. The function returns

$$\|\mathbf{a}_i\|_p = \left(\sum_{j=1}^{N} |a_{ij}|^p \right)^{1/p} .$$

Discrete L^∞-Norms

The second function is `computeLInfinityNorms(A)`, which computes the $\mathbf{L}^\infty$-norm. `A` is an $M \times N$ array of M row vectors $\mathbf{a}_i \in \mathbb{R}^N$, for $i = 1, \ldots, M$. The function returns

$$\|\mathbf{a}_i\|_\infty = \max_j |a_{ij}|.$$

Discrete L^p Error Norms

The third function is `computeDiscreteNormError(p, A, B)`, which computes the average L^p-norm of the error between the M rows of $\mathbf{A}$ and $\mathbf{B}$. `A` and `B` are $M \times N$ arrays, and `p` indicates the L^p-norm. The function returns

$$\text{Error} = \frac{1}{M} \sum_{i=1}^{M} \left(\sum_{j=1}^{N} |a_{ij} - b_{ij}|^p \right)^{1/p} .$$

Weighted Discrete Error Norms

The fourth function is `computeWeightedNormError(p, A, B, w)`, which computes a relative, row-weighted entrywise L^p-norm of the error between $\mathbf{A}$ and $\mathbf{B}$. `A` and `B` are $M \times N$ arrays, `p` indicates the L^p-norm, and `w` is a vector of M weights. The function returns

$$\text{Error} = \left(\frac{\sum_i^M \sum_j^N w_i |a_{ij} - b_{ij}|^p}{\sum_i^M \sum_j^N w_i |a_{ij}|^p} \right)^{1/p} .$$

Discrete L^∞ Error Norms

The fifth function is `computeLInfinityNormError(A, B)`, which computes the average L^∞-norm of the error between the M rows of $\mathbf{A}$ and $\mathbf{B}$. `A` and `B` are $M \times N$ arrays. The function returns

$$\text{Error} = \frac{1}{M}\sum_{i}^{M}\max_{j}|a_{ij} - b_{ij}|.$$

4.2.2. Verification

Once the error is available, calculating the convergence rate of the error and determining whether it agrees with the theoretical rate is possible. There are three different functions within verification tools:

1. `order_of_accuracy_weighted_sets(numbers_of_unknowns_1d, errors)`
2. `pass_criteria_simple(slopes, expected_slope, tolerance_acceptable)`
3. `pass_criteria_robust_tolerance(slopes, expected_slope, tolerance_fine, tolerance_coarse)`

The following sections explain each function in detail.

Order of Accuracy

The order of accuracy function uses the error metrics computed in Section 4.2.1 to calculate the rate at which the error decreases as the discretization is refined (i.e., as the number of unknowns is increased). There are two inputs to the order of accuracy function: 1) the number of unknowns and 2) the array of the errors. The error data are then used to fit an error model, which is based on the leading term of the local truncation error (LTE) of the numerical scheme:

$$e\left(N\right) = CN^{-p},$$

where e is the error as a function of the one-dimensional number of degrees of freedom, N is the one-dimensional number of degrees of freedom, C is independent of the mesh in the asymptotic region, and p is the order of accuracy, which is to be computed. When two errors are available, p can be computed from

$$p = -\log\left(\left|\frac{e_2}{e_1}\right|\right)/\log\left(\frac{N_2}{N_1}\right).$$

When error metrics are available for two or more mesh refinement levels, a linear regression is possible to calculate the order of accuracy. However, it should be noted that the linear regression used here is unweighted, and as such, the error from the coarsest meshes may have an undue influence on the estimated order of accuracy. To mitigate this effect, VVTest generates multiple order-of-accuracy estimates, first using data from all M available meshes, then the $M-1$ finest meshes, and so on, until only data from the two finest meshes are used (for which the linear fit will be exact). This enables the user to determine whether the order of accuracy estimates are converging to the expected value. A visualization of this calculation is shown in Fig. 4-1. This slope array is then passed to one of the two functions for assessing whether the pass criteria has been met.

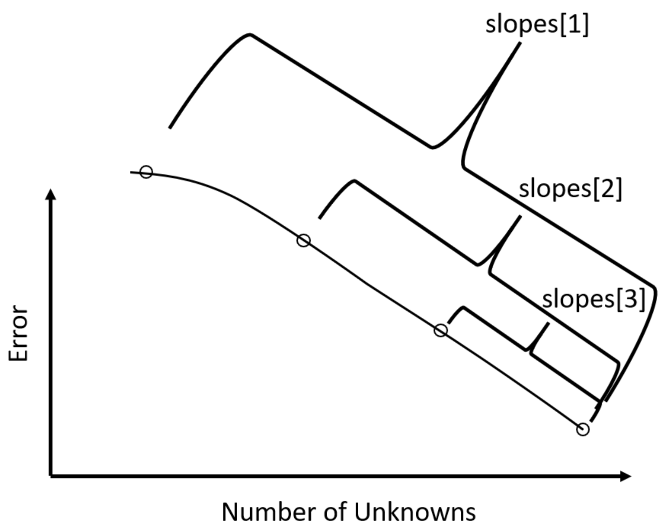

Figure 4-1. Visualization of the datasets used in the calculation of order of accuracy.

Pass Criteria

There are two different functions to determine whether Gemma passes the code-verification assessment:

1. `pass_criteria_simple(slopes, expected_slope, tolerance_acceptable)`
2. `pass_criteria_robust_tolerance(slopes, expected_slope, tolerance_fine, tolerance_coarse)`

Both of these functions check that the estimated order of accuracy matches the theoretical performance of the numerical method. The only difference is that `pass_criteria_robust_tolerance` also checks to make sure the order of accuracy is converging to the theoretical value. Below is an explanation of each function and its arguments.

`pass_criteria_simple` is the standard way to determine if a code passes the order-of-accuracy code-verification test. The first argument is `slopes`, which is a one-dimensional array containing the observed orders of accuracy calculated using the functions described in Section 4.2.2. This array is ordered with the results from the coarsest meshes first. The second argument is `expected_slope`, which is the theoretical order of accuracy. The third argument is `tolerance_acceptable`, which is a decimal value indicating the acceptable level of relative error between the observed and theoretical orders of accuracy $(0.1 = 10\%$ error$)$. If the test does not complete for any reason, it is assigned a `Fail` result. Otherwise, the slope calculated with the two finest meshes is compared to the theoretical order of accuracy. If the two agree within the prescribed tolerance, the test is assigned a `Pass` result, and if the tolerance is not met, the test is assigned a `Diff` result.

`pass_criteria_robust_tolerance` is a modification of the function above, which is intended to increase robustness. The first two arguments are `slopes` and `expected_slope`, which have the same meaning defined above. The third and fourth arguments are `tolerance_fine` and `tolerance_coarse`. These indicate the relative errors acceptable in the estimated orders of accuracy obtained from the two finest meshes and the complete set of meshes, respectively. If the test does not complete for any reason, it is assigned a `Fail` result. Otherwise, both estimates for the order of accuracy are compared against the theoretical value. If both estimates agree with the theoretical value within their respective tolerances, the test is assigned a `Pass` result, and if either or both tolerances are not met, the test is assigned a `Diff` result.

4.2.3. Plotting

We included a tool to generate plots from the results of a single test. The plots show how the errors vary with respect to the number of unknowns. A trend line is included, with the slopes annotated. For example, the top annotation is the slope of the trend line across all data points. The second annotation is the slope of the trend line across all of the data points except that from the coarsest discretization. The annotations continue this trend until only the two finest discretizations are included in the fit. Figure 4-2 provides an example of a plot generated with this tool. Over the course of FY20, the test problem size was increased up to 1.4 million unknowns. Generating the data for this plot required 8 wall-clock hours on Eclipse, utilizing 24 cores per node on 718 nodes.

4.3. TESTS AND RESULTS

To ensure the code-verification tests provide adequate coverage, we develop various tests for different portions of Gemma. The different options exercised in Gemma through the testing include different equations, QoIs, geometries, and excitation lengths. The list that follows provides details of the various convergence tests added in FY20 to Gemma's VVTest suite. The testing information is summarized in Table 4-1 and includes the tests' pass status. Each test has

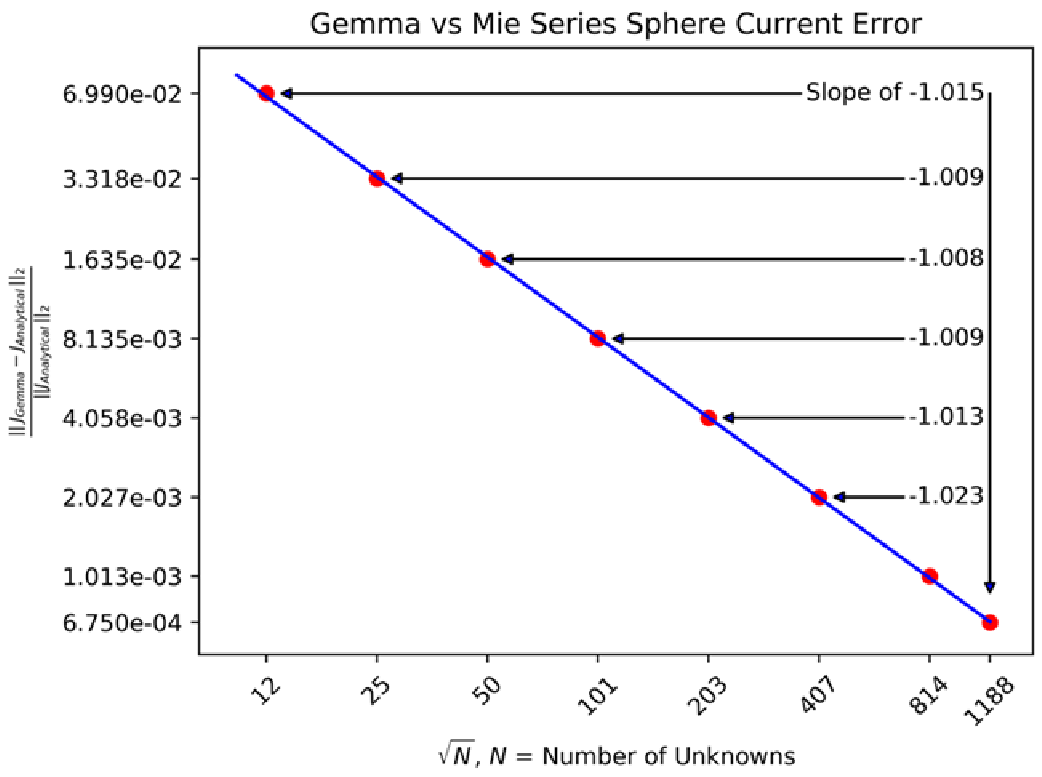

Figure 4-2. Annotated convergence plot for sphere test problem.

an analytical reference solution, which is used to compute an exact error. In Table 4-1, PMCHWT denotes the Poggio–Miller–Chang–Harrington–Wu–Tsai integral equation and Müller denotes the Müller formulation.

1. `disk_tz_pecef_pw.vvt`: This test solves the EFIE for the surface current on a flat disc. The disc is a perfect electrical conductor (PEC), with a radius of 1 m, and the excitation is a plane wave with a wavelength of 1 m and a unit incident magnetic field. The analytical solution comes from a Fortran code written by Andrew Peterson based on Flammer's Solution.

2. `sphere_field_tout_pecef_pw.vvt`: This test solves the EFIE for the electric field, evaluated at certain points away from a sphere. The sphere is a PEC with a radius of 1 m and the excitation is a plane wave with a wavelength of 10 m and a unit incident electric field. The analytical solution is obtained from a Mie series.

3. `sphere_RCS_tout_pecef_pw.vvt`: This test solves the EFIE for the radar cross-section (RCS) computed at 32 locations for the same sphere as in `sphere_field_tout_pecef_pw.vvt`.

4. `sphere_current_tout_lowdiel_pmchwt_pw.vvt`: This test solves the Poggio–Miller–Chang–Harrington–Wu–Tsai (PMCHWT) integral equation for the surface current on a dielectric sphere with $\varepsilon_r = 2$ and a radius of 1 m. The excitation is a plane

wave with a wavelength of 10 m and a unit incident electric field. The analytical solution is obtained from a Mie series.

5. `sphere_current_tout_lowdiel_muller_pw.vvt`: This tests solves the Müller formulation for the surface current on a dielectric sphere with $\varepsilon_r = 2$ and a radius of 1 m. The excitation is a plane wave with a wavelength of 10 m and a unit incident electric field. The analytical solution is obtained from a Mie series.

6. `sphere_current_tout_highdiel_pmchwt_pw.vvt`: This test solves the PMCHWT integral equation for the surface current on a dielectric sphere with $\varepsilon_r = 10$ and a radius of 1 m. The excitation is a plane wave with a wavelength of 10 m and a unit incident electric field. The analytical solution is obtained from a Mie series.

7. `sphere_current_tout_highdiel_muller_pw`: This tests solves the Müller formulation for the surface current on a dielectric sphere with $\varepsilon_r = 10$ and a radius of 1 m. The excitation is a plane wave with a wavelength of 10 m and a unit incident electric field. The analytical solution is obtained from a Mie series.

Table 4-1. List of VVTest simulations.

Case Number	Name	Geometry	Material	QoI	Equation	Excitation	p	Pass/Fail
1	`disk_tz_pecef_pw`	disk ($r = 1$m)	PEC	J	EFIE	$H = 1$ A/m; $\lambda = 1$ m	-2	Pass
2	`sphere_field_tout_pecef_pw`	sphere ($r = 1$m)	PEC	E	EFIE	$E = 1$ V/m; $\lambda = 10$ m	-2	Pass
3	`sphere_RCS_tout_pecef_pw`	sphere ($r = 1$m)	PEC	RCS	EFIE	$E = 1$ V/m; $\lambda = 10$ m	-2	Pass
4	`sphere_current_tout_lowdiel_PMCHWT_pw`	sphere ($r = 1$m)	$\varepsilon_r = 2$	J	PMCHWT	$E = 1$ V/m; $\lambda = 10$ m	-1	Pass
5	`sphere_current_tout_lowdiel_muller_pw`	sphere ($r = 1$m)	$\varepsilon_r = 2$	J	Müller	$E = 1$ V/m; $\lambda = 10$ m	-1	Pass
6	`sphere_current_tout_highdiel_PMCHWT_pw`	sphere ($r = 1$m)	$\varepsilon_r = 10$	J	PMCHWT	$E = 1$ V/m; $\lambda = 10$ m	-1	Pass
7	`sphere_current_tout_highdiel_muller_pw`	sphere ($r = 1$m)	$\varepsilon_r = 10$	J	Müller	$E = 1$ V/m; $\lambda = 10$ m	-1	Pass

5. FUTURE WORK

In this chapter, we describe our ongoing and future work, as related to the previously described activities.

5.1. GEOMETRICALLY SYMMETRIC QUADRATURE RULES FOR THE MFIE

Like the EFIE, the MFIE contains singularities that would benefit from similarly developed quadrature rules designed to integrate the singularities. In FY21, we will work on developing these rules.

5.2. MANUFACTURED SOLUTIONS

In FY21, we will continue efforts to develop manufactured solutions to mitigate the challenges and limitations described in Chapter 3. These approaches consist of finding ways to compute the singular integrals accurately, through variable transformations and adaptive integration.

5.3. AUTOMATED CODE-VERIFICATION TESTING

For FY21, code verification work will continue in four directions:

1. Refactoring `Verification_Tools.py`, `Error_Tools.py`, and `Plotting_Tools.py`
2. Automating code-verification testing through Jenkins
3. Adding additional tests to ensure greater coverage of the relevant physical phenomena
4. Working towards utilizing Sandia's Feature Coverage Tool (FCT) to quantify testing coverage of specific features

One of the first activities for FY21 is to refactor `Verification_Tools.py`, `Error_Tools.py`, and `Plotting_Tools.py`. After reviewing the Python tool scripts, refactoring these scripts will significantly simplify the code and reduce the effort to maintain the code. Refactoring will also simplify the VVTest scripts by having more compact calls to the Python tool functions.

To ensure the code continues to pass the code-verification tests as Gemma is developed, Jenkins (`https://jenkins-srn.sandia.gov`) will be employed to automate testing. Jenkins will be configured to run VVTest and notify the developers if the tests fail. These tests will run nightly for short tests and during the weekends for long tests.

The tests listed in Table 4-1 currently do not test all of the capabilities of Gemma. For example, certain types of boundary conditions and elements, as well as the ability to solve the MFIE and CFIE, are capabilities not covered by the current verification tests. In addition, the tests in Table 4-1 should report more QoIs to ensure complete coverage for each test. Since analytic solutions do not exist for many of Gemma's features, manufactured solutions are being developed to cover these features.

To ensure adequate test coverage and to provide Gemma's users with code-verification evidence, we will ultimately utilize the FCT. The FCT will be able to provide developers and users testing information on specific features utilized in application problems.

REFERENCES

[1] B. A. Freno, W. A. Johnson, B. F. Zinser, S. Campione, Symmetric triangle quadrature rules for arbitrary functions, Computers & Mathematics with Applications 79 (10) (2020) 2885–2896. `doi:https://doi.org/10.1016/j.camwa.2019.12.021`.

[2] B. A. Freno, W. A. Johnson, B. F. Zinser, D. F. Wilton, F. Vipiana, S. Campione, Characterization and integration of the singular test integrals in the method-of-moments implementation of the electric-field integral equation, arXiv preprint arXiv:1911.02107 (2020).

[3] P. J. Roache, Code verification by the method of manufactured solutions, Journal of Fluids Engineering 124 (1) (2001) 4–10. `doi:10.1115/1.1436090`.

[4] C. J. Roy, C. C. Nelson, T. M. Smith, C. C. Ober, Verification of Euler/Navier–Stokes codes using the method of manufactured solutions, International Journal for Numerical Methods in Fluids 44 (6) (2004) 599–620. `doi:10.1002/fld.660`.

[5] R. B. Bond, C. C. Ober, P. M. Knupp, S. W. Bova, Manufactured solution for computational fluid dynamics boundary condition verification, AIAA Journal 45 (9) (2007) 2224–2236. `doi:10.2514/1.28099`.

[6] S. Veluri, C. Roy, E. Luke, Comprehensive code verification for an unstructured finite volume CFD code, in: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, 2010. `doi:10.2514/6.2010-127`.

[7] T. Oliver, K. Estacio-Hiroms, N. Malaya, G. Carey, Manufactured solutions for the Favre-averaged Navier–Stokes equations with eddy-viscosity turbulence models, in: 50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, 2012. `doi:10.2514/6.2012-80`.

[8] B. A. Freno, B. R. Carnes, V. G. Weirs, Code-verification techniques for hypersonic reacting flows in thermochemical nonequilibrium, Journal of Computational Physics (2020). `doi:https://doi.org/10.1016/j.jcp.2020.109752`.

[9] É. Chamberland, A. Fortin, M. Fortin, Comparison of the performance of some finite element discretizations for large deformation elasticity problems, Computers & Structures 88 (11) (2010) 664 – 673. `doi:10.1016/j.compstruc.2010.02.007`.

[10] S. Étienne, A. Garon, D. Pelletier, Some manufactured solutions for verification of fluid–structure interaction codes, Computers & Structures 106-107 (2012) 56–67. doi:10.1016/j.compstruc.2012.04.006.

[11] A. Veeraragavan, J. Beri, R. J. Gollan, Use of the method of manufactured solutions for the verification of conjugate heat transfer solvers, Journal of Computational Physics 307 (2016) 308–320. doi:10.1016/j.jcp.2015.12.004.

[12] P. T. Brady, M. Herrmann, J. M. Lopez, Code verification for finite volume multiphase scalar equations using the method of manufactured solutions, Journal of Computational Physics 231 (7) (2012) 2924–2944. doi:10.1016/j.jcp.2011.12.040.

[13] R. G. McClarren, R. B. Lowrie, Manufactured solutions for the p_1 radiation-hydrodynamics equations, Journal of Quantitative Spectroscopy and Radiative Transfer 109 (15) (2008) 2590–2602. doi:10.1016/j.jqsrt.2008.06.003.

[14] J. R. Ellis, C. D. Hall, Model development and code verification for simulation of electrodynamic tether system, Journal of Guidance, Control, and Dynamics 32 (6) (2009) 1713–1722. doi:10.2514/1.44638.

[15] P. J. Roache, Verification and Validation in Computational Science and Engineering, Hermosa Publishers, 1998.

[16] K. Salari, P. Knupp, Code verification by the method of manufactured solutions, Sandia Report SAND2000-1444, Sandia National Laboratories (Jun. 2000). doi:10.2172/759450.

[17] P. Monk, et al., Finite element methods for Maxwell's equations, Oxford University Press, 2003.

[18] C. J. Roy, M. A. McWherter-Payne, W. L. Oberkampf, Verification and validation for laminar hypersonic flowfields, part 1: Verification, AIAA Journal 41 (10) (2003) 1934–1943. doi:10.2514/2.1909.

[19] R. Gollan, P. Jacobs, About the formulation, verification and validation of the hypersonic flow solver Eilmer, International Journal for Numerical Methods in Fluids 73 (1) (2013) 19–57. doi:10.1002/fld.3790.

[20] R. G. Marchand, The method of manufactured solutions for the verification of computational electromagnetic codes, phdthesis, Stellenbosch (Mar. 2013).

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email—Internal [redacted]

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov