SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Causal Inference Based Fault Localization for Numerical Software with NUMFL

Zhuofu Bai, Gang Shu, Andy Podgurski*

Department of Electrical Engineering and Computer Science Case Western Reserve University Cleveland, OH 44106

SUMMARY

We present NUMFL, a value-based causal inference model for localizing faults in numerical software. NUMFL combines causal and statistical analyses to characterize the causal effects of individual numerical expressions on output errors. Given value-profiles for an expression's variables, NUMFL uses generalized propensity scores (GPSs) or covariate balancing propensity scores (CBPSs) to reduce confounding bias caused by evaluation of other, faulty expressions. It estimates the average failure-causing effect of an expression using quadratic regression models fit within GPS or CBPS subclasses. We report on an empirical evaluation of NUMFL involving components from four Java numerical libraries, in which it was compared to five alternative statistical fault localization metrics. The results indicate that NUMFL is the most effective technique overall. We also found that NUMFL works fairly well with data from failing runs alone. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: value-based statistical fault localization; causal inference; confounding bias; generalized propensity score; covariate-balancing propensity score; failure-causing effect estimation

1. INTRODUCTION

Numerical software plays a very important role in science, industry, and defense, and failures of numerical software have been reported as the cause of several well-publicized "disasters" [1, 2]. However, automated techniques intended specifically for localizing numerical faults based on execution data have received relatively little attention from researchers, although there has been substantial research on general *statistical fault localization* (SFL) techniques (e.g. [3, 31, 5]) and on other general automated debugging techniques (see Section 6).

Typical SFL techniques take data characterizing a set of both passing and failing program executions, including PASS/FAIL labels (provided by testers or end users) and recorded profiles of internal program dynamics, and they compute statistical measures of the strength of the association, if any, between the occurrence of software failures and the occurrence of certain runtime events at particular program locations. These measures are then used to help guide the search for the causes of observed failures, typically by ranking program statements by the strengths of their associations with failures. When SFL techniques are evaluated they are generally used as the sole source of information about possible fault locations. However, it seems more realistic to envision them ultimately being used in combination with other sources of information, such as programmer hunches and *fault prediction models* [6] based on static code properties and project history. Potentially, SFL techniques provide a relatively inexpensive way to maximize the information obtained by testing.

*Correspondence to: Department of Electrical Engineering and Computer Science Case Western Reserve University Cleveland, OH 44106

In the vast majority of research on SFL, execution dynamics have been characterized either purely by *code-coverage profiles* (also called "spectra"), indicating which statements, branches, or paths were covered by each execution (e.g., [3, 47]), or by indicators of the outcomes of *predicates* (e.g., [31]) both existing branch predicates and predicates inserted specifically to enhance SFL (e.g., ones that compare the values of numeric variables to zero). Although the outcomes of such predicates reflect the *values* of program variables, they may do so inadequately for SFL, e.g., because predicates or conditions are omitted mistakenly, because they are hidden in library code or microcode, or because few predicates are actually required in the program. These facts make typical SFL techniques *poorly suited* for localizing faults in numerical programs and subprograms having relatively few conditional branches. Even predicates inserted in a program to enhance SFL are likely to characterize the values of numeric variables inadequately, unless application-specific information is available that indicates which predicates should be inserted.

Although purely numerical programs, and numerical components of other programs, are often relatively small, the intricacy of their computations can make them very difficult to debug manually. In order to provide automated fault localization assistance to developers who need to debug numerical programs or components, we present a new approach to SFL, denoted by *NUMFL*, that is designed specifically to localize faults in numerical code. NUMFL, like some other recent work on SFL [7, 8, 9, 10], is based on *causal inference methodology* [11], which seeks to mitigate biases like confounding bias that can badly distort SFL scores used to guide the search for faults. However, in contrast to previous work, NUMFL is based on generalized forms of *propensity scores* [12, 41], which are a fairly recent development in causal inference that provide a means of reducing confounding bias when estimating the causal effect of a numeric *treatment* or *exposure* variable on an outcome variable.

In this paper, each value of the "treatment" variable T_e is the result of evaluating a numerical expression or subexpression with a *unique program location* identified by e, and the outcome variable Y represents the error (possibly zero) in the output of the numerical program or component under consideration. The causal effect of T_e on Y is characterized by a function $r_e(t)$, which in medical research is called a *dose-response function* (DRF) [13]. To enable localization of faults, NUMFL summarizes the behavior of $r_e(t)$ over a set of evaluations of expression e, with a single quantity $susp(e)$ called a *suspiciousness score*. It employs generalized propensity scores to make it less likely that a correct expression will receive a high suspiciousness score because it used variables whose values were erroneous due to faults in *other* expressions. Note that we do *not* use propensity scores themselves as suspiciousness scores; the two kinds of scores have very different purposes.

We assume that (in most cases) users of NUMFL have a set of program or component inputs that induces multiple failing and multiple passing runs. For these inputs, NUMFL requires corresponding *expected outputs* (not just PASS/FAIL labels) and execution profiles characterizing variable values. The expected outputs are provided by testers or end-users, and the profiles are recorded automatically by program instrumentation. To compute the suspiciousness score $susp(e)$, NUMFL requires, in addition to values of the treatment variable T_e and the output-error variable Y, corresponding values of the set of variables *used* in expression e. Although the value-profiles used by NUMFL are more detailed than basic coverage profiles used by typical SFL techniques, the potential cost of recording and analyzing value profiles is offset by the relatively small size of many numerical programs. Moreover, we assume that developer-time is a more critical resource in debugging than is computation time, because developers usually can do other work while an SFL algorithm runs.

The main contributions of this paper are:

- NUMFL, a new, value-based approach to SFL for numerical programs, which is based on sound causal inference methodology
- A new, value-based approach to SFL for numerical programs, named NUMFL, that is based on sound causal inference methodology, and two variants of NUML, denoted NUMFL-GPS and NUMFL-CBPS, based on alternative forms of generalized propensity scores
- The first use of generalized propensity scores to reduce confounding bias in SFL
- An implemented platform to profile the variables used and defined by numerical expressions in Java programs

DOI: 10.1002/stvr

- An empirical comparison of NUMFL to five competing SFL techniques on components from four widely-used Java numerical libraries, which indicates that NUMFL is more effective overall than the other techniques
- Empirical evaluations of NUMFL on programs with single faults, with multiple faults, with both passing and failing executions, and with only failing executions
- An empirical comparison of NUMFL-GPS and NUMFL-CBPS to each other

The present paper is a revised and extended version of [45], which presented NUMFL-GPS, but not NUMFL-CBPS, and which evaluated NUMFL-GPS on only single-fault programs and with a mixture of passing and failing executions. The new contributions of this paper, besides NUMFL-CBPS, include the evaluation of NUMFL with multi-fault programs, its evaluation with only failing executions, and the empirical comparison of NUMFL-GPS and NUMFL-CBPS.

The rest of the paper is organized as follows: we present a motivating example in Section 2; background for our approach is presented in Section 3; NUMFL is described in detail in Section 4; we report on its empirical evaluation in Section 5; related work is surveyed in Section 6; and Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

Figure 1 shows a short function *harmean*, which correctly calculates the harmonic mean of two floating point numbers, in the middle column and shows a faulty version of *harmean* in the rightmost column. (In real fault localization scenarios only the faulty code is usually available.) We injected a fault into statement s_3 by adding a floating point number c to the original expression $a * b$. Given a pair of inputs (a, b), the faulty version of s_3 produces an erroneous value for the variable x. This value may propagate to statement s_8 and, via the variable r, cause an incorrect return value at statement s_9. We use $r_{correct}$ and r_{faulty} to denote the return value of the correct code and faulty code, respectively. Let us assume that the faulty version of *harmean*, like many numerical programs, is considered to fail if the output error exceeds a predefined threshold ϵ, that is, if

$$outerr = |r_{correct} - r_{faulty}| > \varepsilon$$

Statements	Correct Code	Faulty Code
s_1	`double harmean(a ,b)`	`double harmean(a,b,c)`
s_2	`k=a+b;`	`k=a+b;`
s_3	`x=a*b;`	`x=a*b+c;  //bug`
s_4	`if (x==0) {`	`if (x==0) {`
s_5	`return null;`	`return null;`
s_6	`}`	`}`
s_7	`else {`	`else {`
s_8	`r=2*x/k +k;`	`r=2*x/k +k;`
s_9	`return (r);`	`return (r);`
	`}`	`}`

Figure 1. Motivating Example

Our goal is to automatically localize faults in numerical expressions, given variable-value profiles and expected outputs. In the faulty code of Figure 1, there are three numerical expressions, in statements s_2, s_3 and s_8 , which define variables k, x, and r. A simple approach to detecting which expressions contain numerical faults is to obtain a "suspiciousness" score for each numerical expression by calculating a measure of the statistical association between the values of the

DOI: 10.1002/stvr

expression and the output error. This naïve approach has two potential problems, however. First, erroneous values due to a fault in one expression may propagate to other, correct expressions, causing them to receive high suspiciousness scores. For example, in Figure 1 the fault in statement s_3 causes variable x to have erroneous values that propagate to statement s_8 and cause variable r to have erroneous values. Consequently, the output error of *harmean* is associated with the values computed by both s_3 and s_8, even though s_8 is correct. This problem, which is an instance of *confounding bias* (or *confounding*), is due to the fact that the computation of x at s_3 is a *common cause* of program failure and of the computation of r at s_8. A naïve measure of the association of r with the output error, which does not account for confounding, will reflect both the true causal effect of r on failure and the biasing effect of the confounder x on r and $outerr$.

The second problem with the naïve approach is that common association measures like Pearson's correlation coefficient [14] are often inadequate to measure the causal effect of a numerical expression on a program's output error, even without confounding. For the example in Figure 1, $outerr = |r_{correct} - r_{faulty}| = |2c/(a+b)|$. If $a = 1$ and $b = 0$, then $outerr = 2|c|$ and $x = c$. By definition, the correlation between x and $outerr$ is:

$$corr(x, outerr) = \frac{cov(x, outerr)}{\sigma_x \sigma_{outerr}} = \frac{cov(c, 2|c|)}{\sigma_x \sigma_{outerr}}$$

where $cov(x, outerr)$ is the covariance between x and $outerr$ and where σ_x and σ_{outerr} are the standard deviations of x and $outerr$, respectively. Using basic covariance identities, we find that $cov(c, 2|c|) = 2cov(c, |c|) = 2E[(c - \mu_c)(|c| - \mu_{|c|})]$, where μ_c and $\mu_{|c|}$ are the means of c and $|c|$ respectively. If the distribution of variable c is symmetric about 0, $cov(c, |c|)$ is equal to 0, and so is $corr(x, outerr)$. Therefore, $corr(x, outerr)$ can be 0, even though the statement defining x contains a fault.

NUMFL is intended to address the aforementioned problems and to provide less-biased estimates of the *failure-causing effect* of a numerical expression. We now map the causal variables defined in the Introduction to the variables in our example:

1. Treatment variable T_e is the variable defined in a numerical expression e. In Figure 1, variables k, x and r are the treatment variables associated with statements s_2, s_3 and s_8, respectively.
2. Outcome variable Y is the absolute difference between the output of the faulty program and the expected (correct) output. For the example, Y is $outerr$.
3. X represents the confounding variables, which are each causes of both treatment T_e and outcome Y. For example, in the numerical expression $r = 2 * x/k$, the variables x and k are confounders because they each influence the values of both the treatment $T_e = r$ and the outcome $Y = outerr$.

Figure 2 shows the causal relationships between treatment, outcome, and confounding variables.

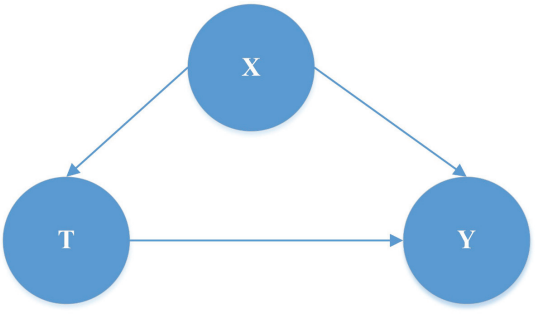

Figure 2. Causal diagram of treatment, outcome, and confounder.

DOI: 10.1002/stvr

3. BACKGROUND

3.1. Causal Inference Methodology

Causal inference methodology [11] has emerged in recent decades from research in a number of applied fields, to provide a basis for making valid inferences, from experimental or observational data, about the causal effects of specified treatments or exposures upon outcomes of interest. Data about study variables is augmented with background knowledge about the causal relationships among variables, which is represented as *causal DAG*: a directed acyclic graph, in which nodes correspond to variables and in which there is an edge $A \rightarrow B$ if A is an actual or assumed cause of B. Causal inference theory establishes graph-theoretic conditions, such as Pearl's Backdoor Criterion [11], under which a given causal effect can be estimated statistically without confounding or other forms of bias such as selection bias and measurement bias.

3.2. Ordinary Propensity Scores

Consider a treatment variable T with two possible values 1 and 0. (These values may indicate "treated" and "untreated", respectively, or they may indicate alternative treatments.) One of the reasons that ideal randomized experiments, when they are feasible, are often considered the "gold standard" for the design of causal inference studies [15] is that randomization makes it likely that the joint distribution of the covariates of the units with $T = 1$ is similar to the covariate distribution of the units with $T = 0$. Such similarity, which is called *covariate balance* [16], implies that the two groups of units are comparable with respect to the values of confounding covariates, so that any difference in average outcomes of the groups is likely to be due to the treatment variable and not to confounding. In randomized experiments, covariate balance is a consequence of the fact that the randomized treatment variable is *independent* of the covariates.

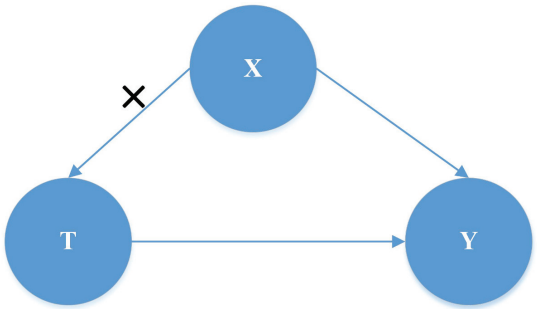

Figure 3. Matching units based on their propensity scores breaks the link between cofounding variables and treatment.

An important approach to achieving covariate balance in observational studies is the use of (ordinary) *propensity scores* [16]. For a binary treatment variable T and a vector of covariates $\boldsymbol{X}$, the propensity score $Pr(T = 1|\boldsymbol{X} = \boldsymbol{x})$ is the conditional probability of treatment $T = 1$ given that the observed value of $\boldsymbol{X}$ is $\boldsymbol{x}$. It has been shown that the propensity score is a *balancing score* in the sense that comparison groups with similar propensity scores tend to have similar covariate distributions [16]. True propensity scores are generally unknown, but they can be estimated from data about study units' actual treatment and covariate values.

Suppose that for each unit i examined in an observational study, we have recorded the values of a binary treatment variable $T_i \in \{0, 1\}$ and a vector of covariates $\boldsymbol{X}_i$. We can use a parametric model $\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)$ to estimate the propensity score for unit i. One choice is the logistic model:

$$\pi_{\boldsymbol{\beta}}(X_i) = Pr(T_i = 1|\boldsymbol{X}_i) = \frac{\exp(\boldsymbol{X}_i{}'\boldsymbol{\beta})}{1 + \exp(\boldsymbol{X}_i{}'\boldsymbol{\beta})}$$

Prepared using stvrauth.cls
DOI: 10.1002/stvr

We can estimate the parameter $\boldsymbol{\beta}$ using maximum likelihood estimation [46], which involves maximizing the log-likelihood function:

$$\hat{\boldsymbol{\beta}}_{MLE} = \arg\ \max \sum_{i=1}^{N} T_i \log\{\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)\} + (1 - T_i) \log\{1 - \pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)\} \quad (1)$$

With the fitted model, we can estimate the ordinary propensity score $\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)$ for each observed unit.

After the propensity score model is estimated, covariate balance may be achieved by matching units based on their estimated propensity scores [16]. For example, each unit with $T = 1$ can be matched to a unit with $T = 0$ that has a similar estimated propensity score, if such a unit exists. As illustrated in Figure 3, matching units based on their propensity scores breaks the link between the confounding variables and the treatment variable.

Ordinary propensity scores are not applicable to continuous treatment variables, however. To address confounding of SFL scores for numerical expressions, we consider two variants of ordinary propensity scores that can accommodate continuous (and integer) treatment variables. The first variant is the Generalized Propensity Score (GPS) proposed by Imai and Van Dyk [12]. The second variant is the Covariate Balancing Propensity Score (CBPS) proposed by Imai and Ratkovic [41]. Since CBPS is quite new, it has not been established whether it is superior to GPS. Hence, we shall present and evaluate two versions of NUMFL, denoted NUMFL-GPS and NUMFL-CBPS, which are based on GPS and CBPS, respectively. First, though, we briefly describe GPS and CBPS in the next two subsections.

3.3. Generalized Propensity Score (GPS)

The *Generalized Propensity Score* proposed by Imai and van Dyk [12] is intended for use with any kind of treatment variable, including continuous ones. Let $p_{\boldsymbol{\psi}}(T|\boldsymbol{X})$ be a model, with parameters $\boldsymbol{\psi}$, of the *conditional probability density function* (p.d.f.) of the treatment variable T given the covariate vector $\boldsymbol{X}$. The GPS for a specific unit with treatment level $T = t$ and covariate values $\boldsymbol{X} = \boldsymbol{x}$ is the density value $r = p_{\boldsymbol{\psi}}(T = t|\boldsymbol{X} = \boldsymbol{x})$. Imai and van Dyk [12] require that the model $p_{\boldsymbol{\psi}}(T|\boldsymbol{X})$ has a *unique parameter* $\boldsymbol{\theta} = \boldsymbol{\theta}_{\boldsymbol{\psi}}(X)$ such that $p_{\boldsymbol{\psi}}(T|\boldsymbol{X})$ *depends on* $\boldsymbol{X}$ *only through* $\boldsymbol{\theta}$. For example, if the treatment is represented by a *linear Gaussian model* $T|\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{X}'\boldsymbol{\beta}, \sigma^2)$ with mean $\boldsymbol{X}'\boldsymbol{\beta}$ (the scalar product $\boldsymbol{X}$-transpose times $\boldsymbol{\beta}$) and variance σ^2, so that $\boldsymbol{\psi} = (\boldsymbol{\beta}, \sigma^2)$, then $\boldsymbol{\theta} = \boldsymbol{X}'\boldsymbol{\beta}$ (here $\boldsymbol{\theta}$ is scalar). The model parameters $\boldsymbol{\psi}$, r, $\boldsymbol{\theta}$, $\boldsymbol{\beta}$, etc. are estimated from data; the corresponding estimates are denoted with by $\hat{\boldsymbol{\psi}}$, $\hat{r}$, $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\beta}}$, etc. In Imai and van Dyk's approach, it is the values of the parameter estimate $\hat{\boldsymbol{\theta}}_{\hat{\boldsymbol{\psi}}}(\boldsymbol{X})$ that are used to control confounding, rather than the values of $p_{\hat{\boldsymbol{\psi}}}(T|\boldsymbol{X})$ themselves.

Table I lists 10 test cases of faulty code in the motivating example. The three inputs a, b and c are generated randomly in the range $[-5, 5]$. Table I also shows the values of intermediate variables values of k and x which are defined in expressions s_2 and s_3 respectively. The variable r in expression s_8 is the faulty code's output and the variable r_c represents the output of the correct code in motivating example with the same inputs. Outcome variable Y is equal to $|r - r_c|$, so all the values of Y are positive in the table. All the values in the table are rounding to three decimals. We use expression s_8 as an example to demonstrate generalized propensity score is estimated. For expression s_8, the treatment variable T_e is variable r and the confounding variables $\boldsymbol{X}$ are covariates x amd k. We first fit a linear regression model $T_e = \beta_0 + \beta_1 x + \beta_2 k$ with r as response, x and k as predictors. Given the data in Table I, the fitted regression model is $T_e = 0.614x + 1.212k - 1.462$. Then we estimate GPS $\hat{\boldsymbol{\theta}}_{\hat{\boldsymbol{\psi}}}(x, k)$ by inputting observed value of x and k into the fitted regression model. The estimated GPS for expression s_8 is denoted by$\hat{\theta}_{s_8}$and is shown in the last column of the table.

The confounding variables' effect on treatment variable can be controlled if conditioned on GPS. This can be proved with the data in TableI. For the confounding variable x's effect on treatment variable r, we fit two regression models: the first model uses r as respond and x as predictor; the second model uses r as respond, but uses both x and GPS $\hat{\theta}_{s_8}$ as predictors. Given the data in TableI, the first model is fitted as $r = 0.487x - 1.729$ and the second model is fitted as

DOI: 10.1002/stvr

$r = -2.271E^{-16}x + 1.0\hat{\theta}_{s_8} - 1.461$. In the first model, the coefficient of x means the variable r will increase 0.487 if x increased 1 unit. In the second model, the coefficient of x is close to 0, while the coefficient of propensity score $\hat{\theta}_{s_8}$ is 1. This means the variable value of r is mostly depend on the value of $\hat{\theta}_{s_8}$. Similarly, we can have another two fitted model for the other confounding variable k: $r = 0.999k - 1.415$ and $r = -2.755E^{-16}k + 1.0\hat{\theta}_{s_8}$-1.461. The coefficient of k is also reduced to 0 in the second model. Thus, given GPS, the effect of confounding variables on treatment variable is significantly reduced, which can also be illustrated by Figure 3

Table I. VARIABLE VALUES AND GPS OF THE MOTIVATING EXAMPLE

Test #	1	2	3	4	5	6	7	8	9	10
a	-1.697	4.088	0.242	-0.269	-3.495	-2.081	-2.063	-2.896	3.839	-3.412
b	-1.430	0.660	4.843	-1.803	-3.352	1.282	-1.520	4.568	1.796	0.325
c	-3.990	3.778	-2.242	-0.347	-1.678	-0.358	3.766	-4.690	-2.535	-1.645
k	-3.127	4.748	5.084	-2.072	-6.847	-0.799	-3.583	1.672	5.636	-3.088
x	-1.564	6.478	-1.071	0.138	10.036	-3.025	6.902	-17.919	4.362	-2.752
r	-2.127	7.477	4.663	-2.205	-9.778	6.777	-7.436	-19.768	7.184	-1.305
r_c	-4.679	5.885	5.545	-2.540	-10.268	5.881	-5.334	-14.157	8.083	-2.371
Y	2.552	1.592	0.882	0.335	0.490	0.896	2.102	5.611	0.900	1.065
$\hat{\theta}_{s_8}$	-4.749	9.731	5.502	-2.426	-2.132	-2.825	-0.102	-8.978	9.506	-5.431

3.4. Covariate Balancing Propensity Score (CBPS)

Section 3.2 explained that the ordinary propensity score is a balancing score and that it can be estimated from data using a parametric statistical model. However, if the propensity score model $\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)$ is *misspecified*, matching on the estimated propensity scores (and related techniques) may fail to balance the covariates, yielding biased estimates of causal effects. To address this problem, Imai and Ratkovic proposed the *Covariate Balancing Propensity Score* (CBPS), which is robust to mild misspecification of the propensity score model [41]. The CBPS optimizes covariate balance by modifying the estimation procedure for parameter $\boldsymbol{\beta}$. Differentiating equation (1), equating the result to zero, and taking expectations, we have [41]

$$E\left\{\frac{T_i\pi'_{\boldsymbol{\beta}}(\boldsymbol{X}_i)}{\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)} - \frac{(1-T_i)\pi'_{\boldsymbol{\beta}}(\boldsymbol{X}_i)}{1-\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)}\right\} = 0, \tag{2}$$

where $\pi'_{\boldsymbol{\beta}}(\boldsymbol{X}_i) = \partial\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)/\partial\boldsymbol{\beta}$. Equation (2) is called a *balancing condition* or a *moment condition* [41]. (The mean of a random variable is its "first moment" [43].) In CBPS, the balancing condition is generalized as

$$E\left\{\frac{T_i\tilde{\boldsymbol{X}}_i}{\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)} - \frac{(1-T_i)\tilde{\boldsymbol{X}}_i}{1-\pi_{\boldsymbol{\beta}}(\boldsymbol{X}_i)}\right\} = 0, \tag{3}$$

where $\tilde{\boldsymbol{X}}_i = f(\boldsymbol{X}_i)$ represents a vector-valued function of the covariates $\boldsymbol{X}_i$. To ensure that the first moment of each covariate is balanced, even when the model is misspecified, we can set $\tilde{\boldsymbol{X}}_i = \boldsymbol{X}_i$. CBPS employs a parameter estimation framework called the *generalized method of moments* (GMM) [42], which is based on solving moment-condition equations for the parameters of interest and substituting sample moments for unknown population moments. Imai *et al*. extended CBPS to treatments with $K > 2$ integer values, using a *multinomial model* for the conditional probability of treatment given the covariates. We adapted this approach slightly to make it applicable to value-based fault localization (see Section 4.2), and we evaluate CBPS in NUMFL as an alternative to GPS .

DOI: 10.1002/stvr

4. TWO VERSIONS OF NUMFL

In this section, we define two versions of NUMFL, NUMFL-GPS and NUMFL-CBPS, and explain how they are used to reduce confounding bias during estimation of a numerical expression's *average failure-causing effect* (AFCE). Intuitively, an aggregate measure of the failure-causing effect of a numerical expression e should summarize the effects on output errors of evaluating e over a sample of program runs that each cover or reach e. Accordingly, in causal inference the designated "treatment" variable T_e should reflect the values produced by individual evaluations of e. (Of course e may be evaluated repeatedly in a single program run, due to iteration.) Runs that do not reach e are *not relevant* to such a measure. This is unlike measures of the failure-causing effect of covering a statement s at least once during a run (e.g., [3]), which contrast runs that cover s with runs that don't cover s. With such measures, the treatment variable indicates whether or not s was covered during a given run.

Recall that confounding of the causal effect of a treatment variable T upon an outcome variable Y is bias that is due to the presence of one or more common causes of T and Y. The influence of such a common cause X can make the effect of T on Y, as reflected by a naïve measure of statistical association, appear to be stronger or weaker than it really is. As mentioned in Section 3.2, ordinary propensity scores cannot be used in causal inference to control confounding when the treatment variable is continuous. Since the "treatments" that NUMFL deals with are the values of numeric program variables, we instead define and evaluate two versions of NUMFL, namely NUMFL-GPS and NUMFL-CBPS, that are based respectively on GPS and CBPS, the two generalizations of ordinary propensity scores described in Sections 3.3 and 3.4.

4.1. NUMFL-GPS

As with ordinary propensity scores, GPS is used to control for statistical associations between the treatment variable and confounding covariates, which in the case of NUMFL are variables used (read) in a numeric expression e whose AFCE we wish to estimate. In general, these program variables are potential confounders because they may contain erroneous values just before expression e is evaluated, due to faults in previously evaluated expressions. If these erroneous values cause program failures (that is, "coincidental correctness" does not occur) the result of evaluating e may be strongly associated with failures even if e is correct, unless steps are taken to control confounding. In NUMFL, therefore, when estimating the AFCE of T_e on the output error Y, we control for the values of the variables used in e. (Those familiar with Pearl's Backdoor Adjustment Criterion [11] should note that this breaks any backdoor paths between T_e and Y in the acyclic data flow graph induced by a program execution.)

4.1.1. Control of Confounding. For a given numerical expression e in a program, NUMFL-GPS currently fits a linear Gaussian GPS model $\boldsymbol{T}_e|\boldsymbol{X}_e \sim N(\boldsymbol{X}'_e\boldsymbol{\beta}_e, \sigma_e^2)$, where the treatment variable T_e represents the result of evaluating e (once) and where the possibly confounding covariates in the vector $\boldsymbol{X}_e$ represent the corresponding values of the program variables used in evaluating e. Observe that one program run may generate multiple values of T_e and $\boldsymbol{X}_e$, due to iteration. To reduce profiling overhead, these may be sampled rather than recorded exhaustively. We currently sample only the values from the *last iteration* of a loop.

Applying GPS in NUMFL entails the following three steps:

1. Given a sample of observed values for the treatment variable T_e and for the covariates $\boldsymbol{X}_e$, fit a linear regression model $T_e = \boldsymbol{X}'_e\boldsymbol{\beta}_e$ to obtain the estimated parameter vector $\hat{\boldsymbol{\beta}}_e$.
2. For each observed value $\boldsymbol{x}_{e,i}$ of $\boldsymbol{X}_e$, compute the estimate $\hat{\theta}_{e,i} = \boldsymbol{x}_{e,i}{}'\hat{\boldsymbol{\beta}}_e$.
3. Group observations with the same or similar values of $\hat{\theta}_{e,i}$ into m subclasses of roughly equal size.

The regression model fitted in the first step characterizes how the values of the covariates in $\boldsymbol{X}_e$ influence the value of the treatment variable T_e. In the second step, $\hat{\theta}_{e,i}$ estimates this influence for a particular value $\boldsymbol{x}_{e,i}$ of $\boldsymbol{X}_e$. The scalar $\hat{\theta}_{e,i}$ summarizes the influence of the vector value $\boldsymbol{x}_{e,i}$ on the

DOI: 10.1002/stvr

treatment, and hence $\hat{\theta}_{e,i}$ may be used in place of $\boldsymbol{x}_{e,i}$ for confounding control. Accordingly, in the third step, each subclass of observations with similar $\hat{\theta}_{e,i}$ values corresponds to a set of $\boldsymbol{x}_{e,i}$ values that influence the treatment similarly. Thus, within each subclass, there is little or no association between confounders and the treatment variable T_e.

To illustrate this procedure, we refer again to the faulty version of the function *harmean* in Figure 1. Statement s_8 is $r = 2 * x/k + k$. To control possible confounding of the average failure-causing effect of r by the variables x and k, we first fit a linear model $r = \beta_1 z + \beta_2 k$, where $z = x/k$, using a sample of corresponding observed values (r_i, x_i, k_i). Second, we apply the fitted model to each pair (x_i, k_i) to obtain a predicted value $\hat{\theta}_i$. Third, we group the observations with similar $\hat{\theta}_i$ values into subclasses. This can be done by sorting the $\hat{\theta}_i$ into descending order and partitioning the sorted values into m roughly equal-size bins.

Although GPS can control confounding bias caused by multiple confounding covariates, there is one challenge to applying it in NUMFL-GPS. If the dimension of $\boldsymbol{X}_e$ is high, the first step of GPS requires a large number of observations to fit a suitable regression model. In practice, a numerical expression defined by a statement could contain more than 10 confounders, but the number of tests may not be large enough to fit the regression parameters with adequate precision. Thus, to apply GPS, we need to control the dimension of confounding variables.

To address this problem, we *decompose* a complex numerical expression into several subexpressions, each involving operators with the same precedence level. For example, a numerical statement $a = (b + c + d) * e$ will be decomposed into two subexpressions: (1) $temp = b + c + d$ and (2) $a = temp * e$. Here, $temp$ is a temporary variable that plays the role of the treatment variable for subexpression (1) and the role of a possible confounder for subexpression (2). Thus, each subexpression has fewer confounding variables and a relatively simple structure. For each subexpression, we use GPS to control confounding bias and then estimate the subexpression's AFCE.

4.1.2. Failure-causing Effect Estimation. Within each of the subclasses created by grouping observations with similar $\hat{\theta}_i$ values, the treatment values $T_{e,i}$ should be largely independent of the covariate values $\boldsymbol{X}_{e,i}$, provided that the GPS model is adequate. Hence, the relationship between the treatment variable T_e and the outcome variable Y should be nearly unconfounded in each subclass. If this relationship is roughly linear in each subclass, we can estimate the expected dose-response function $E[Y|T_e]$ *within a particular subclass* using a simple linear regression model [17]:

$$Y = \gamma_e T_e + b \tag{4}$$

Recall that Y is the absolute difference between the output of the faulty program and the expected (correct) output. In equation (4), b is a constant intercept. The coefficient γ_e is the slope of $E[Y|T_e]$. It indicates how much the expected value of the outcome variable changes when the treatment value increases by one unit. The slope coefficient γ_e can be estimated by the least-squares estimator $\hat{\gamma}_e$:

$$\hat{\gamma}_e = (\boldsymbol{t}_e' \boldsymbol{t}_e)^{-1} \boldsymbol{t}_e' \boldsymbol{y}, \tag{5}$$

where $\boldsymbol{t}_e'$ is the transpose of the vector of treatment values and where $\boldsymbol{y}$ is the vector of outcome values (output errors).

Intuitively, an association measure used as an SFL suspiciousness metric for numerical programs should, when applied to a numerical expression e, reflect the likelihood that e is faulty. Although the coefficient estimate $\hat{\gamma}_e$ summarizes the dose-response function, it is not an adequate SFL metric, because of the *symmetry* of many numerical errors. For example, suppose that $Y = |T_e|$. Since the output error Y is completely determined by T_e, expression e should receive a high suspiciousness score. However, if the value of T_e is distributed symmetrically around zero, then $\hat{\gamma}_e$ will be close to zero, which suggests misleadingly that e has no causal effect on program failures.

To derive a better suspiciousness metric, we first analyze the relationship between a numeric fault and the DRF slope estimator $\hat{\gamma}_e$. Assume that the numeric expression e has a fault, which results in erroneous treatment values represented by the treatment variable $T_e^f = T_e$. Let the treatment

DOI: 10.1002/stvr

variable T_e^c represent the corresponding treatment values in the correct program. Then we have $T_e^f = T_e^c + \varepsilon_e$, where ε_e is a treatment error term. Then $\hat{\gamma}_e$ may be decomposed as follows:

$$\begin{aligned} \hat{\gamma}_e &= (\boldsymbol{t}_e^{f\prime}\boldsymbol{t}_e^f)^{-1}(\boldsymbol{t}_e^c + \boldsymbol{\varepsilon}_e)'\boldsymbol{y} \\ &= (\boldsymbol{t}_e^{f\prime}\boldsymbol{t}_e^f)^{-1}\boldsymbol{t}_e^{c\prime}\boldsymbol{y} + (\boldsymbol{t}_e^{f\prime}\boldsymbol{t}_e^f)^{-1}\boldsymbol{\varepsilon}_e{}'\boldsymbol{y} \\ &= \hat{\gamma}_e^c + \hat{\gamma}_e^f, \end{aligned} \tag{6}$$

where $\boldsymbol{t}_e^f$, $\boldsymbol{t}_e^c$, and $\boldsymbol{\varepsilon}_e$ are the vectors of sample values for T_e^f, T_e^c, and ε_e, respectively, and where $\hat{\gamma}_e^c = (\boldsymbol{t}_e^{f\prime}\boldsymbol{t}_e^f)^{-1}\boldsymbol{t}_e^{c\prime}\boldsymbol{y}$ and $\hat{\gamma}_e^f = (\boldsymbol{t}_e^{f\prime}\boldsymbol{t}_e^f)^{-1}\boldsymbol{\varepsilon}_e{}'\boldsymbol{y}$. The component $\hat{\gamma}_e^f$ of $\hat{\gamma}_e$ characterizes the causal relationship between the treatment error ε_e and the output error Y. If ε_e is symmetrically distributed then the contributions to $\hat{\gamma}_e^f$ of the positive and negative values of ε_e will tend to cancel each other, since Y is an absolute value, leaving $\hat{\gamma}_e^f$ close to zero.

We have considered two possible approaches to this problem. We call the first of these a *dual linear regression model* (DLRM). The basic idea of DLRM is simple: compute $|\hat{\gamma}_e^f|$ separately for positive ε_e and for negative ε_e and then add the two resulting values. (Using the absolute values of the two estimates ensures that they do not cancel each other.) However, although the values of T_e are known, the values of T_e^c and ε_e are unknown, so $|\hat{\gamma}_e^c|$ and $|\hat{\gamma}_e^f|$ cannot be estimated individually. This issue can be addressed given the following assumptions:

Assumption 1*: If expression e is faulty then within each GPS subclass the variance of T_e^c is much smaller than the variance of ε_e.*
Assumption 2*: If expression e is faulty then within each GPS subclass, ε_e is symmetrically distributed about zero.*

Assumption 1 implies that the error ε_e is responsible for most of the variance of treatment variable T_e within a subclass. We have observed that this is often the case. Assumption 2 asserts that the previously mentioned issue with symmetrically distributed treatment errors pertains generally.

Given these two assumptions, using DLRM involves the following steps, which are applied separately to each GPS subclass:

1. Sort the values of the treatment variable T_e into descending order. Then split the data into two subsets which contain the values that are larger and smaller than the median value, respectively.
2. Estimate the slope of $E[Y|T_e]$ in each subset separately using linear regression model (1).
3. Summarize the failure-causing effect of T_e by adding the two estimated slopes absolute values.

Under Assumption 1, if expression e is faulty then T_e^c can be viewed as a constant relative to ε_e within a subclass, so that a large value of T_e corresponds to a large ε_e. This implies that T_e and ε_e have the same sorted order. Assumption 2 implies that splitting the treatment data at its median value separates the treatments with positive ε_e values from the treatments with negative ε_e values. Step 3 prevents the two causal effect estimates from cancelling each other out. Figure 4 (a) shows the DRF curve of the DLRM.

In practice, the distribution of ε_e is usually unknown, so it is quite uncertain whether Assumption 2 holds. To address this issue, we have used an alternative to DLRM based on a *quadratic regression model* [18], which we shall refer to as QRM. QRM is more flexible than DLRM, because it does not require sorting and splitting the data within subclasses. It is based on the following alternative to Assumption 2:

Assumption 3*: Executions with large absolute treatment errors $|\varepsilon_e|$ have larger output errors Y than executions with small values of $|\varepsilon_e|$.*

We have observed that this assumption often holds. If both Assumption 1 and Assumption 3 hold, the failure-causing effect of T_e on Y can be estimated by fitting a quadratic regression model $Y = \varsigma T_e^2 + \eta T_e + c$. Figure 4 (b) shows the dose-response curve of QRM, which clearly reflects

DOI: 10.1002/stvr

Assumption 3. We use the fitted value $\hat{\zeta}$ of the coefficient ζ as the AFCE estimate within each subclass.

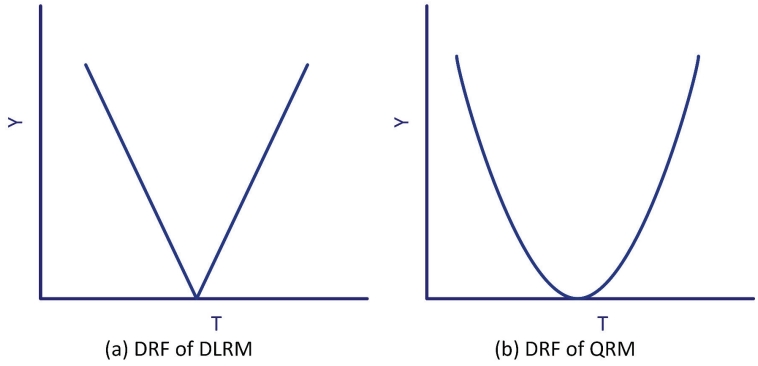

Figure 4. Matching units based on their propensity scores breaks the link between cofounding variables and treatment.

Finally, we chose to use the absolute value of fitted a regression coefficient's t-statistic [19] as a suspiciousness score, rather than the coefficient itself, because we wish to "penalize" coefficients with large standard errors. We use $|t|$ to denote the absolute value of the t-statistic for a fitted coefficient $\hat{\beta}$: $|t| = |\hat{\beta}/std(\hat{\beta})|$, where $std(\hat{\beta})$ is the standard error of $\hat{\beta}$. Thus, for DLRM the failure-causing effect of treatment T_e is estimated by the summation of two linear regression models'$|t|$ values for the fitted slope coefficient $\hat{\gamma}_e$. For QRM, this effect is estimated by the $|t|$ value of the estimated coefficient $\hat{\zeta}$ of the quadratic term of the regression model.

Figure 5 shows the NUMFL-GPS algorithm. Note that all input values are standardized ($x \rightarrow \frac{x-\mu}{\sigma}$) to adjust for differences in scale between variables. NUMFL-GPS processes each numeric expression or subexpression e in three stages. In the first stage (lines 2-5 and 10-14), it fits a linear propensity score model regressing the treatment variable T_e on the confounding variables X_e, then uses the fitted model to calculate the generalized propensity score for each observation. The scores are grouped into m roughly equal-sized subclasses. In the second stage (lines 15-28), the algorithm estimates the "penalized" AFCE of T_e on Y separately in each subclass. If DLRM is employed, the data within a subclass is split into two equal-size groups. The algorithm fits a linear model for each group, and it records the t-statistics τ_1 and τ_2 for the coefficients of T_e in the two models. The penalized AFCE for subclass k is given by the sum $\tau^k = |\tau_1| + |\tau_2|$. If QRM is employed, the algorithm fits a quadratic model and records the coefficient $\hat{\zeta}$ of the quadratic term. The penalized AFCE for subclass k is estimated by the absolute value of the t-statistic for $\hat{\zeta}$. In the third stage, the overall AFCE or suspiciousness score $\tau(e)$ for subexpression e is computed as a weighted average of τ^k over all subclasses k (line 29). Finally, the suspiciousness scores for all subexpressions are sorted in descending order (line 31). The developer uses the sorted list to guide fault localization.

4.2. NUMFL-CBPS

NUMFL-CPBS differs from NUMFL-GPS in that the former employs the Covariate Balancing Propensity Score instead of GPS for confounding control. Recall from Section 3.4 that the CBPS seeks to optimize covariate balance by employing the Generalized Method of Moments (GMM), which solves balance equations for the parameters of interest.

4.2.1. Control of Confounding. In Imai and Ratkovic's extension of CBPS to treatment variables with $K > 2$ discrete values [42], a multinominal logistic regression model is used to estimate the conditional probability $\pi^k_{\boldsymbol{\beta}}(\boldsymbol{x}) = Pr(T = k|\boldsymbol{X})$ of treatment k. To handle a continuous treatment variable T_e associated with a numerical expression e, we discretize the values of T_e by partitioning them into ordered bins. For each k, $1 \leq k < K$, the probability that the original value of T_e belongs

DOI: 10.1002/stvr

Figure 5. *NUMFL* Algorithm

Input: Subexpression value tables $E = [T_e, \boldsymbol{X}_e]$, with all values standardized, and type of model M. The number of clusters m. The number of subexpressions n.
Output: Suspiciousness scores $\tau\,(e)$ of subexpressions, sorted in descending order

for $i = 1 \ldots n$ **do**
  $T_e \leftarrow$ treatment variable of ith subexpression
  $\boldsymbol{X}_e \leftarrow$ confounding variables of ith subexpression
  if using GPS **then**
    Fit linear regression model $T_e = \boldsymbol{X}_e{}'\boldsymbol{\beta}_e$
  else if using CBPS **then**
    Fit models for $Pr(T_e = k|\boldsymbol{X}_e)$,
    $k = 1 \ldots K$, using multinomial logistic regression
  end if
  for each $\boldsymbol{x}_{e,j}$ **do**
    Estimate the propensity score for $\boldsymbol{x}_{e,j}$ using fitted GPS or
    CBPS model(s).
  end for
  Group the observations into m roughly equal-size subclasses.
  for each subclass k **do**
    if M is DLRM **then**
      Fit two linear regression models:
      $Y = \gamma_{e,1}T_e + b$ for observations $T_e > median(T_e)$
      $Y = \gamma_{e,2}T_e + b$ for observations $T_e \leq median(T_e)$
      $\tau_1 = t$-value of $\gamma_{e,1}$
      $\tau_2 = t$-value of $\gamma_{e,2}$
      $\tau^k = |\tau_1| + |\tau_2|$
    end if
    if M is QRM **then**
      Fit quadratic regression model $Y = \zeta {T_e}^2 + \eta T_e + c$
      $\tau^k = |t$-value of $\hat{\zeta}|$
    end if
  end for
  $\tau(e)$ = weighted average of τ^k across m subclasses
end for
Sort $\tau(e)$ values in descending order.

in bin k is modeled by

$$\Pr(T_e = k|\boldsymbol{X}_e) = \frac{e^{\boldsymbol{\beta}_k \boldsymbol{x}_e}}{1 + \sum_{i=1}^{K-1} e^{\boldsymbol{\beta}_i \boldsymbol{x}_e}}$$

and

$$\Pr(T_e = K|\boldsymbol{X}_e) = \frac{1}{1 + \sum_{i=1}^{K-1} e^{\boldsymbol{\beta}_i \boldsymbol{x}_e}}$$

The CBPS can be estimated using the open source R package "CBPS" created by Fong, *et al* [44].

The whole process of applying CBPS in NUMFL involves the following three steps:

1. Given a sample of observed values for the discretized treatment variable T_e and for the covariates $\boldsymbol{X}_e$, use the R package "CBPS" to estimate the coefficients of a multinominal logistic propensity score model $\pi^k_{\boldsymbol{\beta}}(\boldsymbol{x}_{e,i}) = P(T_{e,i} = k|\boldsymbol{x}_{e,i})$ for each k.
2. For each observed value $\boldsymbol{x}_{e,i}$, estimate propensity score $\hat{\pi}_{e,i}$ with the fitted propensity score model.
3. Group observations with the same or similar values of $\hat{\pi}_{e,i}$ into m subclasses of roughly equal size.

DOI: 10.1002/stvr

Here $\hat{\pi}_{e,i}$ represents the calculated propensity score for each observed value. Similar $\hat{\pi}_{e,i}$ values correspond to a set of $\boldsymbol{x}_{e,i}$ values that influence the treatment similarly, so we group the observations into subclasses as we did in NUMFL-GPS. Propensity score estimation and grouping in NUMFL-CBPS corresponds to lines 6-14 in the algorithm of Figure 5.

4.2.2. Failure-causing Effect Estimation. Failure-causing-effect estimation in NUMFL-CBPS follows the same process followed in NUMFL-GPS, which is shown in lines 15-29 of Figure 5. For a given subexpression e, QRM or DLRM is used to estimate the "penalized" AFCE of T_e on Y within each subclass individually. Then the failure-causing effect is calculated as a weighted average the penalized AFCEs over all subclasses. Finally, the results for all subexpressions are sorted.

Table II. STANDARDIZED VARIABLE VALUES OF THE MOTIVATING EXAMPLE

Test #	1	2	3	4	5	6	7	8	9	10
a	-0.166	0.876	0.183	0.091	-0.490	-0.235	-0.232	-0.382	0.831	-0.475
b	-0.364	0.023	0.797	-0.433	-0.720	0.138	-0.381	0.746	0.233	-0.039
c	-0.523	0.834	-0.218	0.113	-0.119	0.111	0.832	-0.646	-0.269	-0.114
k	-0.336	0.580	0.619	-0.213	-0.769	-0.065	-0.389	0.222	0.683	-0.331
x	-0.110	0.404	-0.079	-0.001	0.631	-0.204	0.431	-1.156	0.269	-0.186
r	-0.027	0.517	0.358	-0.031	-0.460	0.478	-0.328	-1.027	0.501	0.020
Y	2.552	1.592	0.882	0.335	0.490	0.896	2.102	5.611	0.900	1.065

We will use the expression s_3 and s_8 in the faulty code of motivating example shown in Figure1 to demonstrate the NUMFL algorithm in Figure5 step by step. The variable values are come from TableI. In the motivating example, we know that the bug exists in expression s_3, so the estimated suspiciousness score of s_3 should be larger than the estimated suspiciousness score of s_8. In practice, running the NUMFL algorithm requires decomposing complex numerical expression in to several subexpressons , which has relatively simple structure. But the expressions s_3 and s_8 already have simple structure, so we choose to estimate the AFCE of the expressions without decomposing them into subexpressions.

For expression $s3$, the first step is to identify the treatment variable and confounding variables. In the expression $s3$, the treatment variable T_e is variable x and the confounding variables are a, b and c. Then we adjust the differences in scale between variables by standardizing the variables' values. TableII shows all the variables values after standardization. Note that the outcome variable Y is not standardized. Next we estimate propensity scores for each observations. To estimate GPS $\hat{\theta}_{s_3}$, we can fit a linear regression model as described in section3.3. If NUMFL-CBPS is used, we need to fit a multinomial logistic regression model, which is some what complex. Here we use R package "CBPS" to estimate the CBPS which can automatically fit a multinomial regression model with the treatment variables and confounding variables and then estimate the propensity score. For the detail of the package "CBPS", the reader should refer to [44]. TableIII shows expression s_3's estimated GPS $\hat{\theta}_{s_3}$ and CBPS $\hat{\pi}_{s_3}$ for each observation. With the estimated propensity scores, we group observations with same or similar propensity scores into m subclasses. For simplicity, we use $m = 2$ for the motivating example. We rank the tests according to their propensity scores in descending order. The first 5 tests are in subclass 1 and the last 5 tests are in subclass 2. The last two columns of TableIII shows the subclass index of each test for NUMFL-GPS and NUMFL-CBPS. From TableIII, we can see there is some difference between the subclassification results of NUMFL-GPS and NUMFL-CBPS. In NUMFL-GPS, test 4 and test5 are in subclass 2 , while in NUMFL-CBPS, test4 and test5 belong to subclass1. Similarly, we calculate GPS and CBPS for expression s_8, which is summarized in TableIV.

The suspiciousness score of expression is estimated by the average of failure-causing effect in two subclasses. Assume we use QRM to estimate failure-causing effect, a quadratic regression model $Y = \zeta {T_e}^2 + \eta T_e + c$ is fitted in each subclass. The $|t\text{-value}|$ of the coefficient ζ is the estimated failure-causing effect. TableV shows the estimated failure-causing effect in each subclass when NUMFL-GPS is used. In TableV, $|t - value|_1$ denotes the estimated failure-causing effect in

DOI: 10.1002/stvr

subclass1 and $|t-value|_2$ denotes the estimated failure-causing effect in subclass2. The estimated suspiciousness score (AFCE) is equal to $(|t-value|_1 + |t-value|_2)/2$. For comparison, we also fit a quadratic regression model with full sample without subclassification and the estimated failure-causing effect is denoted by $|t-value|_0$. In TableV, $|t-value|_0$ of s_8 is close to $|t-value|_0$ of s_3, even though s_8 is not faulty. This is because the faulty variable x in s_3 is confounder of variable r in s_8 and make s_8 associate to program failure. When NUMFL-GPS is applied, the estimated AFCE of s_8 is significantly smaller than the estimated AFCE of s_3. The estimated suspiciousness scores of NUMFL-CBPS are shown in TableVI. In TableVI, the difference between s_3 and s_8's suspiciousness score is larger than the difference between their $|t-value|_0$s. This means both GPS and CBPS are effective in controlling confounding bias.

Table III. ESTIMATED GPS AND CBPS OF EXPRESSION s_3

Test #	$\hat{\theta}_{s_3}$	$\hat{\pi}_{s_3}$	Subclass index	
			GPS	CBPS
1	0.057	0.847	2	2
2	0.364	0.716	2	2
3	-0.606	5.65E-11	1	1
4	0.310	5.05E-06	2	1
5	0.692	1.60E-09	2	1
6	-0.143	5.872	1	2
7	0.637	2.570	2	2
8	-1.119	5.22E-06	1	1
9	-0.113	0.130	1	1
10	-0.079	5.694	1	2

Table IV. ESTIMATED GPS AND CBPS OF EXPRESSION s_8

Test #	$\hat{\theta}_{s_8}$	$\hat{\pi}_{s_8}$	Subclass index	
			GPS	CBPS
1	-0.258	0.521	1	2
2	0.562	3.156	2	2
3	0.323	3.878	2	2
4	-0.127	3.099	1	2
5	-0.110	0.004	2	1
6	-0.149	1.14E-07	1	1
7	0.005	0.008	2	1
8	-0.498	1.51E-06	1	1
9	0.549	3.079	2	2
10	-0.297	0.069	1	1

Table V. ESTIMATED AFCE WITH NUMFL-GPS

Expression	$\|t-value\|_1$	$\|t-value\|_2$	Suspiciousness score	$\|t-value\|_0$
s_3	13.861	0.228	7.045	2.763
s_8	1.045	0.515	0.78	2.601

5. EMPIRICAL EVALUATION

To evaluate the effectiveness of NUMFL, we conducted an empirical study involving several subject programs, in which the performance of NUMFL was compared with the performance of several

DOI: 10.1002/stvr

Table VI. ESTIMATED AFCE WITH NUMFL-CBPS

Expression	$\|t-value\|_1$	$\|t-value\|_2$	Suspiciousness score	$\|t-value\|_0$
s_3	4.034	2.219	3.127	2.763
s_8	1.898	0.325	1.112	2.601

baselines techniques. In the study, we investigated four main research questions: (1) What is the performance of NUMFL compare to baseline techniques when each subject program version contains only one fault? (2) What is the performance of NUMFL compare to the baselines when each subject program version contains two faults? (3) Given data from only failing executions, is NUMFL effective? (4) Which is most effective, NUMFL-GPS or NUMFL-CBPS?

5.1. Experimental Platform and Data Collection

We developed a data collection and analysis platform for numerical fault localization studies involving Java programs. The platform is based on a modified Java parser [20] and the ASM Java bytecode manipulation framework [21]. We first parse the source code of a subject program. Each complete numerical expression is decomposed into a set of subexpressions as described in Section 4.1. The ASM compiler is used to instrument the bytecode files of the subject program to record variable values used and produced by the evaluation of each subexpression. Information is recorded to permit the recorded values to be mapped back to the corresponding source-code subexpressions. Each instrumented subject program is executed on a test suite and the induced variable values are recorded. For a subexpression within a loop, we currently record the associated values only for the last iteration of the loop, in order to reduce overhead.

The data collection algorithm is implemented in Java and can be downloaded from [22]. The NUMFL algorithm and other baseline techniques were implemented using the R statistical computing environment [23]. In this study, the number of propensity score subclasses used in the NUMFL algorithm was set to 10.

5.2. Subject Programs, Faults, and Tests

Subject Programs. For the empirical study, we selected 16 subject programs from four Java numerical libraries: (1) Apache Common Math (versions 2.1 and 3.1.1), which is a popular Java library for scientific computing [24]; (2) the Oj! Algorithms library Ojalgo (version 33.0), which is an open source Java library for mathematics, linear algebra and optimization [25]; (3) JAMA, which is a basic linear algebra package for Java [26]; and (4) SciMark 2.0, which is a Java benchmark for measuring the performance of numerical codes occurring in scientific applications [27]. We chose subject programs whose purpose we understood (so that we would be able to write tests for them if necessary) and having as many numerical expressions as possible. We chose four large (≥ 1500 SLOC) and eight small subject programs (< 1500 SLOC) from Apache Common Math; for Ojalgo, we wrote a subject program to calculate the Schur decomposition of a matrix that uses two Ojalgo classes (PrimitiveDenseStore and HermitianEVD32); for JAMA, we created one subject program that uses four different matrix decomposition algorithms; we used two of SciMark's five computation kernels as subject programs, FFT and LU factorization. A summary of our subject programs is shown in Table VII.

Faults. To evaluate NUMFL, we required faulty versions of subject programs. After much searching, we were able to find few well documented faults (including fixes) that produced numerical output that was sometimes, but not always, incorrect. Hence, we *injected* three types of simulated numerical faults into the subject programs: (1) adding a small random number to a numerical expression (the number varied between and within runs); (2) multiplying a numerical expression by a random number; (3) randomly changing one of the operators of a numerical expression (once only). Each faulty version of a subject program contained one bug. We generated 12 faulty versions of each subject program whose number of lines of source code was larger than 1500. For each subject program with fewer than 1500 SLOC, we generated two faulty versions.

DOI: 10.1002/stvr

Table VII. SUMMARY OF SUBJECT PROGRAMS

Subject Program	SLOC	# of Sub-expressions	# of Tests	# Faulty versions
Apache_EigenDecompose	1858	611	8000	12
Apache_DScompiler	1774	220	5000	12
Apache_BigMatrix	1580	244	3000	12
Apache_Rotation3D	1704	504	3000	12
Ojaljo_SchurDecompose	2129	649	5000	12
Jama_MatrixDecompose	1952	578	5000	12
SciMark_LU	295	35	3000	2
SciMart_FFT	197	59	3000	2
Apache_SymmLQ	1226	57	5000	2
Apache_SplineInterpolator	130	45	5000	2
Apche_SimpleRegress	869	61	5000	2
Apache_SchurTransformer	458	154	3000	2
Apache_MillerUpdatRegress	1110	152	1000	2
Apache_HarmonicFitter	385	50	3000	2
Apache_FastSine	197	66	5000	2
Apache_FastCosine	188	77	5000	2

Except for SciMark, every subject program came with a unit test suite which provided a predefined error tolerance; we chose a tolerance of 1.0E-5 for SciMark. We ran the faulty versions of the subject programs and the original correct versions with the same inputs. If the output error Y exceeded the tolerance, the execution was considered to be failure. If the subject program's output was a scalar variable, the output error Y was the absolute difference between the correct and faulty programs' outputs. If the subject program's output was an array, we calculated the difference between each element in the faulty program's output and the corresponding element in correct program's output. The output error Y was taken to be the first difference encountered that was larger than the tolerance or zero if no such difference was found.

Tests. Since a number of our subject programs came with very few test cases, it was necessary to create test cases for them. We first analyzed the subject programs and the test cases provided by their developers. Then we randomly generated inputs similar to those of the test cases. Note that for NUMFL, we did not generally use the total number of tests shown in Table VII, though we did so for the baseline techniques. Instead, we first selected all N tests of the smaller of the set of failed tests and the set of passed tests, and we then randomly selected N tests from the larger set. (See Section 5.5 for an exception.)

5.3. Baseline SFL Metrics and Cost Measure

Baselines. We compared NUMFL to four baseline SFL metrics. Two are coverage-based metrics that have performed well in comparisons with other such metrics: Ochiai [28] and DStar (with $star = 2$) [29]. The Ochiai metric estimates the suspiciousness of a statements s with the equation

$$O(s) = f_s / \sqrt{f(f_s + p_s)} \tag{7}$$

where: f_s is the number of failed tests that cover statement s; p_s is number of passed tests that cover s; and f is the total number of failed tests. The Dstar metric (with $star = 2$) is

$$D(s) = {f_s}^2 / (p_s + f - f_s) \tag{8}$$

where $f - f_s$ is the number of failed tests that do not cover s.

The other two baselines are predicate-level SFL metrics, SOBER [5] and the Exploratory Software Predictor (ESP) [30]. SOBER computes the probability $\pi(P)$ that a predicate P evaluated to be true in an execution. If the distribution of $\pi(P)$ in passing runs differs significantly from its

DOI: 10.1002/stvr

distribution in failing runs then P is considered to be related to the fault. In this study, for a treatment variable T_e we use these predicates[31]: $T_e > 0$, $T_e = 0$ and $T_e < 0$.

ESP employs "elastic predicates" to partition the value space for each variable x at each assignment to x. ESP provides two instrumentation schemes: single variable and scalar pairs. The single variable scheme uses execution profiles to compute the mean μ_x and standard deviation σ_x for a variable x. These statistics are then used in nine predicates (e.g., $x < \mu_x + \sigma_x$) that collectively partition the values of x for three standard deviations above and below μ_x. ESP then computes an importance score for each predicate. The suspiciousness of each assignment to x is taken to be the highest importance score among the corresponding elastic predicates. In a similar way, the scalar pair [31] scheme addresses differences in the values of pairs of variables (e.g., $x - y > \mu_{x-y} + \sigma_{x-y}$).

Cost Measure. We measured the costs of applying NUMFL and the baseline metrics by the percentage of *subexpressions* that need to be examined, in decreasing order of suspiciousness scores, to find the fault, assuming the fault is recognized when it is encountered. This contrasts with most SFL techniques, which localize faults at the statement level. There are two reasons that we chose to have NUMFL compute suspiciousness scores for subexpressions rather than for statements. First, the location of a faulty subexpression not only implies the location of the numerical statement that contains the subexpression, but it also indicates which part of the statement is problematic. Second, in this study we collected data only for floating point variables involved in numerical expressions. As a result, the number of subexpressions was usually smaller than the number of lines of code. To ensure a fair comparison between NUMFL and the predicate-level baseline metrics, predicates were associated with subexpressions. In comparing NUMFL to the statement-level, coverage-based baseline metrics, subexpressions belonging to same statement received the *same suspiciousness score*. To compare NUMFL fairly to the latter metrics, if multiple subexpressions got the same suspiciousness score as the faulty subexpression, we considered the faulty subexpression to rank in the middle of them.

5.4. Comparative Performance of NUMFL vs. Baselines

Table VIII shows, for NUMFL-GPS and the baseline metrics and for each subject program, the average percentage of subexpressions that had to be examined to find the fault, computed across all the faulty versions. Here we ~~show~~ focus on the results for QRM but not for DLRM, which QRM outperformed. (Nevertheless, DLRM outperformed the 5 baseline metrics.) For 14 of 16 subject programs, NUMFL-GPS-QRM performed better than Ochiai, DStar ($star = 2$), and SOBER. NUMFL-GPS-QRM performed better than ESP-SIV (single variable) and ESP-SCP (scalar pair) for all 16 subject programs. Overall, NUMFL-GPS-QRM was more effective in localizing numerical faults than any of the baseline metrics.

Table VIII. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM AND BASELINE METRICS ON SINGLE-FAULT PROGRAM VERSIONS

Subject Program	**Technique**						
	NUMFL-GPS-DLRM	NUMFL-GPS-QRM	Ochiai	Dstar	ESP(SIV)	ESP(SCP)	SOBER
Apache_EigenDecompose	24.5%	12.3%	9.2%	9%	22.2%	17.2%	7.8%
Apache_DScompiler	22.1%	11%	19.9%	16.5%	22.9%	18.9%	24.2%
Apache_BigMatrix	16.5%	11.4%	51.8%	51.8%	37.9%	31.8%	28.9%
Apache_Rotation3D	11.7%	8%	30.5%	30.5%	20.1%	23.6%	28.5%
Ojaljo_SchurDecompose	5.0%	11.4%	22.7%	22.7%	20.2%	27.3%	30.9%
Jama_MatrixDecompose	40.1%	14.2%	11.4%	11.4%	24.3%	27%	46.4%
SciMark_LU	19.4%	15.3%	38.9%	68.1%	27.8%	18.8%	12.5%
SciMart_FFT	5.6%	9.5%	48.3%	75%	68.1%	36.6%	12.5%
Apache_SymmLQ	6.1%	2.6%	37.7%	85.1%	7%	10.5%	32.9%
Apache_SplineInterpolator	53.9%	31.7%	51.1%	47.8%	65%	64.4%	56.1%
Apche_SimpleRegress	2.5%	4.3%	49.2%	29.9%	7.3%	7.2%	7.9%
Apache_SchurTransformer	16.8%	3.3%	8.2%	8.2%	36.2%	10.2%	44.4%
Apache_MillerUpdatRegress	37.7%	7.5%	12.7%	12.7%	37.6%	19.6%	14.5%
Apache_HarmonicFitter	37.3%	27.3%	45%	58.5%	39.5%	41.7%	49.3%
Apache_FastSine	34.6%	1.5%	30.2%	87.1%	41.9%	3.8%	26.7%
Apache_FastCosine	42.8%	1.3%	33.2%	94.9%	79.7%	41.4%	31.2%
Average Cost	23.5%	10.8%	31.3%	44.3%	34.9%	25%	28.4%

DOI: 10.1002/stvr

From Table VIII, we note that coverage based SFL metrics, Ochiai and DStar, have slightly better performance than NUMFL-GPS-QRM on two subject programs: *Apache_EigenDecompose* and *Jama_MatrixDecompose*. To explain this result, we analyze the faulty versions that coverage based SFL performs better than NUMFL-GPS-QRM. We found that the coverage based SFL performed well in fault localization for numerical programs when both of the following two requirements are met. First, the subject programs should have relative many conditional branches. In this empirical study, *Apache_EigenDecompose* and *Jama_MatrixDecompose* have most conditional branches among all the subject programs. Second, if the tests cover the faulty expression, the program has a high failure rate (¿95%). In the subject programs *Apache_EigenDecompose* and *Jama_MatrixDecompose*, coverage based SFL performed well if the injected fault type is "randomly mutating operator". If we randomly change the operator of an expression, the executions of the program often have large output errors that exceed the pre-defined tolerance, and thus are labeled as failing runs.

We also compare the performance of NUMFL-GPS-QRM to that of each baseline SFL metric graphically. The comparison measure is the difference between the cost for the baseline metric and the cost for NUMFL-GPS-QRM. Since the two costs are percentages, the difference is the percentage reduction in cost obtained by using NUMFL-GPS-QRM, which may be positive or negative. For example, if the cost of using NUMFL-GPS-QRM is 10% and the cost of using the baseline metric is 40%, then the improvement is 30%. Figure 6 shows the results of comparisons of NUMFL-GPS-QRM with each of the baseline metrics. In each graph, the vertical axis represents the percentage improvement (reduction) in cost. The horizontal-axis represents different subject-program versions for which there are cost differences between the metrics, with each version represented by a vertical bar. Bars above the zero-line represent versions for which NUMFL-GPS-QRM performed better than the baseline metric and bars below zero represent versions for which NUMFL-GPS-QRM performed worse. The length of each bar represents the magnitude of the corresponding cost difference.

NUMFL-GPS-QRM vs. Ochiai. Over all 92 faulty subject-program versions, NUMFL-GPS-QRM performed better than the Ochiai metric on 67 versions but NUMFL-GPS-QRM performed worse on 25 versions. There were 38 versions for which NUMFL-GPS-QRM performed at least 20% better than the Ochiai metric. There were just 5 versions for which the latter performed better than NUMFL-GPS-QRM.

NUMFL-GPS-QRM vs. DStar. NUMFL-GPS-QRM performed better than DStar on 70 subject-program versions but NUMFL-GPS-QRM performed worse than DStar on 22 versions. NUMFL-GPS-QRM performed at least 20% better than DStar on 42 versions, whereas DStar performed at least 20% better than NUMFL-GPS-QRM on just 5 versions.

NUMFL-GPS-QRM vs. ESP-SIV. NUMFL-GPS-QRM performed better than ESP-SIV on 70 subject-program versions but NUMFL-GPS-QRM performed worse than ESP-SIV on 22 versions. NUMFL-GPS-QRM performed at least 20% better than ESP-SIV on 31 versions, whereas ESP-SIV performed at least 20% better than NUMFL-GPS-QRM on just 5 versions.

QRM vs. ESP-SCP. NUMFL-GPS-QRM performed better than ESP-SCP on 61 subject-program versions but NUMFL-GPS-QRM performed worse than ESP-SCP on 31 versions. NUMFL-GPS-QRM performed at least 20% better than ESP-SIV on 27 versions, whereas ESP-SCP performed at least 20% better than NUMFL-GPS-QRM on just 3 versions.

QRM vs. SOBER. NUMFL-GPS-QRM performed better than SOBER on 64 subject-program versions but NUMFL-GPS-QRM performed worse than SOBER on 28 versions. NUMFL-GPS-QRM performed at least 20% better than SOBER on 32 versions, whereas SOBER performed at least 20% better than NUMFL-GPS-QRM on just 7 versions.

5.5. NUMFL-GPS Applied only to Failing Runs

Conventional, coverage-based and predicate-based SFL techniques require profiles and PASS/FAIL labels from both passing and failing runs, in order to rank statements based on estimates of quantities like $Pr[failure|s\ covered]$ [7] that vary between statements only if the data come from a mix of passing and failing runs. However, because many numerical programs have few conditional

DOI: 10.1002/stvr

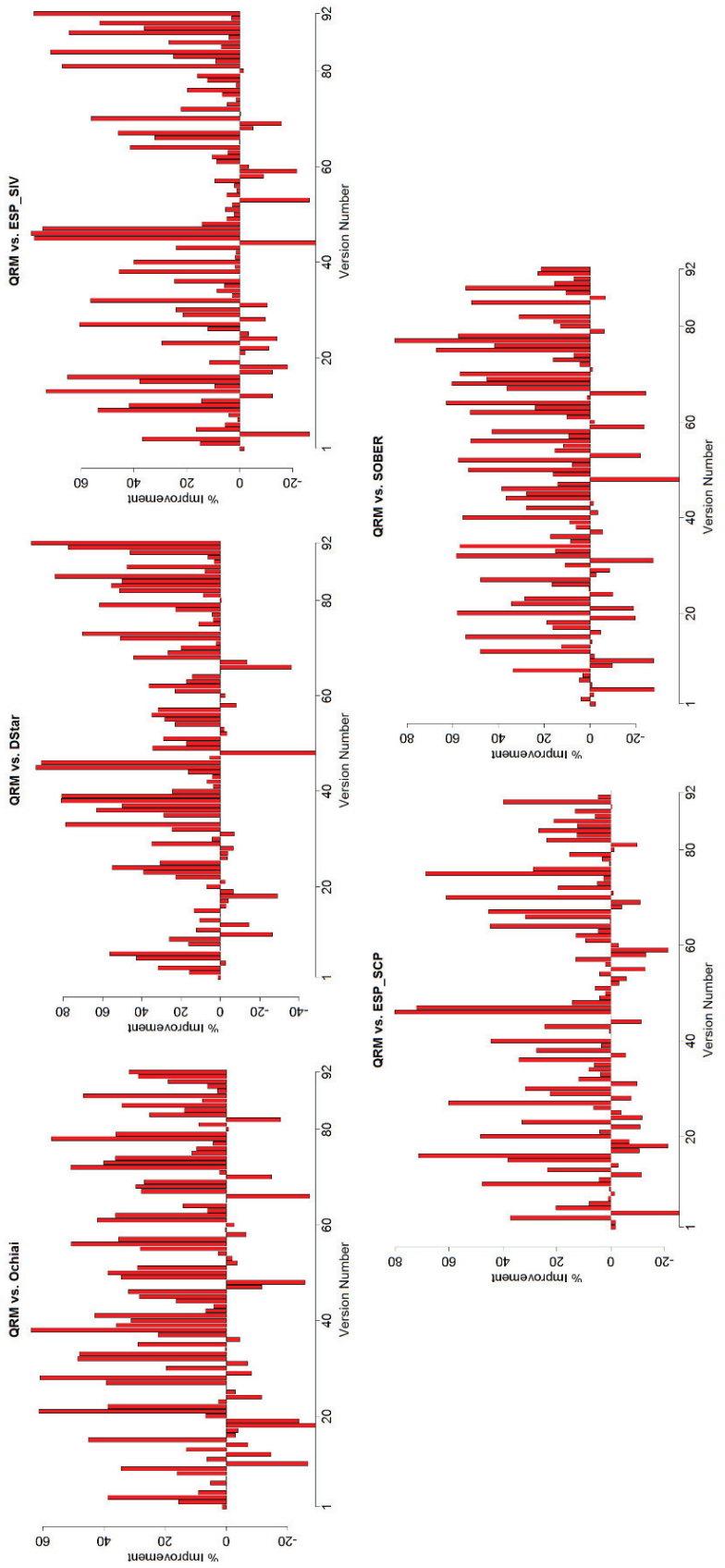

Figure 6. Performance of NUMFL-GPS-QRM relative to baseline metrics on individual single-fault program versions.

DOI: 10.1002/stvr

branches, a particular fault may be executed and cause erroneous values to occur on every run that is observed. If these values propagate to the program output, a failure may occur on every run, depending on the numerical accuracy required.

Even if it is applied to data from only failing runs, NUMFL can still produce different AFCE estimates for different statements, because the causal outcome of interest is a continuous variable– the output error of a numerical program. To evaluate whether NUMFL-GPS is actually effective when it is applied to data from only failing runs, we redid the study just described, after creating a new data set by removing the data from passing runs. Table IX and Figure 7 compare the performance of QRM with such data to its performance with the original data from both passing and failing runs. Figure 7 shows that over all 92 faulty subject-programs versions, QRM performed better with data from only failing runs on 43 program versions and it performed worse on 49 versions. There were only 8 versions for which QRM performed at least 20% worse with data from only failing runs.

Table IX. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM WITH AND WITHOUT DATA FROM PASSING RUNS, ON SINGLE-FAULT PROGRAM VERSIONS

Subject Program	**Input**	
	Pass and Fail	Fail only
Apache_EigenDecompose	12.3%	11.4%
Apache_DScompiler	11%	11%
Apache_BigMatrix	11.4%	8%
Apache_Rotation3D	8%	18.9%
Ojaljo_SchurDecompose	11.4%	17.7%
Jama_MatrixDecompose	14.2%	24.5%
SciMark_LU	15.3%	54.1%
SciMart_FFT	9.5%	17.2%
Apache_SymmLQ	2.6%	2.6%
Apache_SplineInterpolator	31.7%	21.7%
Apche_SimpleRegress	4.3%	3.4%
Apache_SchurTransformer	3.3%	8.2%
Apache_MillerUpdatRegress	7.5%	12.7%
Apache_HarmonicFitter	27.3%	47.5%
Apache_FastSine	1.5%	1.5%
Apache_FastCosine	1.3%	1.3%
Average Cost	10.8%	16.4%

From Table IX and Figure 7, we can see that NUMFL works fairly well for most subject programs when used with data from only failing runs, although its performance is generally better with data from both passing and failing runs. Note that with many numerical programs the tolerance for output errors is very small (between 1.0E-5 and 1.0E-12). For such programs, the output errors from passing runs are clustered around 0, while output errors from failing runs are larger and more varied. Consequently, the data from failing runs carries more information about the dose-response function than does the data from the passing runs. Hence, using only data from failing runs to estimate the DRF does not necessarily result in severe bias.

5.6. Application of NUMFL-GPS to Programs with Multiple Faults

The empirical results presented in Section 5.4 indicate that NUMFL-GPS performs better than the baseline techniques on the subject program versions containing a single fault. This section reports on the application of NUMFL-GPS to subject program versions with multiple faults. For this sub-study, we selected five of the larger subject programs (shown in Table X) and generated 5 faulty versions of each for a total of 25 faulty versions. To create each faulty version, we randomly injected two simulated faults. (Injecting a larger number of faults in our programs tended to make them fail badly

DOI: 10.1002/stvr

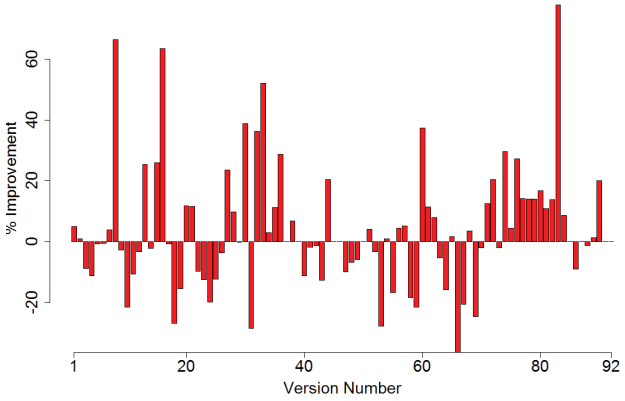

Figure 7. Relative performance of NUMFL-GPS-QRM with and without data from passing runs, on individual single-fault program versions.

on all inputs). Table X shows the average cost, over the faulty versions of each subject program, of localizing the first fault to be found, for NUMFL-GPS-QRM and each of the baseline techniques. The table shows that NUMFL-GPS-QRM outperformed the other baseline techniques on average, although SOBER and Ochiai each performed better on one of the subject programs.

Table X. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM AND BASELINE METRICS ON TWO-FAULTs PROGRAM VERSIONS

Subject Program	**Technique**					
	NUMFL-GPS-QRM	Ochiai	Dstar	ESP(SIV)	ESP(SCP)	SOBER
Apache_EigenDecompose	10.3%	20%	19.4%	28.3%	25.8%	6.3%
Apache_DScompiler	4.5%	31.5%	23.2%	19.9%	10.4%	9%
Apache_Rotation3D	7%	3.6%	33.6%	17.5%	17.9%	25.8%
Ojaljo_SchurDecompose	5.4%	22.7%	27.1%	8.4%	9.6%	27.1%
Jama_MatrixDecompose	14.1%	26.3%	16.5%	26.3%	26.2%	15.5%
Average Cost	8.3%	20.8%	24%	20.1%	18%	16.7%

Figure 8 graphically contrasts the performance of NUMFL-GPS-QRM on the individual two-fault program versions with that of the baseline metrics. It is evident that SOBER's performance is the second best after NUMFL-GPS-QRM. Over all 25 faulty subject-program versions, NUMFL-GPS-QRM performed better than SOBER on 15 versions but performed worse on 10 versions. There were 8 versions for which NUMFL-GPS-QRM performed at least 10% better than SOBER but only one version for which SOBER performed at least 10% better than NUMFL-GPS-QRM. Both Ochiai and ESP-SCP performed better than NUMFL-GPS-QRM on 8 versions but performed worse on 17 versions. NUMFL-GPS-QRM performed better than ESP-SIV on 19 versions and performed better than DStar on 21 versions.

5.7. Comparison of NUMFL-GPS and NUMFL-CBPS

In this section, we report the results of empirically comparing the performance of NUMFL-GPS-QRM with that of NUMFL-CBPS-QRM. We applied the two techniques to the subject program versions with single faults as well as to the versions with two faults. The average fault localization costs on the single fault versions is shown in Table XI. There were 13 versions for which NUMFL-GPS-QRM performed better than NUMFL-CBPS-QRM. There were only 3 versions for which NUMFL-CBPS-QRM performed better than NUMFL-GPS-QRM. Figure 9 graphically contrasts the performance of the two methods on the single fault versions. NUMFL-GPS-QRM performed

Prepared using stvrauth.cls
DOI: 10.1002/stvr

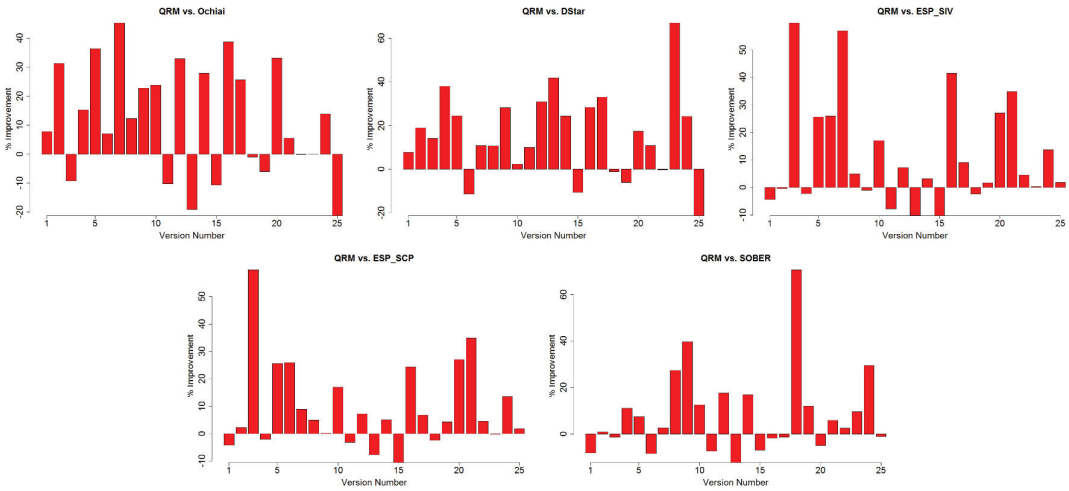

Figure 8. Performance of GPS-QRM relative to baseline metrics on individual two-fault program versions .

better on 61 versions, while NUMFL-CBPS-QRM performed better on 31 versions. NUMFL-GPS-QRM performed at least 20% better than NUMFL-CBPS-QRM on 15 versions, whereas NUMFL-CBPS-QRM performed at least 20% better than NUMFL-GPS-QRM on just 4 versions.

Table XI. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM AND NUMFL-CBPS-QRM ON SINGLE-FAULT PROGRAM VERSIONS

Subject Program	**NUMFL**	
	GPS-QRM	CBPS-QRM
Apache_EigenDecompose	12.3%	8.8%
Apache_DScompiler	11%	12%
Apache_BigMatrix	11.4%	9.9%
Apache_Rotation3D	8%	18.9%
Ojaljo_SchurDecompose	11.4%	14%
Jama_MatrixDecompose	14.2%	23%
SciMark_LU	15.3%	27.8%
SciMart_FFT	9.5%	19.8%
Apache_SymmLQ	2.6%	4.4%
Apache_SplineInterpolator	31.7%	35%
Apche_SimpleRegress	4.3%	17%
Apache_SchurTransformer	3.3%	14.9%
Apache_MillerUpdatRegress	7.5%	9.9%
Apache_HarmonicFitter	27.3%	32%
Apache_FastSine	1.5%	13%
Apache_FastCosine	1.3%	1.2%
Average Cost	10.8%	16.4%

Table XII shows the average cost, over the faulty versions of each subject program into which two faults were injected, of localizing the first fault to be found, for both NUMFL-GPS-QRM and NUMFL-CBPS-QRM. NUMFL-GPS-QRM performed better than NUMFL-CBPS-QRM on the two-fault versions of 4 subject programs. NUMFL-GPS-QRM performed worse than NUMFL-CBPS-QRM on the versions of 1 subject program. Figure 10 graphically contrasts the performance of the two methods on the individual two-fault versions. NUMFL-GPS-QRM performed better than

DOI: 10.1002/stvr

NUMFL-CBPS-QRM on 13 versions and NUMFL-GPS-QRM performed worse on 12 versions. NUMFL-GPS-QRM performed at least 10% better than NUMFL-CBPS-QRM on 7 versions, whereas NUMFL-CBPS-QRM performed at least 10% better than NUMFL-GPS-QRM on just 1 version.

In summary, NUMFL-GPS-QRM performed better than NUMFL-CBPS-QRM on both single-fault and two-fault programs. The generalized propensity score model $\boldsymbol{\theta} = \boldsymbol{X}'\beta$ achieved better covariate balance than the multinomial logistic regression model used with CBPS. This may be due to the composition of the data collected from our subject program versions. The performance of NUMFL-CBPS was poorest with versions that had no more than about 150 failing executions. However, although NUMFL-CBPS performed worse than NUMFL-GPS in our study, the former's average fault localization cost was still lower than those of the baseline techniques.

Table XII. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM AND NUMFL-CBPS-QRM ON MULTIPLE-FAULT PROGRAM VERSIONS

Subject Program	**NUMFL**	
	GPS-QRM	CBPS-QRM
Apache_EigenDecompose	10.3%	11.1%
Apache_DScompiler	4.5%	5.9%
Apache_Rotation3D	7.0%	3.0%
Ojaljo_SchurDecompose	5.4%	19.8%
Jama_MatrixDecompose	14.1%	23.4%
Average Cost	8.26%	12.64%

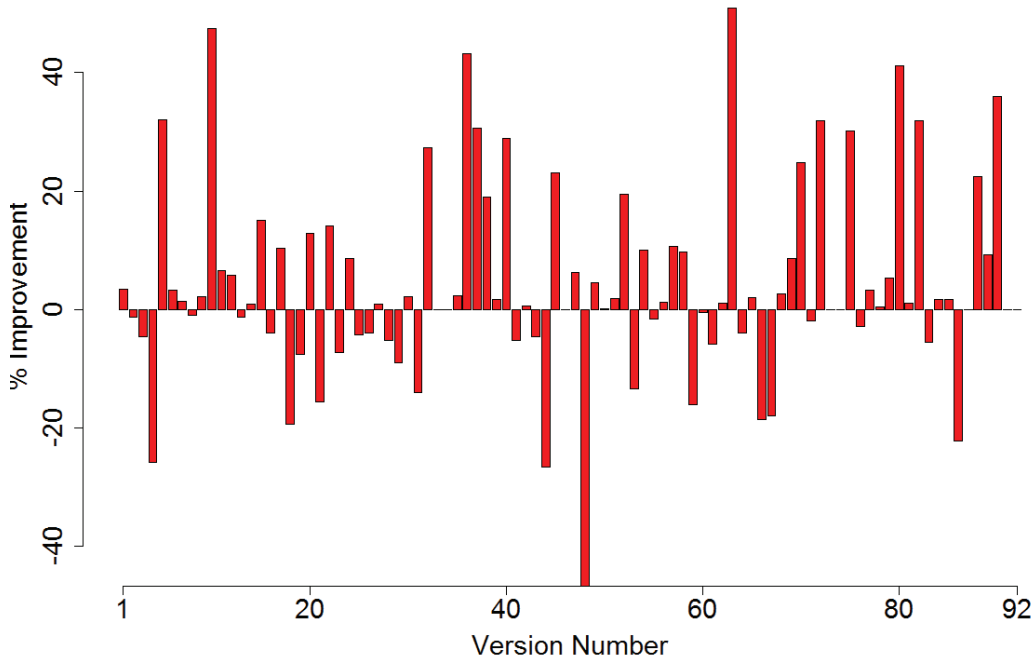

Figure 9. . Relative performance of NUMFL-GPS-QRM and NUMFL-CBPS-QRM on individual single-fault program versions .

DOI: 10.1002/stvr

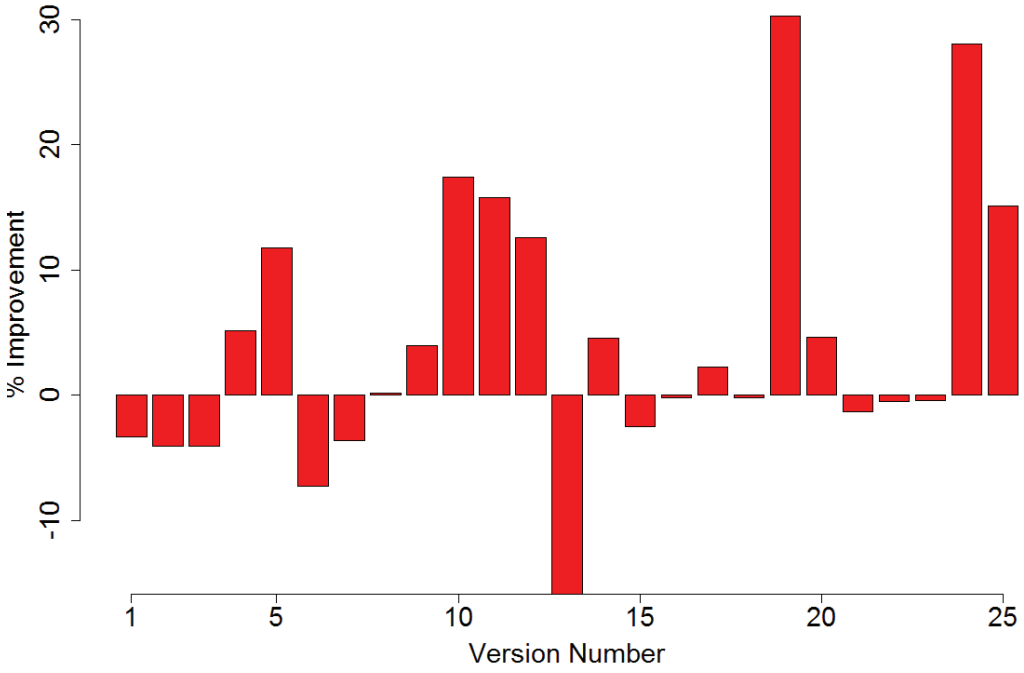

Figure 10. Relative performance of NUMFL-GPS-QRM and NUMFL-CBPS-QRM on individual two-fault program versions .

5.8. *Sensitivity of NUMFL-GPS-QRM to the Number of Subclasses*

In previous empirical evaluations, we choose the number of classes $m = 10$, so the observations are grouped into 10 roughly equal size subclasses based on their GPSs or CBPSs. In this section, we examine whether if increasing or decreasing m would influence the performance of NUMFL. We applied NUMFL-GPS-QRM on single-fault program versions with $m = 5$, $m = 10$ and $m = 15$. The result is shown in Table XIII. From Table XIII, it can be seen that when we increased m from 5 to 10, NUMFL-GPS-QRM has better performance on all 6 large subject programs. But there are 4 small subject programs where NUMFL-GPS-QRM's performance became worse after the number of subclasses was increased. The average cost across 16 subject programs is improved from 15.4% to 10.8%. When we increased m from 10 to 15, NUMFL-GPS-QRM's performance is improved on 5 out of 6 large subject programs. For small subject programs, $m = 5$ and $m = 10$ have similar performance. The average cost is roughly the same for $m = 10$ and $m = 15$. This suggests that $m = 10$ is enough to get good AFCE estimation with programs that are similar to those used in the study. The result also indicates that increasing number m will only improve the performance of NUMFL-GPS-QRM on large subject programs, but not small subject programs. The reason is most small subject programs have less than 70 subexpressions. So a small change of rank may reflect a big change in the percentage of code need to be examine. This makes the result of small subject programs are not as reliable as the result of large subject programs.

5.9. *Computation Time*

We used two machines for this study: (1) a Dell Precision T5600 with two 2.30 GHz intel Xeon CPUs and 64 GB RAM; and (2) a Dell PowerEdge T300 with a 2.83 GHz Intel Xeon CPU and 24GB RAM. The computation time required to apply any of the non-causal baseline techniques was less than 1 minute per program version. The computation time for NUMFL-QRM and NUMFL-CBPS per version are summarized in table XIV

DOI: 10.1002/stvr

Table XIII. AVERAGE FAULT LOCALIZATION COSTS OF NUMFL-GPS-QRM WITH DIFFERENT NUMBER OF SUBCLASSES ON SINGLE-FAULT PROGRAM VERSIONS

Subject Program	**NUMFL-GPS-QRMl**		
	$m = 5$	$m = 10$	$m = 15$
Apache_EigenDecompose	17.4%	12.3%	10.8 %
Apache_DScompiler	15.2%	11%	10 %
Apache_BigMatrix	13.1%	11.4%	10 %
Apache_Rotation3D	8.9%	8%	10.2 %
Ojaljo_SchurDecompose	20.7%	11.4%	10 %
Jama_MatrixDecompose	22.8%	14.2%	12.4 %
SciMark_LU	33.3%	15.3%	8.3 %
SciMart_FFT	33.6%	9.5%	21.5 %
Apache_SymmLQ	1.8%	2.6%	2.6 %
Apache_SplineInterpolator	26.1%	31.7%	20.1 %
Apche_SimpleRegress	8.4%	4.3%	7 %
Apache_SchurTransformer	13.2%	3.3%	3 %
Apache_MillerUpdatRegress	2.6%	7.5%	10 %
Apache_HarmonicFitter	26.3%	27.3%	35.3 %
Apache_FastSine	1.5%	1.5%	1.5 %
Apache_FastCosine	1.3%	1.3%	1.3 %
Average Cost	15.4%	10.8%	10.8 %

Table XIV. AVERAGE COMPUTATION TIME OF NUMFL-GPS-QRM AND NUMFL-CBPS-QRM ON ONE SINGLE-FAULT PROGRAM VERSIONS

Subject Program	**NUMFL**	
	GPS-QRM(Seconds)	CBPS-QRM (Seconds)
Apache_EigenDecompose	452.32	689.16
Apache_DScompiler	104.66	143.92
Apache_BigMatrix	122.98	170.83
Apache_Rotation3D	427.53	646.92
Ojaljo_SchurDecompose	463.44	585
Jama_MatrixDecompose	403.72	507.33
SciMark_LU	24.72	54.8
SciMart_FFT	38.20	88.83
Apache_SymmLQ	64.98	240.31
Apache_SplineInterpolator	31.7	69.93
Apche_SimpleRegress	54.74	188.30
Apache_SchurTransformer	73.62	130.20
Apache_MillerUpdatRegress	97.05	65.52
Apache_HarmonicFitter	58.94	69.68
Apache_FastSine	46.59	140.40
Apache_FastCosine	72.75	154.26
Average Cost	85.79	246.59

From XIV, The NUMFL is mildly slower than baseline techniques. The NUMFL-GPS-QRM's computation time is between 2 minutes and 8 minutes for large subject programs and is between 20 seconds to 100 seconds for small subject programs.The NUMFL-CBPS-QRM's computation time is between 2 minutes and 12 minutes for large subject programs and is between 50 seconds to 4 minutes for small subject programs. It is important to mention that we did not attempt to parallelize

DOI: 10.1002/stvr

the computation of AFCE estimates, even though AFCE estimates for different sub-expression can be computed independently.

5.10. Limitations

NUMFL and its evaluation are subject to several limitations. First, NUMFL employs regression models for two purposes: estimating generalized propensity scores and estimating the causal effect of the treatment on the outcome. We used Gaussian (Normal) linear models for both purposes in NUMFL-GPS, because they are supported by fast and robust software, and they performed well in our empirical study. In NUMFL-CBPS, we use multinomial logistic regression propensity score models. However, other model choices may be more appropriate for applying NUMFL to different numerical software. Determining that would require preliminary analysis of the data. A related issue is that sufficient data must be available to adequately fit each model. We have found that the number of failing tests should at least 100 to fit adequate regression models. This is not a serious problem, since faults in numerical programs are usually not guarded by many branch conditions that are likely to cause "coincidental correctness." A limitation of our empirical study is that the seeded faults were of three basic types. In future work, we intend to evaluate NUMFL on subject programs with more varied faults. Finally, validity of NUMFL (with QRM) depends on Assumptions 1 and 3 in section 4. These assumptions do not always hold in practice. For example, if the effect of a numerical fault does not propagate to the output (coincidental correctness), then Assumption 3 will be violated.

6. RELATED WORK

The seminal SFL research of Jones et al. [3], which is coverage based, and of Liblit et al. [4], which is predicate based, has inspired much subsequent research. The use of causal inference methodology in SFL is due to Baah et al. [7]. They showed that estimating the AFCE of covering a statement s using a linear regression model that adjusts for coverage of the direct control dependence predecessor of s is effective for reducing confounding bias that often distorts fault localization scores. Later, Baah et al. [8] extended this work by using covariate matching to control confounding bias involving both control and data dependence predecessors. More recently, Gore et al. [9] extended Baah's initial approach by providing a model that reduces what they called "failure flow confounding bias," which involves predicate outcomes. Recently, Shu et al. [10] proposed a method-level causal inference model to localize faulty methods in large programs. The aforementioned techniques, unlike NUMFL, do not use values of program variables in fault localization.

State-altering techniques, such as *cause-transitions* [32] and *value replacement* [33] are among the few fault localization techniques based on values of variables. Cause-transitions employs *delta debugging* [34] to isolate the cause of a program failure. Value replacement localizes faults by switching program states and re-running the program. Unlike SFL techniques, these techniques actually alter program states, and they require an oracle to determine if alterations cause program failures. On the other hand, they are non-statistical and do not require a sample of both passing and failing executions. The *Daikon* system identifies possible invariant conditions in a program based on a sample of executions [35]. The *DIDUCE* system uses Daikon to report violations of dynamic invariants, which may indicate failures [36]. However, these techniques do not address confounding bias.

Some recent studies have developed techniques to analyze instability in floating-point computations. *CADNA* [37] is a library to estimate round-off errors with Monte-Carlo Arithmetic. Tang et al. [38] proposed a technique to automatically detect such errors by systematically altering the underlying numerical calculation. Lam et al. [39] proposed to use the detection of significant digit cancellation events to test the precision of numerical programs. Later, Zhang et al. [40] extended Lam's work by monitoring the propagation of cancelled bits. These studies focused on detecting accumulated round-off errors due to finite-precision computation, but they do not address the problem of localizing faults in numerical expressions generally.

DOI: 10.1002/stvr

Other researchers have proposed alternatives to Imai and van Dyk's approach to defining and using the generalized propensity score. Hirano and Imbens estimate the causal effect of a continuous treatment variable by fitting one regression model with the GPS as a predictor [13]. This method requires discretizing the treatment variable when calculating the propensity score. Zhao and Imai extend Hirano and Imbens's work by using a smooth coefficient model (SCM) in the regression, so the model can estimate the dose-response function of the treatment variable instead of average causal effect [17]. In preliminary work (with different subject programs), we found that these GPS methods did not perform as well as Imai and Van Dyk's approach, which is used in this paper.

Other related work involve model-based debugging (MBSD) which is an application of model-based diagnosis techniques to localize faults in softwares. Wolfgang et al. proved that MBSD outperforms traditional debugging techniques like "slicing" in fault localization [48]. Rui et al. integrates MBSD with spectrum based fault localization to refine the ranking of statements citeWolfgang2009, Abreu2009. Wotawa et al. proposed a MBSD approach relies on constraint solver which is effective in debugging smaller programs[51].

Wotawa et al. have also done extensive constraint-based debugging of spreadsheets

7. CONCLUSION

In this paper, we have presented and evaluated a value-based causal model, denoted NUMFL, for localizing faults in numerical software. NUMFL employs generalized propensity scores and covariate balancing propensity scores to control confounding bias involving floating-point program variables that carry erroneous values to correct statements. NUMFL uses a quadratic regression model (QRM) to estimate the average failure-causing effect of a numerical expression. We reported the results of an empirical comparison of NUMFL to several competing techniques on both single-fault subject programs and multiple-fault subject programs. NUMFL performed notably better than the other techniques. We also found that NUMFL-GPS works well with data from failing *runs alone*. Finally, we compared the performance two versions of NUMFL, denoted NUMFL-GPS and NUMFL-CBPS, based on the two aforementioned types of propensity scores. We found that the NUMFL-GPS performed better than NUMFL-CBPS for both single-fault programs and two-fault programs. In the future work, will seek to extend our empirical results to a broader range of subject programs with more varied fault types. We intend eventually to evaluate NUMFL in a user study, but given the difficulty of conducting an unbiased one, we think it is desirable to refine NUMFL as much as possible beforehand. Finally, we will also explore the integration of NUMFL with coverage or predicate based causal SFL techniques.

8. ACKNOWLEDGEMENTS

This research was supported by NSF awards CNS-1035602 and CCF-1525178.

REFERENCES

1. Vuik K. *Some disasters caused by numerical errors*. Website at: http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html.
2. Kanewala U, Bieman JM. 2014. *Testing scientific software: A systematic literature review*. Information and software technology, 56(10), 1219-1232.
3. Jones JA, Harrold MJ, Stasko J. 2002. *Visualization of test information to assist fault localization*. In Proceedings of the 24th international conference on Software engineering, 467-477.
4. Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. 2005. *Scalable statistical bug isolation*. In ACM SIGPLAN Notices. 40(6), 15-26.
5. Liu C, Yan X, Fei L, Han J, Midkiff SP. 2005. *SOBER: statistical model-based bug localization*. ACM SIGSOFT Software Engineering Notes. 30(5), 286-295.
6. Fenton NE, Neil M. 1999. *A critique of software defect prediction models*. Software Engineering, IEEE Transactions on. 25(5), 675-689.
7. Baah GK, Podgurski A, Harrold MJ. 2010. *Causal inference for statistical fault localization*. In Proceedings of the 19th international symposium on Software testing and analysis. 73-84.

DOI: 10.1002/stvr

8. Baah GK, Podgurski A, Harrold MJ. 2011. *Mitigating the confounding effects of program dependences for effective fault localization*. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 146-156.
9. Gore R, Reynolds Jr PF. 2012. *Reducing confounding bias in predicate-level statistical debugging metrics*. In Proceedings of the 34th International Conference on Software Engineering. 463-473.
10. Shu G, Sun B, Podgurski A, Cao F. 2013. *Mfl: Method-level fault localization with causal inference*. In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. 124-133.
11. Pearl, J. 2003. *Causality: models, reasoning, and inference*. Econometric Theory. 19, 675-685.
12. Imai K, Van Dyk DA. 2004. *Causal inference with general treatment regimes*. Journal of the American Statistical Association. 99(467).
13. Hirano K, Imbens GW. 2004. *The propensity score with continuous treatments*. Applied Bayesian modeling and causal inference from incomplete-data perspectives. 226164, 73-84.
14. S. Philip. *Pearson's correlation coefficient*.BMJ British Medical Journal 345, 2012.
15. Grossman J, Mackenzie FJ. 2005. *The randomized controlled trial: gold standard, or merely standard?*. Perspectives in biology and medicine. 48(4), 516-534.
16. Rosenbaum PR, Rubin DB. 1983. *The central role of the propensity score in observational studies for causal effects*. Biometrika. 70(1), 41-55.
17. Zhao S, van Dyk DA, Imai K. 2013. *Propensity-Score Based Methods for Causal Inference in Observational Studies with Fixed Non-Binary Treatments*.
18. Kubáček, L. 1996. *Quadratic regression models*. Mathematica Slovaca. 46(1), 111-126.
19. Nachtsheim CJ, Neter J, Kutner MH, Wasserman W. 2004. *Applied linear regression models*. McGraw-Hill Irwin. 47.
20. Java 1.5 Parser Available : https://code.google.com/p/javaparser/
21. ASM Framework Available: http://asm.ow2.org/
22. NUMFL Profiler: https://github.com/zhuofubai/Subexpression_Profiler
23. R Project Available: http://www.r-project.org/
24. Commons Math: http://commons.apache.org/proper/commons-math/
25. Oj! Algorithms Available: http://ojalgo.org/getting.html
26. JAMA package Available: http://math.nist.gov/javanumerics/jama/
27. SciMark 2.0 Available: http://math.nist.gov/scimark2/
28. Abreu R, Zoeteweij P, Van Gemund AJ. 2007. *On the accuracy of spectrum-based fault localization*. In Testing: Academic and Industrial Conference Practice and Research Techniques. 89-98.
29. Wong WE, Debroy V, Gao R, Li Y. 2014. *The dstar method for effective software fault localization*. Reliability, IEEE Transactions on. 63(1), 290-308.
30. Gore R, Reynolds Jr PF, Kamensky D. 2011. *Statistical debugging with elastic predicates*. In Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. 492-495. IEEE.
31. Liblit BR. *Cooperative Bug Isolation*. Diss. UNIVERSITY OF CALIFORNIA, BERKELEY, 2004.
32. Cleve H, Zeller A. 2005. *Locating causes of program failures*. In Proceedings of the 27th international conference on Software engineering. 342-351.
33. Jeffrey D, Gupta N, Gupta R. 2008. *Fault localization using value replacement*. In Proceedings of the 2008 international symposium on Software testing and analysis. 167-178.
34. Zeller A. 2002. *Isolating cause-effect chains from computer programs*. In Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering. 1-10.
35. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C. 2007. *The Daikon system for dynamic detection of likely invariants*. Science of Computer Programming. 69(1), 35-45.
36. Hangal S, Lam MS. 2002. *Tracking down software bugs using automatic anomaly detection*. In Proceedings of the 24th international conference on Software engineering. 291-301.
37. Scott NS, Jézéquel F, Denis C, Chesneaux JM. 2007. *Numerical health checkfor scientific codes: the CADNA approach*. Computer Physics Communications. 176(8), 507-521.
38. Tang E, Barr E, Li X, Su Z. 2010. *Perturbing numerical calculations for statistical analysis of floating-point program (in) stability*. In Proceedings of the 19th international symposium on Software testing and analysis. 131-142.
39. Lam MO, Hollingsworth JK, Stewart GW. 2013. *Dynamic floating-point cancellation detection*. Parallel Computing. 39(3), 146-155.
40. Bao, T., Zhang, X. 2013. On-the-fly detection of instability problems in floating-point program execution. In ACM SIGPLAN Notices. 48(10), 817-832.
41. Imai, K., Ratkovic, M. 2014. Covariate balancing propensity score. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 76(1), 243-263.
42. Hansen, L. P. 1982. Large sample properties of generalized method of moments estimators. Econometrica: Journal of the Econometric Society, 1029-1054.
43. Rosenblueth, E. (1981). Two-point estimates in probabilities. Applied Mathematical Modelling, 5(5), 329-335.
44. CBPS R package Available: http://imai.princeton.edu/software/CBPS.html
45. Bai, Z., Shu, G., Podgurski, A. (2015, April). NUMFL: Localizing Faults in Numerical Software Using a Value-Based Causal Model. In Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on(pp. 1-10). IEEE.
46. Silvapulle, Mervyn J. "On the existence of maximum likelihood estimators for the binomial response models." Journal of the Royal Statistical Society. Series B (Methodological) (1981): 310-313.
47. Harrold, Mary Jean. "Testing: a roadmap." Proceedings of the conference on the future of software engineering. ACM, 2000.
48. Mayer, Wolfgang, and Markus Stumptner. "Evaluating models for model-based debugging." Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2008.

DOI: 10.1002/stvr

49. Mayer, Wolfgang, et al. ”Prioritising model-based debugging diagnostic reports.” Proceedings of the International Workshop on Principles of Diagnosis (DX). 2009.
50. Abreu, Rui, et al. ”Refining spectrum-based fault localization rankings.” Proceedings of the 2009 ACM symposium on Applied Computing. ACM, 2009.
51. Wotawa, Franz, Mihai Nica, and Iulia Moraru. ”Automated debugging based on a constraint model of the program and a test case.” The journal of logic and algebraic programming 81.4 (2012): 390-407.

DOI: 10.1002/stvr