The California Department of Water Resources maintains Delta Simulation Model 2, a one dimensional hydrodynamic and transport model for rapidly simulating flow and water quality in the Sacramento-San Joaquin Delta. Recently, the authors commenced work on a more flexible and more rigorously verified transport component for this suite. Our target problems include river and estuary advection, 1D approximations of common mixing mechanisms and source terms associated with sediment, radiation and non-conservative water quality kinetics.

In this paper, we describe our approach and experiences developing a software verification framework for a transport model. We begin by describing the motivation and requirements for testing. The 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 suite, emphasizing the incremental nature of the tests, quantitative criteria for testing and the tension between the silent, automatic perspective of software testing and the verbose, graphical requirements required for public reporting of numerical verification results.

Both the scaling of the problem and our choice of algorithm influence the components of our test suite. The dynamics are formulated in conservative form as follows:

\*\*\*\*\*\*

The problem domain includes estuaries and river channels and even some open water areas grossly approximated as channels. The mixing mechanisms we anticipate are \*\*\*\*. We anticipate blah to dominate blah, with Blah Numbers exceeding \*\*\*. We also contemplate significant, non-linear source terms. While none are so quickly varying as to constitute truly stiff reactions, we have \*\*\* with \*\*\*\* Numbers of \*\*\*\*. O

Our algorithms includes \*\*\*\* advection with \*\*\*\* and \*\*\*\*. The advection and reaction solver are coupled as a predictor corrector pair, and diffusion is implemented using operator splitting. Two features of the algorithm are particularly important. First, the scheme requires a flow field (\*\*\* and \*\*\*) that preserves mass continuity. In some cases tests from the literature were written in non-conservative or primitive form in terms of a velocity 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.

The target accuracy is strict second order accuracy for individual operators and near second-order accuracy for the algorithm as a whole. Second order allows coarser discretization for a modest increase in work. As computer architectures favor multiple operations with minimal movement of data, this advantage of computation over grid density is on the increase. 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 writing our splitting is first order Godunov splitting; one goal of our testing is determine how applicable the practical observation that near second-order accuracy can be achieved with first order splitting (e.g. Leveque \*\*) is to our algorithm.

Testing Principles:

Flow and transport codes inherently comprise both numerical algorithms and pieces of software. A well-developed testing literature exists for both. \*\*\*SANDIA describes some elements of software quality engineering in the context of numerical verification, and notes some cultural reasons why it is seldom implemented

There are myriad approaches to software testing. The principles that we want to emphasize are:

1. Testing should be automatic and continuous.
2. The approach should foster good architecture and exact specification of every unit of code
3. Tests should be designed around the specific ways developers make mistakes.
4. The approach should test ever important special case for subprocesses.
5. Tests need not have diagnostic value.

One goal of tests is that they be a continuous assessment of the code. The tests themselves stay static, and establish a gauntlet of through which future changes must be passed. Such a test is usually called a regression test, and regressions are harvested from past unit tests, system tests and mistakes.

One consequence of automation and regression is that test suites must be based on binary *assertions,* true and fall statements that can be tested without human intervention and that reveal whether the aspect of the code under consideration is correct.

Unit tests exercise single units of code, such as one small subroutine calculating a gradient over a mesh. Properly designed unit tests exercise all the important “corner” cases and special inputs. Examples for a gradient would be:

1. One well-behaved central case.
2. A test of edges of the mesh.
3. Tests that exercise features such as flux limiters with steep or zero gradients in both directions.

Unit tests are fast, and they allow a great number of special cases to be exercised, which might take months when the case is embedded in a larger solver. For instance, in the example of a gradient with limiter, imagine that the limiter contains a bug in one direction. Normal verification based on convergence might miss this in two ways. First, convergence is usually assessed without limiters, as they are order reducing. Second, algorithmic tests are usually constructed to show the algorithm solves a Partial Differential Equation, not that the code does this right in both directions.

\*\*\*\* unit test vs Sandia assertion the convergence tests catch everything.

Incremental 9454948223

above example, it is easy to imagine a full system test

1. Once passed, unit tests act as regression tests.

The only downsides of unit tests seem to be unawareness or reluctance to code them. These problems can be exacerbated during parallel computation.

Regression tests: these tests rehearse old mistakes to prevent new coding from re-introducing errors. Regression testing is often the only type of test that can be applied to a legacy code whose architecture is not test friendly.

System or integration tests: System tests include the interaction of components in a meaningful way. A small system might represent a single operator in the algorithm. A large system might represent the whole time update. The behavior tested by a system test will make sense to a user and the criteria are is often reportable artifacts that might be used in a publication. In this regard, we have observed at least one “cultural” issue: the criterion that a test be repeatable and automatic may conflict with the desire to report results.

Numerical testing: