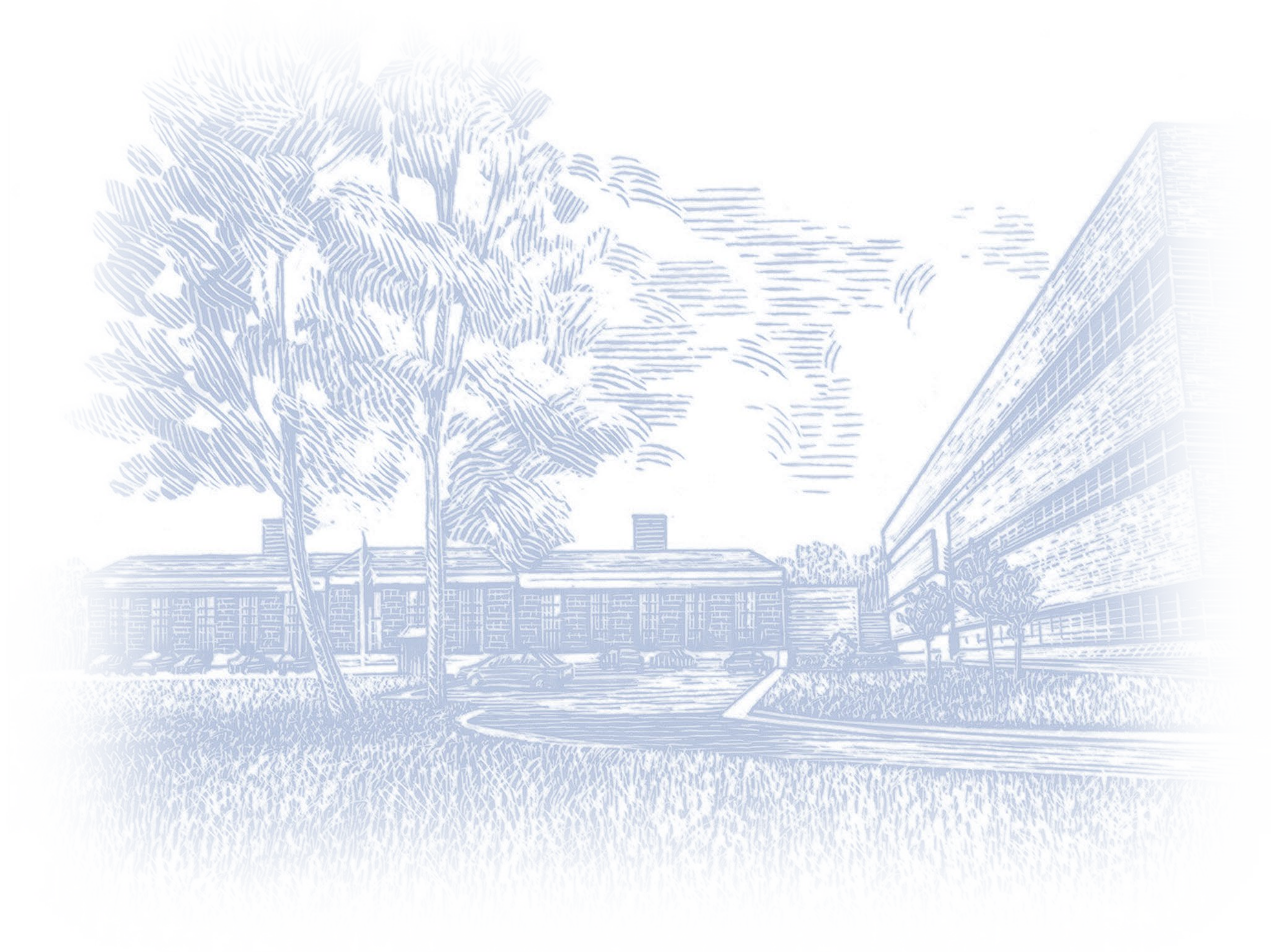
Producing Correct Software

Publication No.: N/A **1999**



U.S. Department of Transportation
Federal Highway Administration

Federal Highway Administration
Turner-Fairbank Highway Research Center
6300 Georgetown Pike, McLean, VA 22101

Foreword

James A Wentworth
Advanced Research Team
Federal Highway Administration
Rodger Knaus
Instant Recall, Inc.

Greetings:

I have a strong conviction that the engineering community and the software development community are equally in the dark when it comes to software testing and evaluation (T/E). Nobody knows how to do it, so nobody requires it; since nobody requires it, nobody does it; since nobody does it, nobody knows how to do it; since nobody knows how to do it...[Green and Keys, 1987].

It is of concern that as we get into more and more complex software development and applications (e.g. advanced/autonomous traffic control strategies) that both the current philosophies of software development and software T/E must change or we in the highway business could potentially have our own Cali and Guam crashes (the August 1996 fatal airline crash in Cali, Columbia caused in part by software deficiencies and the August 1997 airline crash in Guam most probably caused by a failure in air traffic control software). New programming techniques and new chip manufacturing--advancements of OOPS, intelligent software and the coming "systems-on-a-chip" [in the next generation of systems chips most circuit nodes will be inaccessible from the outside and will therefore be untestable. Built in self-test will be heavily relied on. If you don’t design it right before you build it you are in big trouble.]--will change software development and drastically complicate software T/E.

As a very preliminary step in addressing this problem, a draft document on **PRODUCING CORRECT SOFTWARE** has been prepared. The purpose of this communication is not to present an official document, but to share a work in process and to solicit advice. It is the intention of the Development Team to produce a quality product that will truly be of value to those developing and testing software. Thus I encourage you to critically review this incomplete draft document and provide any suggestions for improvement and expansion. We want to hear the bad news as well as the good so that we can improve this document.

It will be impossible to respond directly to every comment, but be assured that every comment will be reviewed and given the consideration it deserves. Thank you in advance for your consideration.

Sincerely yours,

James A. Wentworth
Chief, Advanced Research Team
Office of Safety and Traffic Operations R&D
Federal Highway Administration

This web page accomplishes the following:

- Defines software correctness.
- Suggests a list of concrete tasks and problems, which, particularly when used together, can improve software correctness.
- Introduces techniques to solve these problems.

More detailed information about techniques introduced here will appear later in subdocuments linked to this document.

Table of Contents

Technical Report Documentation Page

Introduction

Software development is a relatively new activity being utilized by an ancient profession. Construction and roadway engineering began in prehistoric times, and over time the industry has raised the standards in design, construction, and the documentation of practice. Through modernizing and improving design, construction, and maintenance, new approaches and technologies have been incorporated into civil engineering practice. Many of the new tools and technologies did not initially achieve the levels of reliability and standardization that the civil engineering profession demanded. Regrettably, software development and computer programs fall into this category.

Software planning and development should emulate construction project planning, design, and construction, integrating testing and evaluation. The end result will be more reliable software and more reliable transportation systems.

Software developers must use tools for improving software and catching design problems at an early stage of the software development life cycle, when fixing those problems is relatively cheap and easy. These tools must be easy to use for both the engineer responsible for software development and for the software developer. The software should not be designed for just those individuals with unusual mathematical training.

In traditional software engineering, developers claim that testing is an integral part of the design and development process. However, as programming techniques become more and more advanced and complex, there is little consensus on what testing is necessary or how to perform it. Furthermore, many of the procedures that have been developed for verification and validation (V&V) are so poorly documented that only the originator can reproduce the procedures. The complexity and uncertainty of these procedures has led to the inadequate testing of software systems (even operational systems). As software becomes more complex, it becomes more difficult to produce correct software¾ and the penalties for errors will increase.

Definition of Software Correctness

This document is an introduction to some techniques for producing correct software. Correct software must accomplish the following:

- G1: Compute accurate results.
- G2: Operate safely, and causes the system containing the software to operate safely.
- G3: Perform the tasks required by the system containing the software, as explained in the software specifications.
- G4: Achieve these goals for all inputs.
- G5: Recognize inputs outside its domain.

These goals will be referred to collectively as "software correctness," and achieving these goals will be referred to as "achieving correctness (in software)" or as the "correctness problem."

Common Elements for All Correctness Problems

Designing and building software is a branch of engineering. As such, the following is true:

- Scientific and engineering knowledge is used to produce designs that satisfy the requirements G1-G5.
- Theory is used to check the completed design.
- Like other creations of engineering, the end product (the software) must be tested before it is known to work.

As with other designed objects, having completely reliable computer software is not possible with current technology. However, there are some techniques that engineers in other areas use to increase reliability in general, which appear likely to increase software reliability:

- Employing careful management and work practices during design and construction.
- Using theory to check completed designs.
- Designing alarms and safety devices into the deployed device.

Most current software standards have focused on management and work on software projects rather than focusing on whether the software will work safely and reliably. Management and work standards are valuable, but these standards by themselves do not guarantee correct software for the following reasons:

- It is impossible to enforce the work standards 100 percent of the time.
- Software developers make mistakes even when following the standards.

Therefore, software developers need to accomplish the following:

- Theoretically establish correctness of the currently built program (usually assuming that underlying hardware and support software is correct).
- Make software self-checking and self-correcting.

These topics are presented in subpages:

- Formal methods to establish correctness (under development).
- Making software self-checking and self-correcting.

Correctness Given Mathematical Specifications

Summary

This web page presents ways to achieve software correctness for software that has, or could have, mathematically precise specifications. More detailed information about techniques introduced here will appear later in subdocuments linked to this document.

Definition of Software Correctness

This document is an introduction to some techniques for producing correct software. Correct software must accomplish the following:

- G1: Compute accurate results.
- G2: Operate safely and cause the system containing the software to operate safely.
- G3: Perform the tasks required by the system containing the software, as specified in the software specifications.
- G4: Achieve these goals for all inputs.
- G5: Recognize inputs outside its domain.

Software Techniques to Produce Correctness

G1 through G5 describe the desired outcome of a software project but do not explain how to reach that outcome. By following three tasks, G1 through G5 can be achieved:

- T1: Determine specification that is sufficient in the mathematical sense for G1-G5.
- T2: Build the software.
- T3: Demonstrate that the software satisfies the specifications from T1. This task is called verification and validation (V&V).

Suggested techniques for addressing T3 (showing that software meets specifications) will be included in the following sections. This document will not address T1 (specifications) and T2 (system development). Although T1 and T2 are crucial to the success of the project, these topics are addressed in other sections.

1. Systems analysis.
2. Knowledge engineering.
3. Safety engineering, e.g., see Leveson [references] for several techniques for producing specifications that imply system safety.
4. Software standards such as ISO 9001 and SEI CMM [see Hatton, p. 16-23 in refs.] focus on the software development process (T2).
5. Demonstrating that software satisfies specifications (T3) requires techniques not found in development process (T2) oriented standards (as discussed below).

On the other hand, while there has been considerable academic work on techniques for V&V, this work has two flaws:

1. It does not address all the problems faced by real-world software developers
2. Theory is used to check the completed design.
3. It is not in a form that is readily usable by actual software developers.

Process Standards Are Insufficient

A process standard is insufficient to insure software correctness (G1-G5). This is because the behavior of a system depends on the state of the system at the time when it is used, not the history of its construction. If a process standard were used in civil engineering, one would require that a bridge be designed by a licensed professional engineer, and built by certified journeyman craftspersons, but one would not be required to do a structural analysis of the design nor a physical inspection of the bridge itself. Admittedly, it is more likely that quality personnel following careful procedures will produce a correct bridge or system; however, even the best professionals make mistakes, and less qualified personnel may produce a sound bridge or system. The built system itself determines its behavior.

Top Level Subtasks of Verification and Validation

Like the requirements for correctness (G1-G5), the definition of verification and validation (V&V) does not indicate how it should be achieved. The following list of V&V subtasks breaks the V&V problem into parts that are more manageable:

1. VV1: Show that the software satisfies the desired conditions if the computation was performed in an ideal computer with infinite memory and precision.
2. VV2: Show that the actual computation approximates the ideal computation for inputs within the domain of application.
3. VV3: Show that the actual computation can identify inputs or computations for which the actual computation does not approximate the ideal computation of VV1.

The Best Situation

The best situation is when the software is 100 percent correct. This is the case when T1 and VV1 through VV3 can be carried out. Together the two requirements, finding sufficient specifications (T1) and showing that the software satisfies them (T3), would imply that G1 through G4 are satisfied. Together VV1 (correctness on an ideal computer) and VV2 (actual approximation in the ideal computer), imply T3. VV3 is a more elaborate restatement of G5. Therefore, in the ideal case when T1 and VV1 through VV3 are achieved, theory suggests that the correctness conditions (G1 to G5) should always be satisfied. Note that in the section "Testing" below, it is strongly suggested that all theoretical predictions about software, including this one, be experimentally verified.

Computations Using Formulas

Software modules (e.g., functions in C) that plug numbers into a formula to compute a result are an important special example of VV1-VV3 goals being achieved. This happens when the following occurs:

1. The required condition on the output is given by the formula.
2. In most current programming languages, the formula appears directly in the program. The version in the program is an image of the original formula under a syntactic mapping that preserves the semantics of the original.
3. The semantics of the programming language are such that if the formula were computed on an ideal computer, the result would satisfy the formula.

Therefore, to achieve correctness for programs that are based on formulas, it is sufficient to write the program in the following way:

1. All input data used in the computation approximate the corresponding ideal mathematical representations of the corresponding input data items.

2. Each computation approximates the computation on an ideal computer.

If these two conditions are satisfied, and mathematical induction on the number of steps in the computation has been performed, the final results will approximate the ideal. However, in the real world, the closeness of approximation will decay as the number of steps increases. Therefore, it is necessary to show that the final result approximates the ideal computation closely enough for its intended application. Methods for writing numerical software that approximate the ideal on a domain that the software itself can recognize (VV2 and VV3) will be discussed in Safe Numerical Techniques [not yet available].

Software with Algorithmic Specifications

Sometimes software specifications, as referred to in VV1, are stated as an algorithm. Such an algorithm is usually stated in some combination of natural language, mathematics, and/or pseudo-code. For example, Euclid uses this method by stating an algorithm for finding the greatest common divisor. Euclid's algorithm can be used as the specifications for a computer program to find the greatest common divisor. Euclid proved this algorithm correct and set a standard over 2000 years old, which is rarely matched by contemporary programmers.

For software with algorithmic specifications, VV1 is established by showing that the computerized version of the algorithm is equivalent to the specification version of the algorithm. Two algorithms are equivalent if they compute the same function, i.e., for any point in the input domain of the specification algorithm, the specification and implementation algorithms compute the same value(s).

One important way to do this, although not the only way, is to show that the computer implementation is an image of the original algorithm under a mapping that translates the syntax of the original into a programming language, while preserving the semantics of the original.

Once VV1 is established, VV2 and VV3 can be established using the same techniques used in formula-based programs.

Software with Logical Specifications

Often the specifications of VV1 are stated as logical formulas containing both inputs and outputs. The built software is supposed to satisfy these formulas.

One source of these formulas is a hazard analysis (see Ch. 14, Safeware). Another source is the application area for which the software is intended (e.g., knowledge about pavements for a pavement management system). A third source is basic science and mathematics. For example, a computed inverse matrix should satisfy the following formula of abstract algebra defining an inverse:

1. A*B=B*A=I where
    - A is the input matrix
    - B is the output matrix
    - I is the identity matrix

When logical specifications exist, there are two ways to establish them: wrapping and mathematical proof.

Wrapping

Wrapping is a technique that uses a modified version of a piece of software to compute its own correctness. To wrap a program P used in system S with a requirement R on P,

1. R is translated (using the techniques for translating formulas) into an equivalent representation R' in the programming language of P.
2. R' is inserted at the end of P, right before P returns its value(s). This insertion is done in such a way that the calling context of P can examine the result of R. The modified version of P will be called P'.
3. S is modified to
    a. Test the value of R'.
    b. Discard the values from P' if R' is FALSE.

For example, let the following be a matrix inverse program that puts what it thinks the inverse of A into B:

```
void inverse (matrix A, *matrix B)
```

A wrapped version of this is

```
boole wrapped_inverse(matrix A, *matrix B)

{
matrix X, Y;
boole result;
inverse(A,B);
matrix_mul(A,B,&X);
matrix_mul(A,B,&Y);

result = matrix_eq( X, I) && matrix_eq(Y, I);
return (result);
}
```

where

1. boole matrix_eq(matrix A, matrix B) is a test for matrix equality
2. void matrix_mul(matrix A, matrix B, *matrix X) puts the product of matrices into X.
3. I is the identity matrix (of the same number of dimensions as A and B)

The other half of wrapping the matrix inverse is to make the program where it is used respond to the success or failure of the inverse program.

Let

inverse (A,&B); F(B);

be a call to the matrix inverse program, where A and B are matrix variables, and F(B) is a code segment that uses B.

A wrapped version of this is

```
boole1 = wrapped_inverse (A,&B);
```

if (boole1) F(B);

else G;

where boole1 is a boolean variable and G is a code segment that does not use G.

Wrapping has several different V&V applications:

1. Wrapping a software module with a logical formula specification
2. Wrapping a software module with necessary but not sufficient conditions for correctness that the output must satisfy. This is useful when sufficient conditions for correctness are not available, e.g., with software that predicts values that have not yet been observed.
3. Wrapping inputs to all programs to insure that the inputs are in the domain for which the program is intended.

The following is an important theorem for wrapping:

Let PROG be a program on which P(x) is to be proved. P(x) is a logical formula such that

1. x is the only computed result in P(x).
2. All functions appearing in P(x) are continuous.
3. All computations in P(x) approximate ideal-computer computations when inputs are in domain D.
4. PROG detects when inputs are outside D.

Then if P(x) is approximately satisfied, P(x) is true for PROG.

Mathematical Proofs About Programs

Mathematical proofs about programs are sometimes useful for showing the correctness of programs with logical specifications. Verifying the correctness of a computation typically involves two steps:

1. Showing that the computation is correct on an ideal computer.
2. Showing that the ideal computation is approximated on an actual computer.

To address the first step, one typically defines an ideal computer as a mathematical construct capable of running the program to be proved correct. The ideal computer often consists of a set of states and transitions, defined to mirror the semantics of a programming language. Program execution is simulated by transitions to new states in this ideal machine. A typical proof shows that for any inputs in the legal input domain, all computational paths reach end states where the program specifications are true.

Verification & Validation for Prediction Software

Predicting future or hypothetical events is an important application of computers, e.g., traffic simulation is used to predict traffic flows during the next hour. When the domain for the predictions has a precise theory by which predictions are made and the predictions have been extensively confirmed by experiment, the techniques for verifying software with algorithmic or logical specifications can be applied to the prediction software.

However, many prediction software programs make predictions when theory, past experimental verification or both is missing. An example of this is prediction using a neural net. The following describes many neural net applications:

- Nothing much is known about the function to be predicted.
- The neural net uses an algorithm unrelated to the domain of application.
- The domain on which predictions are to be made is poorly defined.

When there is no precise, experimentally verified predictive theory, the only way to establish the correctness of prediction software is by experimentally verifying that the software predicts accurately. In addition, the reliability of predictive software, particularly when experimentally verified theory is lacking, can be improved in these ways:

- Wrapping the prediction software with tests on necessary conditions the prediction must satisfy (e.g., that the traffic flow not exceed the physical capacity of a road).
- Combining several different prediction methods (e.g., by averaging or polling). Testing Even when there is a theoretical proof using the methods described above, a program should be statistically tested because there can be errors:
    - In the proofs.
    - In the wrapping code.
    - In details beneath the level covered by the proofs or wrapping, e.g., hardware errors

One must test the hypothesis of whether the program satisfies a logical formula. The logical formula does not have to be proved or used for wrapping. For example, for prediction programs, one must experimentally verify whether the formula prediction comes close to the observed values.

If P is the logical formula to be verified, one designs an experiment to statistically test the hypothesis that P is satisfied by outputs of the software. To design this experiment, one must choose a confidence level (C) through which one wants to experimentally verify P. The traditional scientific values for C are 95 percent, 99 percent, or perhaps 99.9 percent. However for safety-critical computations, a higher confidence level, e.g., 99.9999 percent (only one failure in a million) may be appropriate.

The result of testing is an error estimate, or an estimate of the difference between computed and observed values. This error estimate should only be applied to new data in the population sample. Furthermore, as the software runs on additional data, the original error estimate can be tested against results on these new runs. In addition, it is sometimes possible to improve the original error estimate by adding the new data points to the original sample, creating a new, larger, more reliable sample from which a more reliable error estimate can be computed.

In addition, running the experiment produces an observed confidence level C', which is almost never exactly C (although C' may be close to C). Designing the experiment to achieve C does not cause the confidence level to be C, only that there are enough data points in the experiment to possibly achieve C.

Producing Reliable Software – Other References

Here is a list of useful references for developing high reliability software.

- *Safeware, System Safety and Computers*, by Prof. Nancy Leveson [Addison Wesley, 1995, 0-201-11972-2]. This is a truly excellent book. It covers the following topics:
    - How software safety relates to system safety.
    - An introduction to safety and system engineering for software developers.
    - The Safeware lifecycle.
    - Techniques for hazard analysis.
    - Developing specifications for safety-critical software.
- Handbook for Verification, Validation and Evaluation of Expert Systems, Advanced Research Team, Office of Traffic and Safety Operations, Turner-Fairbank Highway Research Lab, Federal Highway Administration, U.S. Dept. of Transportation, 1995. This is the most complete reference to date on these topics:
    - Verification and validation of existing expert systems
    - How to develop high reliability expert systems.

[Note: Rodger Knaus, president of Instant Recall, is one of the authors of this handbook.]

- *Fatal Defect* by Ivars Peterson [Times Books/Random House, 1995, 0-8129-2023-6.] This is a very readable history of software and hardware bugs and the problems caused by them. This book is written for the nontechnical audience, but also contains a lot of interest for computer professionals:
    - A very complete list of references.
    - Analyses of a number of computer-caused accidents.
    - Well-presented discussions of attempts to manage hardware and software problems.
- Landauer, Christopher and Kirstie L. Bellman. "Constructed Complex Systems: Issues, Architectures and Wrappings", pp. 233-38 in *Proceedings EMCSR 96: Thirteenth European Meeting on Cybernetics and Systems Research, Symposium on Complex Systems Analysis and Design*, 9-12 April 1996, Vienna.
    - This is a paper about wrapping by the inventors of the concept.
    -

Verification And Validation For Prediction Software

Summary

This web page accomplishes the following:

- Defines a method for finding specifications for predictive software, including simulations.
- Describes a method for wrapping a simulation.
- Describes the wrappings that users of simulation or other predictive output should have on inputs from predictive programs.

Introduction

Predicting future or hypothetical events is an important application for computers. Computers are used, for example, to predict future traffic flows. When the domain has a precise theory by which predictions are made and the predictions have been extensively confirmed by experiment, one can apply the techniques for verifying software with algorithmic or logical specifications to the prediction software.

However, many prediction software programs make predictions when theory, past experimental verification, or both is missing. An example of this is prediction using a neural net. The following describes many of these applications:

- Nothing much is known about the function to be predicted.
- The neural net uses an algorithm unrelated to the domain of application.
- The domain on which predictions are to be made is poorly defined.

When there is no precise, experimentally verified predictive theory, the only way to establish the correctness of prediction software is by experimentally verifying that the software predicts accurately.

In addition, the reliability of predictive software, particularly when experimentally verified theory is lacking, can be improved by doing the following:

- Wrapping the prediction software with tests on necessary conditions the prediction must satisfy (e.g., that the traffic flow not exceed the physical capacity of a road).
- Combining several different prediction methods, e.g., by averaging or polling.

The Required Level of Accuracy

For most predictive programs, it is known that the predictions made are not completely accurate because of the following causes, among others:

- The predictive programs use measurements containing errors.
- The predictive algorithms are approximations to reality that are not completely true in the real world.
- Both the prediction and the real-world variables being predicted are stochastic processes.

However, a prediction may be accurate enough for a given application, depending on the particular application. The application determines whether the prediction is accurate enough, but once a tolerance in the application domain is established, that tolerance can be changed to a required tolerance on that prediction.

Many decisions based on predictions depend on knowing the value of a function in another prediction. For example, finding the best ratio of green time for a freeway ramp meter, based on simulated traffic flows given traffic volume and road geometry, can be solved by finding the minimum waiting time for various green time ratios. If the waiting time function is accurately established from the simulation, one can find the approximate minimum of the function in the real world and make a sound traffic planning decision.

Let f be a real-world function of interest to a user of a prediction program P. Generally f depends on a number of variables:

- x1,...,xn, input variables both to f and P, X in vector notation.
- y1,...,ym, input variables to f and outputs of P, Y in vector notation.
- z1,...,zq, variables not used or predicted by P, Z in vector notation; these variables are not used in the following discussion and will sometimes be omitted in functional notation for f and P.

For the y variables, P(yi) will denote the predicted value of yi, and R(yi) will denote the actual, real-world value. Likewise Rf and Pf will denote values of f based on predictions or real-world data.

The user wants the real-world value of f, Rf(X,Y,Z), which is the value of f computed at real-world values of the input variables, i.e.,

Rf(X,Y,Z) = f(X,RY,Z)

What the user actually gets from the prediction program is Pf, the value of f at the data point generated by the prediction program, i.e.,

Pf(X,Y,Z) = f(X,PY,Z)

If Pf is close enough to Rf to be within the application's tolerance for f, Pf can be used in place of Rf. The problem for the user of P is to know when P is within the application tolerance.

One approach to this is to estimate the errors in f using a first order Taylor series approximation to R. Let Rf-Pf = E, the error of the prediction. To a first order approximation,

Rf = Pf + SUM df/dyi*ei, for 1<= i <=m.

where df/dyi is the partial derivative of f with respect to yi. From this equation one sees that Pf is a good approximation of Rf when the following conditions are met:

- f does not depend much on variables where P is a bad approximation of R; mathematically this means that if df/dyi is small enough, the ith term of the sum may be within the tolerance even when ei is fairly large.
- P is a good prediction on the variables on which f substantially depends.

Wrapping a Prediction to Maintain Tolerance

The above formula provides a method for an application to wrap the input from a prediction program. P can be used for estimating f when fabs(f(P) - f(P+E)) < T, or when approximately SUM fabs(df/dyi)*fabs(ei) < T, if the following occurs:

- The acceptable tolerance T of f is known for the application.
- There is a good estimate of the error E = Rf . Pf.
- The partial derivatives df/dyi are known.

If the errors in estimating the yi's are known, the last inequality lets one estimate if P is useful for estimating f.

To wrap the prediction program and the application so that P is only used when it provides a close enough estimate of f, one can do the following:

- Equip P with a wrapping that computes an estimate of E, ideally with a confidence interval around the point value.
- Equip the application with procedures for doing at least one of the following:
    - Test to insure that fabs(f(P) - f(P+E)).
    - Test to insure that SUM fabs(df/dyi)*fabs(ei) < T.

Estimating the Error of a Prediction

To estimate the error of a prediction program P, one can compute the sample mean and variance for the errors of P for points in the following instances:

- Both the inputs and outputs of P were measured.
- The inputs of the sample points are close to the input for the point where the error is to be estimated.

The generalized regression neural net provides a procedure for drawing such a local sample from a large database and for computing a sample mean and variance for the error.

Handling Uncertainties in P and E

Many prediction programs, particularly simulations, use random variables, so that the value of each yi on a particular run is a random variable. Also, the error in P will generally only be known until it reaches some accuracy represented by a sample variance. To insure that these errors do not exceed the tolerance for f, the application program should determine the region for possible values of Y+E and that the change in f in that region is within the tolerance.

Applying Experimental Results

Summary

This web page presents techniques for applying a database of previously experimentally measured input-output pairs to test the correctness of software.

The Problem

Experimental values can be used to wrap a computer program. Each data record in the set of experimental values should contain the following:

- Measurements of the inputs used by the program.
- Measurements of the program outputs that occurred for those inputs.

When the program is asked to compute the outputs at a new point, the known experimental values are used to predict the outputs, using some interpolation techniques. Only rarely is a record for the new input point already in the database; usually known experimental results for inputs closest to the new points have to be combined to make an experimentally based prediction of the program output.

If the output of the program agrees with the prediction based on experimental data within an acceptable tolerance, the results of the program are accepted. Otherwise the results of the program are rejected, i.e., not used without further examination, perhaps by alerting an expert to the discrepancy between the experimentally predicted value and computed value.

To wrap a program with predictions based on observations, the following problems must be addressed:

- What interpolation technique should be used to derive the experimentally based prediction?
- How can one decide among possible interpolation techniques?

Some Interpolation Techniques

Note: This section is under development.

- Local Sampling Techniques
- Generalized Regression Neural Nets
- Linear Regression
- Polynomial Regression
- Other Regression Techniques
- Backpropagation/Feedforward Neural Nets
- Projection Pursuit Regression

Deciding Between Interpolation Techniques

Notation

- x, xi, xj, etc. are points in input space
- ex(x) = experimental output at x
- p(x), pi(x) etc. are predicted outputs at x

- S = set of input points in experimental database. S excludes any point used in creating any interpolation function currently being studied
- N = cardinality of S
- err(x), erri(x) etc. are errors in predicted outputs, i.e., err(x) = ex(x)-p(x). If we need to distinguish errors of different prediction functions, we will use the notation err(p,x).
- abs_err(x), abs_erri(x) etc. are errors in predicted outputs, i.e., abs_err(x) = abs(ex(x)-p(x)), the absolute value of the error
- mean({f(x)|x in S}): This is the mean of the function f on x
- var({f(x)|x in S}) = var({f(x)|x in S}): This is the variance of f on the sample S.

Comparing Estimators

Given a prediction function, p(x), one measures the error in the prediction when compared to measured experimental values. This error function is abs_err(x), the absolute value of the error. [Taking the absolute value prevents positive and negative errors from canceling each other out in the mean and other statistics.]

For each sample test point x in S, one can obtain abs_err(pa,x) and abs_err(pb,x) from the measured experimental value ex(x) and the predictions pa(x) and pb(x). The sets {abs_err(pa,x)|x in S} and {abs_err(pb,x)|x in S} are matched pair data, i.e., measurements of two random variables are made at the same sample points. One can compare functions on a matched pair sample by looking at the difference of the functions on the sample, i.e.,

{err_diff(x) | x in S}, where

err_diff(x) = abs_err(pb,x) - abs_err(pa,x).

The mean of err_diff on S is a measure of how much better pb predicts than pa predicts. If this mean is positive, pb is a better predictor of ex than pa. If, on the other hand, mean(err_diff) is nonpositive, pb is not a better predictor than pa. Therefor e, to test whether pb is a better predictor, one will find a confidence level for mean(err_diff) to be positive.

In the usual method of statistics, one finds the confidence interval for mean(err_diff)0 by determining the probability of an observed statistic obtaining at least its observed value if the desired conclusion (i.e., pb is better than pa, or in statistica l terms, mean(err_diff)0) is false. The sample mean follows the t distribution centered around the population mean. The integral of the t distribution with mean 0 from the observed value on S to infinity is the probability of obtaining the observed diffe rence of means if the predictor that was supposed to be worse is actually better or at least no worse than the supposed better predictor.

The t statistic with mean 0 is given by the formula

t = mean(err_diff on S)/sd(err_diff on S)

sd(err_diff) = (SUM(err_diff(xi)-mean(err_diff on S)^2/(N-1))^(1/2)

for xi in S. [Note that a better computational formula is sd = (N*SUM(err_diff(xi))^2 - SUM(err_diff(xi)^2/(N*(N-1)))^(1/2).]

The confidence that the expected predictor is better than the other predictor is given by the probability that t has a value less than or equal to the observed value. If t0 is the observed value of t, the confidence is the integral of t from infinity to t0. Standard tables of the t distribution in most statistics books give values of t0 which guarantee standard confidences, e.g., 90 percent, 95 percent, 99 percent, etc. By comparing the value of t0 on S with the values in the table, one can determine w hich of the standard confidence levels has been reached. The tables are for the t distribution with a standard deviation of 1; dividing the sample mean by the sample deviation converts the sample mean to this standard form.

Use of Absolute Values

From inspection of a table for the t distribution, it becomes apparent that the confidence level increases as the value of t increases. In turn, t increases as mean(err_diff) increases. Suppose that the error of a predictor was simply

predicted value - observed value

instead of the absolute value of that difference. If the worse predictor badly underpredicts a large positive observed value, while the better predictor has only a small error, err_diff would be a negative number. The t test only returns useful confidenc e intervals for t values near 2. Without taking absolute values of predictor errors, the t test can fail to detect significant differences in the magnitude of errors in predictors.

Example

Given

Target	1.1	2.2	3.3	4.4	5.5	6.4	7.6	8.9
Better	1.11	2.18	3.33	4.41	5.59	6.62	7.67	8.81
Worse	1.11	1.95	3.03	4.44	4.80	6.31	8.01	9.11
Err_diff	0.00	0.23	0.24	0.03	0.61	-0.13	0.34	0.120

where

- Numbers listed under "Measured" are the observed experimental values.
- Numbers listed under "Better" are the presumed better predictions.
- Numbers listed under "Worse" are the presumed worse predictions.
- "Err_diff" is the difference of absolute errors, i.e.,
- err_diff(x) = abs_err(worse,x) - abs_err(better,x)
- = abs(ex(x)-worse(x)) - abs(ex(x)-better(x))

As a result, statistics for err_diff on the 5 sample points are

- mean = 0.180000
- variance = 0.053028
- standard deviation (s_d) = 0.230279
- t = mean/s_d = 0.781661
- degrees of freedom = 7
- confidence level (that the presumed better predictor really is better) = a little better than 75 percent

Properties of an Improved Approximation

Note that the level of confidence in whether p1 improves p2 depends only on the t statistic, the ratio of the difference of means of the absolute errors divided by the standard error of this difference. This means that the uniformity by which a predictio n function predicts beyond its intended domain is a very desirable property for a prediction function, because it decreases the standard error of the error of the predictor, increasing the value of the t statistic.

Using the Normal Approximation

As the sample size increases, the t distribution approaches the normal distribution. The center of the distribution converges faster than the tail. For 30° of freedom, the value of t required for a 90 percent confidence level is about 2 percent greater t han the corresponding value of the normal distribution (1.310 vs. 1.282) while the value of t for 99.5 percent confidence is about 6 percent greater (2.750 vs. 2.576).