Methodology for Flow and Salinity Estimates in the Sacramento-San Joaquin Delta and Suisun Marsh

32nd Annual Progress Report
June 2011

Chapter 6
Using Software Quality and Algorithm Testing to Verify a One-dimensional Transport Model

Authors: Eli Ateljevich and Jamie Anderson,
Delta Modeling Section,
Bay-Delta Office,
California Department of Water Resources

K. Zamani and F. A. Bombardelli,
Department of Civil and Environmental Engineering,
University of California, Davis

Contents

Figures

6 Using Software Quality and Algorithm Testing to Verify a One-Dimensional Transport Model

6.1 Introduction

In this chapter, we describe our approach and experiences developing a software verification framework for a one dimensional (1-D) transport model of advection, dispersion, and reactions or sources (ADR). We begin by describing the motivation and requirements for testing. Our acceptance criteria are driven by the requirements for the model, but are crafted according to principles from both the software and numerical testing fields. We then describe the components and implementation of the test suite, emphasizing the incremental nature of the tests, quantitative criteria for testing, and the similarities and tension between the silent, automatic perspective of software testing and the verbose, graphical outputs required for public reporting of numerical verification results.

The testing framework described in this paper was developed as part of a project to create a new transport module for the Delta Simulation Model 2 (DSM2), a 1-D hydrodynamic and transport model for flow and water quality in the Sacramento-San Joaquin Delta. Our target problems include river and estuary advection, and 1-D approximations of common mixing mechanisms and source terms associated with conservative and non-conservative water quality kinetics including sediment transport. The transport code is described briefly below followed by the development of the testing framework. The 2 are tightly coupled—since the transport module was created from scratch, it provided an opportunity to structure the code to be rigorously tested.

6.2 1-D Transport Model

The model used to illustrate the testing framework is based on the 1-D transport equations in conservative form:

$$\underbrace{\frac{\partial\left(A(x,t)C(x,t)\right)}{\partial t}}_{\text{Time evolution}}+\underbrace{\frac{\partial\left(A(x,t)C(x,t)u(x,t)\right)}{\partial x}}_{\text{Advection}}=\underbrace{\frac{\partial}{\partial x}\left(A(x,t)K(x,t)\frac{\partial C(x,t)}{\partial x}\right)}_{\text{Dispersion}}+\underbrace{R\left(x,t,C(x,t)\right)}_{\text{Source/Reaction}} \qquad \text{Eq. 6-1}$$

where *x* is the distance, *t* is time, *A* is the wetted area, *C* is the scalar concentration, *u* is the flow velocity, *K* is the longitudinal dispersion coefficient, and *R* is the source term (deposition, erosion, lateral inflow, and other forms of sources and sinks). Eq. 6-1 describes the mass conservation of a pollutant in dissolved phase, or suspended sediment away from the streambed.

The problem domain includes estuarine river channels and even some small open water areas roughly approximated as channels. The main transport process is advection, and the mixing mechanisms we anticipate are turbulent diffusion, gravitational circulation, and shear dispersion (Fischer, et al. 1979) (Abbott and Price 1994). We anticipate the shear dispersion to dominate over the turbulent diffusion. We also expect the gravitational circulation to exert an important role in mixing. We additionally contemplate significant, non-linear source terms from sediment, chemical and biological processes, though none of the processes are so quickly varying as to constitute truly stiff reactions.

Our algorithms include an explicit scheme for advection based on a finite-volume method (FVM) discretization and the Lax 2-step method (Colella and Puckett 1998) with van Leer flux limiter (Saltzman 1994). It also includes an implicit, time-centered Crank-Nicolson scheme for dispersion (Fletcher 1991). The advection and reaction solver are coupled as a predictor corrector pair, and dispersion is implemented separately using operator splitting.

6.3 Testing Requirements

The tests described in this chapter are all designed around suitability of the solver for estuary transport problems. The required accuracy on target modeling applications and choice of algorithm influence the testing requirements and the components of our algorithm test suite.

The scales of estuary transport determine the range of relative strength over which we test advection, diffusion and reactions, which is mostly intermediate Peclet number flow. Our target accuracy is strict second order for individual operators and near second order for the algorithm as a whole. Second order allows a coarser discretization for a modest increase in work per volume of fluid, which is efficient. A second-order algorithm also gives us a buffer of accuracy as details like networks of channels and coarse boundary data are added. At the time of this writing, our splitting is first order Godunov splitting. Some authors, e.g. (Leveque 1986), have observed that near second-order accuracy can be achieved with first order splitting, and the design of the tests probes this point.

Two features of the algorithm feature into the design of our test. First, the scheme requires a flow field (flow discharges and flow areas) that preserves mass continuity. In some cases, tests from the literature were written in non-conservative or primitive form and had to be reworked in conservative form. Second, we employ operator splitting and wanted to exercise the equations with and without known vulnerabilities (such as time-varying boundaries and nonlinear source terms) of this class of algorithm.

6.4 Testing Principles

Flow and transport codes inherently comprise both numerical algorithms and pieces of software. Well-developed testing literature exists for both. Oberkampf and Trucano (2002) describe some elements of software quality engineering (SQE) in the context of numerical verification and note some cultural reasons why it is seldom implemented.

Figure 6-1 is adapted from this work and depicts the relationship between software testing components and algorithmic testing such as convergence tests. We regard numerical verification as our key responsibility and the numerical verification toolset as our greatest assets. Nonetheless, we also comment below on how these tools feature as tests and how, at times, they seem in tension with the principles of good software testing.

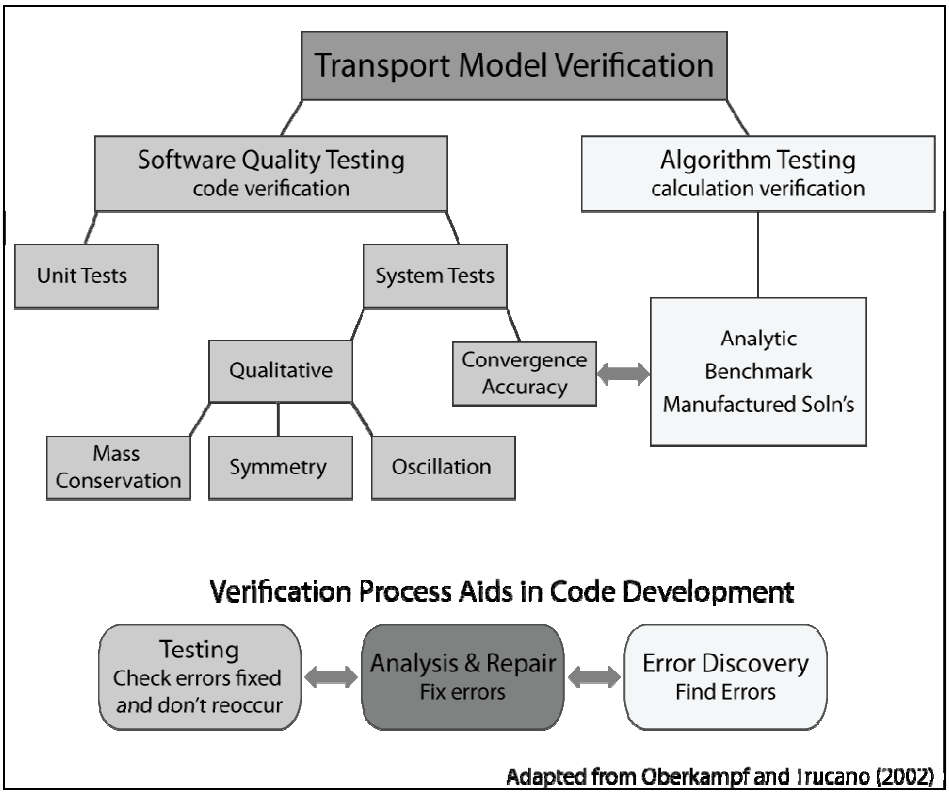

Figure 6-1 Relationship between software testing components and algorithmic testing

6.4.1 Software Testing Principles

The principles that we want to emphasize are:

1. Testing should be automatic and continuous.
2. The approach should foster exact specification of every unit of code.
3. Testing should provide assurance of whether a set of specifications is met.

One goal of tests is that they be a continuous assessment of the code. The entire testing system is a regression suite that establishes a gauntlet through which future code changes must be passed. A consequence of automation is that tests must be phrased in terms of binary assertions, true and false statements that can be tested without human intervention and that reveal whether the aspect of the code under consideration is correct. Convergence criteria are a rigorous basis for assertions, either by requiring strict convergence criteria ("the algorithm is second order accurate in time and space") or a regression criterion ("convergence will not get any worse than last time the code was tested").

The software testing literature further distinguishes between unit tests of atomic routines and system tests of larger subtasks. For example, the evaluation of a gradient might be a unit of code, and it would have a unit test. Convergence tests and other algorithm tests are examples of system tests.

The unit testing point of view is that code must be exercised over a range of inputs that covers every line. For instance, to test a gradient routine with a slope limiter, a developer would want to cover:

1. smooth cases in the middle of the mesh;
2. behavior near the edges of the mesh, where one-sided differences may be used instead of central differences;
3. cases that test the limiters with steep or zero gradients in both directions.

Any system test will certainly exercise the gradient code in the middle of the mesh, which in any event can seldom be wrong without being obvious. However, system-level tests might miss the more unusual cases. For example, a convergence test may miss a bug in the limiter for the case of steep decreasing slope for several reasons. First, convergence is often assessed with limiters turned off, as they are locally order reducing. Second, it is hard to fiddle with the problem in just the right way to make sure the left, right, and center cases of the gradient limiter are all triggered. This is particularly true when trying to exercise other units of code at the same time—parameter choices made to fully exercise gradient limiter may lessen the coverage of another unit.

Although the software and algorithm tests are separate, information discovered during one test can aid in the further development of another test. We began our coding with near-100% coverage by unit tests. These tests were part of the debugging and development processes. Later, discoveries made in the context of system tests were analyzed and pushed back into unit tests whenever possible. The unit test was expanded to verify that the newly discovered error from the algorithm test was fixed and does not reoccur. This flow of information is indicated in Figure 6-1.

One example of this accumulation of tests is our unit test for fluid mass conservation. The observation that our algorithm requires accurate mass conservation of the fluid came from the tidal test case. The flow field we used for this case was adapted for 1-D from Wang et al. (2009). The original solution was based on a linearization and is not mass conservative in 1-D, causing significant problems with transport convergence. Once this requirement was discovered, a unit test was introduced into the suite to check this property for any flow field. At the same time, we found we had to tailor some of the analytical results we were using for other tests.

A second example involved periodic flow. Our uniform flow convergence tests originally had a reversal of flow midway through the test. The out-and-back setup is convenient for advection because the initial condition and final concentration field are the same. We also believed we were exercising the code in 2 directions. In fact, an error accumulated in the positive direction was cancelled by the return pass in the negative direction. We passed the periodic test but failed analogous unidirectional tests. Originally, the discovery was fortuitous because the unidirectional test was “unofficial”; now we test directional dependence using a combination of periodic and unidirectional flow

6.4.2 Numerical verification and algorithmic testing

An important category of a system test includes the algorithm tests normally associated with verification of numerical codes. Algorithm tests serve multiple purposes. They are intended in part to discover bugs and in part to convince ourselves and others of the merit of the algorithm to solve the equations to which it is directed.

One of the well-recognized and standard verification methods of computational fluid dynamics codes is based on the notion of mesh convergence (Roache 2009). Mesh convergence for models that solve

partial differential equations is assessed by successively refining the spatial and temporal discretizations. As the mesh is refined, the error estimates (for us usually an L_1 norm, or sum absolute error) should decrease at a *convergence rate* that is algorithm dependent (Leveque 2002). A second order accurate algorithm, denoted O(2) or $O(\Delta t^2, \Delta x^2)$ should have its error go down proportional to the square of the step sizes. By checking convergence, we ensure that the model is consistent with an underlying formulation rather than numerical artifacts. Failure to converge usually represents either a bug in the implementation or a difficulty of the algorithm on a class of problem.

The verification toolkit is largely targeted at providing test problems and methods to estimate error in situations where an analytical solution is not available from the literature. When nonlinearity, spatially varying coefficients and other complexities are introduced, tricks must be introduced to obtain good test problems.

Depending on the context, error and convergence are usually estimated one of 2 ways:

- When successive refinements are assessed relative to an analytical solution, we have a direct estimate of error and the ratio allows us to estimate a *convergence rate.*

- When successive grids are compared to one another, we can invoke the concept of Richardson extrapolation and Grid Convergence Index (Roache 2009) to indirectly estimate error and convergence even when no solution is available.

In practice, we found the Method of Manufactured Solutions (MMS) (Roache 2009) was able to supply analytical verification problems for all the cases not covered directly in the literature.

At least in theory, convergence rates can be stipulated as a project requirement and software testing assertion. Convergence rates, not absolute error, are what numerical methods tend to promise, and they are very useful in the discovery of code defects. Still, the main goal in practice is a more accurate solver. Therefore, the superiority of methods should be assessed based on both convergence and accuracy (Roache 2009).

The convergence ratio in a very coarse grid oscillates around its main value; as the grid size is refined, convergence becomes monotonic until the mesh size reaches a point where the machine precision overtakes the truncation error of the numerical scheme. At this point, error norms do not change, and the convergence rate is zero. Convergence ratios should be checked for intermediate grid sizes, preferably at the scale of the real phenomenon and discretization used in practice. In the conclusions, we describe the challenge of dealing with tests that returned failed results when the convergence was just slightly below the target level.

As acceptance tests, algorithm tests should be conducted over a range of problems that exercise the major physical features that are to be modeled. The community may help with this by providing benchmarks, but we were unable to ascertain any widely accepted benchmarks for a 1-D transport code. As system tests we believe that the tests should be *glass box*, targeting known or discovered vulnerabilities of the algorithm. The ability to use remote and active boundaries in our convergence tests, for instance, is specifically motivated by known problems related to operator splitting.

Finally, distinction might be made between the reportable set of algorithm tests and other types of system tests aimed at defect discovery. Important examples of the latter are tests of symmetry, such as a whether a 1-D model gives the same result when the upstream and downstream boundaries are

swapped. Others are positivity preservation of constituents, mass conservation, and oscillation detection. In the case of positivity preservation and mass conservation, it is typical to abstract this code for use both in the test suite and in the driver as a user option.

Overall, we agree with the conclusions of Salari and Knupp (2000) that system tests—particularly convergence tests—expose bugs well, particularly when an attempt is made to test symmetrically and over special cases. We feel that the incremental approach we describe in the next section further helps to isolate problems. Nevertheless, a close reading of Salari and Knupp (2000) does reveal that the convergence tests sometimes initially failed to pick up bugs that are exactly the sorts unit tests might catch (e.g., gaffes in corner cells).

6.5 Algorithm Test Suite Description

The algorithm testing used an incremental building block approach that adds complexity on 2 major dimensions (Figure 6-2):

- Operators: The tests were developed for a 1-D transport code that will be applied to an estuary. Thus the key processes tested are the operators of advection, dispersion, and reaction (e.g., growth or decay). These are tested individually, then in combinations of growing complexity

- Flow field and physical setup: Our fixtures included the following cases

  - Uniform flow: This test involved uniform steady flow on a channel, sometimes with a reverse in direction halfway through the simulation. The mass transported is Gaussian. The suite includes advection, diffusion, and reaction alone and in the combinations indicated in Figure 6-2.

  - Tidal flow: This test used a tidal flow field from Wang et al. (2009), adapted to be 1-D and mass conserving, to test advection and reaction. The test itself has no analytical solution, but is periodic in a way that is not symmetric.

  - Spatial variation (Zoppou): This test is due to Zoppou and Knight (1997), and includes velocity proportional to distance and diffusion coefficients proportional to distance squared. This test had to be modified for a conservative fluid flow.

- Boundary complexity: For the uniform flow and Zoppou tests, we include cases where the boundary is far away from the transported mass and cases where the boundary is actively part of the problem. This allows us to determine the extent to which convergence rates are affected by boundaries.

- Nonlinearity: In our final case, which uses the Zoppou and Knight (1997) fixture adapted using the MMS, we include a non-linear source term.

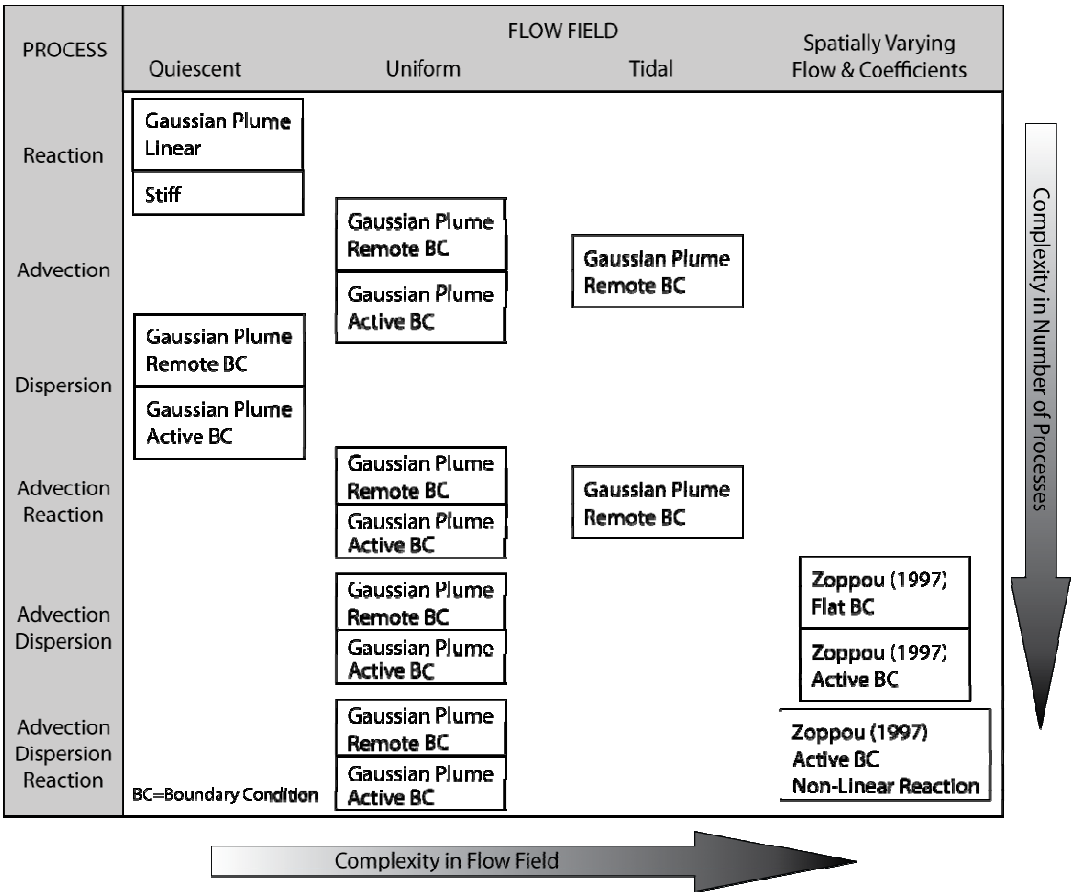

Figure 6-2 Transport algorithm testing with incremental complexity

These tests were conducted for a range of parameter values. Typically the Courant number (a measure of numerical stability of the algorithm), domain length, and dispersion and decay coefficients were fixed; and the grid spacing and time steps were adjusted to maintain the same Courant number. Detailed descriptions of the tests are beyond the scope of this paper and will appear in a planned journal article.

Our incremental suite can identify with good precision exactly which added layer of complexity causes a drop in order of accuracy. For instance, our example algorithm performs well when boundaries are remote, but drops to a convergence rate of *O(1.4)* or so in the presence of active boundaries.

6.6 Architecture and Implementation

The test architecture was implemented using the FORTRAN Unit Testing Framework (FRUIT) for logging assertions and counting pass rates. FRUIT is one of the few test frameworks available in this computer language. FRUIT does not appear to adhere to industry practices in the way it formats results (e.g., the JUnit format), but provides a variety of predefined assertions.

Both the system tests and the unit tests were developed with FRUIT, and the granularity for unit tests is one unit test module per solver module, one unit test routine per solver routine.

Our code was designed for testing. In particular, computational routines were crafted according to the following 3 architectural considerations:

- We isolated any computations that could be described with easy-to-understand names, with the caveat that we did not want to degrade performance or prevent vectorization. Our routines tend to be simple, homogenous calculations over arrays (such as calculating the gradient over the entire domain) rather than long sequences of instructions on individual cells.

- Data are passed to computational routines by argument list. This leads to longer argument lists, but makes the description of input and output much surer—tests are much harder to program when data required by the routine is passed in "behind the scenes" using imported modules.

- The design allows us to dynamically swap in new sources, flow fields, and boundary conditions without halting the tests or recompiling the code. This ability required function pointers and abstract interfaces, a relatively new FORTRAN feature.

6.7 Challenges and Issues with Tests

The key issues associated with unit tests were different than those associated with algorithm tests. The main challenge with unit tests seems to be culture: generating the will to write them and the skills to write them in a way that covers the unusual cases. Without the aid of special coverage tools, test coverage is up to the diligence and craftiness of the developers.

For algorithm tests, nominally we sought a second order convergence rate. A convergence criterion seemed in-keeping with the way numerical algorithm accuracy is expressed and is less arbitrary than a hard-wired, scale-dependent absolute standard. Early on, however, it was clear that the normal noise from observed convergence rates could spoil even a success when the rate is expressed as a hard assertion. It is challenging to deal with situations when a convergence test fails with a value close to the criterion, e.g., 1.97 instead of 2.0, which surely would pass a graphical acceptance test. This issue can be exacerbated by sensitivity to problem parameters.

When one of our tests did not cleanly converge at the specified level, we generally either fixed the code successfully or we searched for bugs until both of the following things happened:

- Convergence properties corresponded well to the expected strengths and limitations of our algorithm; and

- The solution was accurate: convergent above first order, excellent qualitative results when compared graphically to solutions and with relative errors of a hundredth of a percent.

We have done our best to support our claims when attributing any convergence deviations to specific algorithmic or problem quirks. Our incremental suite can identify with good precision exactly which added layer of complexity causes a drop in order of accuracy. Where we intend to relax convergence criteria, we are in the process of changing our assertion criteria to an absolute accuracy requirement coupled with a regression standard for convergence. In our numerical code, cases with multiple operators and very active boundaries are the only ones in which we currently expect such a compromise.

Finally, there is sometimes a tradeoff between the requirements for verification and best practices for error discovery. Part of the community verification process for transport codes is the presentation of results in graphical format. Accommodating this type of display requires output beyond mere reports of assertion failures. We added the required verbosity option, but graphical interpretation plays no part in our regular testing practices other than as a debugging tool.

6.8 Conclusions

Our test suite succeeds both in finding bugs and in elucidating the strengths and weaknesses of a 1-D transport algorithm. We feel that our test suite is parsimonious and reasonably complete for tidal applications. Applying the framework to our own code, we have been able to work towards second order convergence for many tests and to isolate problems in special cases.

We believe the essential ideas in our approach are:

- Codes must be written in a modular format with software testing in mind in order to apply the principals of software quality engineering. Each piece of code must have a clear purpose and criterion for success.

- Tests should be silent and automatic. Test criteria must be binary assertions. Assertions are written to provide more information than simply assessing graphs of expected vs. computed results; however, we include verbosity options to export data for graphs.

- There is a symbiotic relationship between software and algorithm tests; Code bugs detected with algorithm tests can lead to development of additional software regression tests to verify that a bug is fixed and to provide assurance that it does not reoccur.

- Convergence tests are the principal tool used in the algorithm verification literature. Our suite includes convergence tests on a combination of analytical problems from the literature and a manufactured solution using MMS.

- When convergence criteria are implemented as hard test assertions, account must be made of the small random noise typical of convergence results.

- Incremental addition of complexity helps to isolate the causes of problems and to establish that lower complexity solutions are correct.

- Symmetry and directionality tests help discover errors that may be hidden by the setup of the problem.

The software quality and algorithm testing framework described in this paper provides a useful starting point for researchers and practitioners wanting to verify transport codes. Having this rigorous test suite allows developers (1) to verify that each piece of code works properly both individually and as a combined system, (2) to ensure additions to the code do not adversely affect existing code, and (3) to find and fix code bugs that might otherwise be missed. Providing the end user with test results and the ability to rerun the tests themselves, assures the user that the code performs as expected and quantifies the code's strengths and weaknesses.

6.9 Acknowledgments

This work has been possible due to the support given by the California Dept. of Water Resources through a contract with UC Davis. We thank Tara Smith and Dr. Francis Chung from DWR for their support of this project.

6.10 References

Abbott, M. B. and W. A. Price. 1994. *Coastal, Estuarial, and Harbour Engineers' Reference Book.* Chapman & Hall.

California Department of Water Resources. *Delta Simulation Model 2 (DSM2). [Web page].* http://baydeltaoffice.water.ca.gov/modeling/deltamodeling/models/dsm2/dsm2.cfm (accessed 2011).

Colella, P. and E. G. Puckett. 1998. "Modern Numerical Methods for Fluid Flow.".

Fischer, B. H., J. E. List, R. C. Koh, J. Imberger, and N. H. Brooks. 1979. *Mixing in Inland and Coastal Waters.* Academic Press, Inc.

Fletcher, C. A. J. 1991. *Computational Techniques for Fluid Dynamics.* Springer.

Leveque, J. R. 2002. *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press.

Leveque, J. R. 1986. "Intermediate Boundary Conditions for Time-Split Methods Applied to Hyperbolic Partial Differential Equations." *Mathematics of Computation* 47: 37-54.

Oberkampf, W. L., Trucano T. G. 2002. *"Verification and Validation in Computational Fluid Dynamics .* SANDIA REPORT, No. SAND2002-0529.

Roache, P. J. 2009. *Fundamentals of Verification and Validation.* Hermosa Publishers.

Salari, K., Knupp, P. 2000. *Code Verification by the Method of Manufactured Solutions.* SANDIA REPORT, No. SAND2000-1444.

Saltzman, J. 1994. "An Unsplit 3D Upwind Method for Hyperbolic Conservation Laws." *J. of Computational Physics* 115: 153-168.

Wang, S. Y., P. J. Roache, R. A. Schmalz, Y. Jia, and P. E. Smith. 2009. "Verification and Validation of 3D Free-Surface Flow Models." *ASCE, EWRI.*

Zoppou, C., and J. H. Knight. 1997. "Analytical Solution for Advection and Advection-Diffusion Equation with Spatially Variable Coefficients." *J. Hydraulic Eng.* 123(2): 144-148.