Improving the Verification and Validation Process

Mike Fagan
Rice University

Dave Higdon
Los Alamos National Laboratory
Notes to Audience

- I will use the much shorter ‘VnV’ abbreviation, rather than repeat the phrase ‘Verification and Validation’
- I also include the body of knowledge referred to as Uncertainty Quantification in VnV

(You may see the acronym VVUQ in other work, but I will not use it)

- I will (sometimes) use UQ when discussing something specific to Uncertainty Quantification
Why is VnV important?

• Computational Science (and Engineering) is widespread
  — Less expensive than actual physical plant and equipment
    – Wind Tunnels
  — Some experiments cannot (should not) be done
    – Big bang, climate/weather
    – Medical, nuclear explosions
  — Practice of Computational Science is increasing

• Tacit Assumption of Computational Science
  — Simulation programs are an accurate model of reality
    Programs must be ‘right’
    – Degree of accuracy/uncertainty must be known (analogous to experimental error)

• VnV is the body of knowledge that that ensures the tacit assumption
Definitions

• Verification
  — This is about computer codes
  — Verification answers the question:
    Does the computer code implement the specified model?
    – Numerical properties need to be verified as well
  — Are we solving the problem “right”?

• Validation
  — This is about models
  — Validation answers the question:
    Does the model accurately approximate the real world?
  — Are we solving the “right” problem

• Uncertainty Quantification:
  — Given some measure of the ‘uncertainty’ in the inputs, what is the corresponding measure of uncertainty in the outputs?
More on Verification and Validation

• Sometimes (often?) the line between verification and validation is blurred
What This Workshop is About

• Techniques for improving quality of VnV information
• Techniques/Tools for reducing the development time devoted to VnV
• Techniques for reducing the running time of VnV program runs

Note: Goals are not mutually exclusive
Scope/Outline of the Workshop

• VnV is a biiiiiig field
  — A lot of disciplines have knowledge to contribute
  — So, no 1-stop shopping
  — Possibilities for interdisciplinary collaborations are strong!

• Outline for Workshop
  — Validation Process Improvement via Adjoint Methods
  — Verification via the Method of Manufactured Solutions (MMS)
  — UQ via Taylor Models

  — Simulation-Based Augmentation of Experiments
A Unifying Concept

• 3 techniques specified on the previous slide have a unifying enabler --- accurate and efficient computation of derivatives
  — Adjoint Methods work by computing the transpose (adjoint) of the Jacobian (derivative) Matrix
  — MMS requires the derivatives of the manufactured solution
  — Taylor Models require computation of Taylor series coefficients \( \equiv \) derivatives
Methods for Computing Derivatives

• Difference Methods
  — Compute the function
  — Pick a perturbation size
  — Perturb a chosen independent variable*
  — Compute function using perturbed independent variable*
  — Subtract*
  — Divide
  — Repeat for all independent variables of interest

• Symbolic Methods
  — Analyze the function implementation components*
    – Break them down into “manageable” pieces* (possibly single assignment statements)
  — Differentiate each component
  — Compute the derivative by applying the chain rule to each component
Comparison of Derivative Methods

• Difference Methods
  —Development time is minimal +
  —Choosing a perturbation (“h”) –
  —Inaccurate and/or inefficient –
  —No adjoint equivalent –

• Symbolic Methods (by hand)
  —Can be accurate and efficient +
    (depends on the programmer)
  —Development time is long – –
  —Maintaining derivatives an additional burden –

• Ideal: Symbolic methods, but short development time
Realizing the Ideal: Automatic Differentiation (AD)

• Combines the best of finite differences and by hand sensitivity calculation

• Program synthesis tool
  — Shorter development time

• Derivatives computed this way are
  — Analytically accurate
  — Always faster than central differences, frequently faster than 1-sided differences
  — Adjoint/reverse mode is enabled*
How does it work?

- Each assignment statement is augmented with derivatives
- Chain rule assures propagation is correct

\[ Y = A \times X \times X + B \]

\[ \begin{align*}
\text{P}_A &= 2 \times X \\
\text{P}_X &= A \\
\text{P}_B &= 1.0 \\
\text{CALL ACCUM} & (G_Y, \text{P}_A, G_A, \text{P}_X, G_X, 1.0, G_B) \\
Y &= A \times X \times X + B
\end{align*} \]
How does it work (cont.)?

• Given access to the source code, an AD tool can use compiler-like program analysis tools
  — Activity analysis
  — Program Flow Reversal
  — Linearity (Polynomyality) Analysis
  — Common Subexpression Elimination

• AD combines numerical analysis with program analysis
• My research focus is in AD
• LACSI has funded a lot of that research
• In particular, LACSI is primarily responsible for development and improvements of Adifor over the previous 5 years.

• Adifor is an AD tool for Fortran 77, and Fortran 90 that is freely available to Government, and University Researchers.
• It is not open source at this time.
‘Classic’ Validation

• Also ‘test-and-regress’, sometimes ‘learn-and-burn’
• ‘Classic’ validation methodology
  — Separate “real world” data into 2 partitions: “tuning” and “testing”
  — *Optimize the parameter settings on the “tuning” data to minimize simulation vs “real world”
  — Assuming the error in the tuned simulation is “small”
    – Run the tuned simulation on the “testing” data set
    – Check for “small” error
• Many variations on this methodology
  — How to separate data, and how many times
  — How to determine “small”
‘Classic’ Validation Bottleneck

• The Optimize/Tuning step
  — Fundamentally an iterative process
  — So smaller numbers of tuning steps means better performance
  — If the function being validated is differentiable a.e., then you want to be using some flavor of Newton’s method

• Newton-style method bottleneck
  — Newton’s method must have derivatives (or an approximation)
  — Difference methods require at least \#independent-vars + 1 function evaluations.
  — Automatic Differentiation (forward) can improve the efficiency of the required derivative computation, but even AD complexity is still \(O(\#\text{indep})\) function evaluations
What is an adjoint method?

- Mathematically, computing a derivative for an expression is computing a product of matrices [That’s the chain rule]
  \[ J = J_1 \times J_2 \times J_3 \ldots \times J_n' \times J_n \]

- If \( J_n \), however has a small number of columns, then computing the \( J \) product above using this grouping of terms
  \[ J = (J_1 \times (J_2 \times \ldots \times (J_n' \times J_n))) \ldots \]
  will save substantial time because all of the intermediate results are smaller.

- Multiplying in reverse order is equivalent to computing the transpose in forward order, because the transpose operation reverses the order of multiply operands
  \[ J^T = ((J_1 \times J_2) \times \ldots \times (J_n') \times J_n)^T \]
  \[ = (J_n^T \times J_n'^T) \times \ldots \times J_1^T) \]

- Transpose is the matrix operation, in Hilbert space, it is called the adjoint. Hence, **adjoint method**
Adjoint Methods, cont.

• The key requirement for adjoint method efficiency is that the last operation should have a small number of columns. Ideal number of columns for adjoint methods is 1.

• In other words, functions that take a large number of inputs but return a single scalar are prime candidates for an adjoint method. In particular, residual-error type functions are prime candidates.

• Recall that there is no finite difference method that corresponds to the adjoint. Hence, any adjoint method for computer codes must be of the symbolic (chain rule) variety.

• Here is the tricky part: For computer programs, computing the transpose products requires reversing the control flow of the program! Furthermore, since variables may be overwritten, a lot of the intermediate values must be stored.

• Before the widespread availability of AD tools, most adjoint codes were constructed by hand. (AD = grad student). VERY TEDIOUS
Adjoint Methods, cont.

• AD tools, however, have made the development of adjoint codes much tedious, error prone, and fragile.

• Bottom Line: The complexity of an adjoint derivative computation is $O(\#\text{dependent-vars})$ function evaluations

• Theoretical constant is between 4 and 7 (5 is most often cited)

• In practice*, 5 is often achieved by experienced adjoint developers, and I have seen constants as low as 3 for some codes

• In an ideal adjoint situation (such as a metric), \# dependent vars is 1, so computation of derivatives for this metric takes about 5 function evaluations --- no matter how many independent variables!
AD for Adjoints

• In practice, using AD to develop an adjoint code is still not completely automatic. The memory management aspect is still an active research area. [AD is not always A]

• Still, it is a huge win …
  —Informal anecdotal research among colleagues estimates the development of an adjoint without AD is 1 month/1000 lines
  —With a modern AD tool, the development time is estimated at 5 days/1000 lines.
• Recall that Classic validation Newton-based tuning steps require the derivative of a residual error function with respect to the tuning parameters. This is a classic adjoint situation.

• By using adjoint methods, the cost of a Newton style tuning step is fixed at roughly 5 function evaluations. So any time there are more than 5 tuning parameters, Adjoint methods are a win.

• The size of the win depends on how many tuning parameters are present.

• Furthermore, AD enables a substantial reduction in the development of an adjoint code.

• CLAIM: AD improves the classic validation step by enabling efficient development of the adjoint-based tuning steps.
Detonation Shock Dynamics (DSD) Curvature Equation

\[
\frac{D_n}{D_{CJ}} = 1 + A \cdot \left[ (C_1 - \kappa)^{e_1} - C_1^{e_1} \right] - B \cdot \kappa \left[ \frac{1 + C_2 \cdot \kappa^{e_2} + C_3 \cdot \kappa^{e_3}}{1 + C_4 \cdot \kappa^{e_4} + C_5 \cdot \kappa^{e_5}} \right]
\]

How could one tune these 6 parameters??
DSD - better fit of 6 parameters

- SNL DAKOTA package drives the optimization process
- Gradients provided by AD of DSD solver
- ~40 passes improves the fit
Method of Manufactured Solutions (MMS) for Verification

- MMS is a way of verifying correctness of numerical properties for differential equation solvers.
- Take a prototypical differential equation (ODE or PDE):
  \[ \text{find } f(x,t) \text{ s.t. } D(f) = F \text{ subj. to } BC(x,t) \]
  where \( D \) is some differential operator, \( F \) is some forcing function, and \( BC \) is the set of boundary conditions.
- A differential equation solver, then, is a function 
  \[ S(D,F,BC) = f \text{ (approximately)} \]
  The solution is often realized as a set of numbers \( f(x,t) \).
  The true \( f \) often has no closed form solution.
- There are a lot of numerical methods for solving differential equations, all having various convergence and accuracy properties.
- MMS provides a nice general way of testing various properties of a DE solver.
Overview of MMS

• Recall that a solver is a function
  \( S(D,F,BC) = f \)

• It would be simple to verify \( S \) if \( f \) was in closed form --- just check closed form \( f \) with values computed by \( S \)

• MMS is almost as good.

• The MMS process:
  — Select a computable function \( f_{\text{test}} \) (that satisfies the BC)
  — *Manufacture a forcing function \( F_{\text{TEST}} \) by computing \( D(f_{\text{test}}) \) at several points on the grid.
  — Now run the solver \( S(D,F_{\text{TEST}},BC) \)
  — Compare the solver output with \( f_{\text{test}} \) values.
    Grid properties are known, so MMS practitioners can actually test convergence, order-of-accuracy, etc
Improving the MMS process with AD

• To manufacture FTEST, MMS practitioners need to be able to evaluate D(ftest) for various differential operators.

• Accuracy of D(ftest) is crucial. If inaccurate FTEST values occur, then they might be the source of contamination in the verification process.

• Most MMS practitioners pick fairly simple ftest functions, and evaluate the differential operator by inspection, or occasionally employ a computer algebra tool like Mathematica.

• Our innovation, write ftest as a program, and use AD to evaluate D(ftest)
  —You can write more complicated functions → improves quality
  —Development of MMS solutions is easier → improved productivity
Caveats on AD MMS

• Main tedium is higher order derivatives not easily available for many AD tools.
• Scripts help
• Situation is improving
Verification Experiment

Using a 4\textsuperscript{th} order Runge-Kutta on the following equation,

\[ u' = \frac{u^2 + \sin(u)}{u^2 + 1} \]

Verified by AD MMS methods
Uncertainty Quantification via Taylor Models

• Simplest form of Uncertainty Quantification uses linear approximation:
  — All program input variables are represented as 1st order multi-variate Taylor series
  \[ x = x_0 + x_1 U_1 + \ldots x_n U_n, \text{ where the } U \text{ are normal}(0,1) \]

• Under this model, the outputs are also linear models, whose expected value, variance, other moments may be calculated easily if the output Taylor coefficients are known

• AD computes 1st order Taylor coefficients of the output.
Pros and Cons of UQ via Taylor

• For ‘small’ uncertainty, 1st order Taylor much faster than Monte Carlo \( + \rightarrow \) improve efficiency of UQ process sometimes

• ‘Small’ is not necessarily known –

• Take higher order terms in Taylor series improves accuracy and applicability, but may not be a win
Other Interesting Stuff

• Intervals
  — Been around a long time, varying degrees of acceptance
  — Interval Newton for global opt

• Running error bounds
  — A Wilkenson idea --- insert code that tracks roundoff error
  — Not derivative code, but similar

• Taylor models for classic Forward, backward error analysis

• Probability Distributions
  — Alternative to Monte Carlo