Software Reliability via Run-Time Result-Checking

Hal Wasserman
University of California, Berkeley
and
Manuel Blum
City University of Hong Kong and
University of California, Berkeley

We review the field of **result-checking,** discussing **simple checkers** and **self-correctors.** We argue that such checkers could profitably be incorporated in software as an aid to efficient debugging and enhanced reliability. We consider how to modify traditional checking methodologies to make them more appropriate for use in real-time, real-number computer systems. In particular, we suggest that checkers should be allowed to use **stored randomness**: i.e., that they should be allowed to generate, preprocess, and store random bits prior to run-time, and then to use this information repeatedly in a series of run-time checks. In a case study of checking a general real-number linear transformation (for example, a Fourier Transform), we present a simple checker which uses stored randomness, and a self-corrector which is particularly efficient if stored randomness is employed.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; F.2.1 [**Analysis of Algorithms**]: Numerical Algorithms—*Computation of transforms*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Reliability, Algorithms, Verification

Additional Key Words and Phrases: Built-in testing, concurrent error detection, debugging, fault tolerance, Fourier Transform, result-checking, self-correcting

This work was supported in part by a MICRO grant from Hughes Aircraft Company and the State of California, by NSF Grant CCR92-01092, and by NDSEG Fellowship DAAH04-93-G-0267.
A preliminary version of this paper appears as: "Program result-checking: a theory of testing meets a test of theory," *Proc. 35th IEEE Symp. Foundations of Computer Science* (1994), pp. 382–392.
Name: Hal Wasserman
Affiliation: Computer Science Division, University of California at Berkeley
Address: Berkeley, CA 94720, halw@cs.berkeley.edu, http.cs.berkeley.edu/~halw
Name: Manuel Blum
Affiliation: Computer Science Department, City University of Hong Kong
Address: 83 Tat Chee Ave., Kowloon Tong, Kowloon, Hong Kong, csblum@cityu.edu.hk
Affiliation: Computer Science Division, University of California at Berkeley
Address: Berkeley, CA 94720, blum@cs.berkeley.edu, http.cs.berkeley.edu/~blum

1. RESULT-CHECKING AND ITS APPLICATIONS

1.1 Assuring Software Reliability

Methodologies for assuring software reliability form an important part of the technology of programming. Yet the problem of efficiently identifying software bugs remains a difficult one, and one to which no perfect solution is likely to be found.

Software is generally debugged via **testing suites**: one runs a program on a variety of carefully selected inputs, identifying a bug whenever the program fails to perform correctly. This approach leaves two important questions incompletely answered.

First, how do we *know* whether or not the program's performance is correct? Generally, some sort of an **oracle** is used here: our program's output may be compared to the output of an older, slower program believed to be more reliable, or the programmers may subject the output to a painstaking (and likely subjective and incomplete) examination by eye.

Second, given that a test suite feeds a program only selected inputs out of the often enormous space of all possibilities, how do we assure that every bug in the code will be evidenced? Indeed, particular combinations of circumstances leading to a failure may well go untested. Furthermore, if the testing suite fails to accurately simulate the input distribution which the program will encounter in its lifetime, a supposedly debugged program may in fact fail quite frequently.

One alternative to testing is **formal verification**, a methodology in which mathematical claims about the behavior of a program are stated and proved. Thus it is possible to demonstrate once and for all that the program *must* behave correctly on all possible inputs. Unfortunately, constructing such proofs for even simple programs has proved unexpectedly difficult. Moreover, many programmers would likely find it unpleasant to have to formalize their expectations of program behavior into mathematical theorems.

Another alternative is **fault tolerance**. According to this methodology, the reliability of critical software may be enhanced by having several groups of programmers create separate versions. At run-time, all of the versions are executed, and their outputs are compared. The gross inefficiency of this approach is evident, in terms of programming manpower as well as either increased run-time or additional parallel-hardware requirements. Moreover, the method fails if common misconceptions among the several development groups result in corresponding errors [Butler and Finelli 1993].

We find more convincing inspiration in the field of communications, where **error-detecting** and **error-correcting codes** allow for the identification of arbitrary run-time transmission errors. Such codes are pragmatic and are backed by mathematical guarantees of reliability.

We wish to provide such run-time error-identification in a more general computational context. This motivates us to review the field of **result-checking**.

1.2 Simple Checkers

It is a matter of theoretical curiosity that, for certain computations, the time required to carry out the computation is asymptotically greater than the time required, given a tentative answer, to determine whether or not the answer is correct.

As an easy example, consider the following task: given as input a composite integer c, output any non-trivial factor d of c. Carrying out this computation is currently believed to be difficult, and yet, given **I/O pair** $\langle c, d\rangle$, it takes just one division to determine whether or not d is a correct output on input c.

These ideas have been formalized into the concept of a **simple checker** [Blum 1988]. Let f be a function with smallest possible computation time $T(n)$, where n is input length (or, if a strong lower bound cannot be determined, we informally set $T(n)$ equal to the smallest *known* computation time for f). Then a **simple checker for** $\boldsymbol{f}$ is an algorithm (generally randomized) with I/O specifications as follows:

—**Input:** I/O pair $\langle x, y\rangle$.
—**Correct output:** If $y = f(x)$, ACCEPT; otherwise, REJECT.
—**Reliability:** For all $\langle x, y\rangle$: on input $\langle x, y\rangle$ the checker must return correct output with probability (over internal randomization) $\geq p_c$, for p_c a constant close to 1.
—**"Little-o rule":** The checker is limited to time $o(T(n))$.

The "little-o rule" is important in that it requires a simple checker to be efficient, and also in that it forces the checker to determine whether or not $y = f(x)$ by means other than simply recomputing $f(x)$; hence the checker must be in some manner *different* from the program which it checks. We will see in Section 1.5 that this gives rise to certain hopes that checkers are indeed "independent" of the programs they check, and so may more reliably identify program errors.[1]

For an example of simple checking, consider a sorting task: input $\vec{x} = (x_1, \ldots, x_n)$ is an array of integers; output $\vec{y} = (y_1, \ldots, y_n)$ should be $\vec{x}$ sorted in increasing order. Completing this task—at least via a general comparison-based sort—is known to require time $\Omega(n \log n)$. Thus, if we limit our checker to time $O(n)$, this will suffice to satisfy the little-o rule.

Given $\langle \vec{x}, \vec{y}\rangle$, to check correctness we must first verify that the elements of $\vec{y}$ are in increasing order. This may easily be done in time $O(n)$. But we must also check that $\vec{y}$ is a permutation of $\vec{x}$. It might be convenient here to modify our sorter's I/O specifications, requiring that each element of $\vec{y}$ have attached a pointer to its original position in $\vec{x}$. Any natural sorting program could easily be altered to maintain such pointers, and once they are available we can readily complete our check in time $O(n)$.

But what if $\vec{y}$ cannot be augmented with such pointers? Similarly, what if $\vec{x}$ and $\vec{y}$ are only available on-line from sequential storage, so that $O(1)$-time pointer-dereferencing is not possible? Then we may still employ a randomized method due to [Wegman and Carter 1981; Blum 1988], which requires only one pass through each of $\vec{x}$ and $\vec{y}$.

[1]Variants of the little-o rule have also been considered: for example, requiring checkers to use less space than the programs they check, or to be implementable with smaller-depth circuits. For example, a rather trivial result is that, if the successive configurations of an arbitrary Turing Machine computation are available, then the computation may be checked by a very shallow circuit. Unfortunately, it is only the mechanical operation of the Turing Machine which is being checked here, not the correctness of the Turing Machine's program. Hence we return to the traditional little-o rule, which has generally proved to be the best route to creative, non-trivial checkers.

We randomly select a deterministic hash-function h from a suitably defined set of possibilities,[2] and we compare $h(x_1) + \cdots + h(x_n)$ with $h(y_1) + \cdots + h(y_n)$. If $\vec{y}$ is indeed a permutation of $\vec{x}$, the two values must be equal; conversely, it is readily proven that, if $\vec{y}$ is not a permutation of $\vec{x}$, the two values will differ with probability $\geq \frac{1}{2}$. Thus, if we ACCEPT only $\langle \vec{x}, \vec{y} \rangle$ pairs which pass t such trials, then our checker has probability of error $\leq \left(\frac{1}{2}\right)^t$, which may be made arbitrarily small.

A final note: our definition of simple checking is not new, but has perhaps not been clearly expressed in the checking literature. In particular, simple checking has been defined only implicitly, as the most efficient case of **complex checking** (Section 1.3). Here our particular interest in very fast checking has led us to bring out this distinction.

1.3 Self-Correctors

Again let f be a function with smallest (known) computation time $T(n)$. Let $\mathcal{D}$ be a well-defined probability distribution on inputs to f, and let $\mathbf{P}$ be a program such that, if x is chosen $\mathcal{D}$-randomly, $\mathbf{P}(x)$ equals $f(x)$ with probability of error limited to a small value p. In the current paper, $\mathcal{D}$ will always be the uniform-random distribution; so we are requiring simply that $\mathbf{P}$ computes f correctly on most inputs.

That $\mathbf{P}$ indeed has this property may be determined (with high probability) by testing $\mathbf{P}$ on $\gg \frac{1}{p}$ $\mathcal{D}$-random inputs and using some sort of an oracle—or a simple checker—to determine the correctness of each output. We envision this testing stage as being completed prior to run-time. Another possibility is the use of a **self-tester** [Blum et al. 1993], which can give such assurances efficiently, at run-time, and with little reliance on any outside oracle.

A **self-corrector for f** [Blum et al. 1993; Lipton 1991] is then an algorithm (generally randomized) with I/O specifications as follows:

—**Input:** x (an input to f), along with $\mathbf{P}$, a program known to compute f with probability of error on $\mathcal{D}$-random inputs limited to a small value p. The corrector is allowed to call $\mathbf{P}$ repeatedly, using it as a subroutine.

—**Correct output:** $f(x)$.

—**Reliability:** For all $\langle x, \mathbf{P} \rangle$: on input $\langle x, \mathbf{P} \rangle$ the corrector must return correct output with probability (over internal randomization) $\geq p_c$, for p_c a constant close to 1.

—**Time-bound:** The corrector's time-bound, including subroutine calls to $\mathbf{P}$, must be limited to a constant multiple of $\mathbf{P}$'s time-bound; the corrector's time-bound, counting each subroutine call to $\mathbf{P}$ as just one step, must be $o(T(n))$.

As an example [Blum et al. 1993], consider a task of multiplication over a finite field: input is $w, x \in F$; output is product $wx \in F$. We assume addition over F to be quicker and more reliable than multiplication.

Assume we know from testing that program $\mathbf{P}$ computes multiplication correctly for all but a $\leq \frac{1}{100}$ fraction of possible $\langle w, x \rangle$ inputs. (Note that this knowledge is

[2] A rather unusual set of hash functions is required; moreover, the time required to calculate h, and hence to complete a check, is debatable. Refer to [Blum 1988] for details.

in itself only a very weak assurance of the reliability of $\mathbf{P}$, as that $\leq \frac{1}{100}$ fraction of "difficult inputs" might well appear far more than $\frac{1}{100}$ of the time in the life of the program.) Then a self-corrector may be specified as follows:

ALGORITHM 1. *On input* $\langle w, x\rangle$,

—*Generate uniform-random* $r_1, r_2 \in F$.

—*Call* $\mathbf{P}$ *four times to calculate* $\mathbf{P}(w - r_1, x - r_2)$, $\mathbf{P}(w - r_1, r_2)$, $\mathbf{P}(r_1, x - r_2)$, *and* $\mathbf{P}(r_1, r_2)$.

—*Return, as our corrected value for* wx,

$$y_c := \mathbf{P}(w - r_1, x - r_2) + \mathbf{P}(w - r_1, r_2) + \mathbf{P}(r_1, x - r_2) + \mathbf{P}(r_1, r_2).$$

Why does this work? Note that each of the four calls to $\mathbf{P}$ is on a pair of inputs uniform-randomly distributed over F^2, and so will return the correct answer with probability of error at most $\frac{1}{100}$. Thus, all four return values are likely to be correct: the probability of error is at most $\frac{4}{100}$. And if all the return values are correct, then

$$\begin{aligned} y_c &= \mathbf{P}(w - r_1, x - r_2) + \mathbf{P}(w - r_1, r_2) + \mathbf{P}(r_1, x - r_2) + \mathbf{P}(r_1, r_2) \\ &= (w - r_1)(x - r_2) + (w - r_1)r_2 + r_1(x - r_2) + r_1 r_2 \\ &= [(w - r_1) + r_1][(x - r_2) + r_2] \\ &= wx. \end{aligned}$$

Note that corrected output y_c is superior to unmodified output $\mathbf{P}(w, x)$ in that there are no "difficult inputs": for *any* $\langle w, x\rangle$, each time we compute a corrected value y_c for wx, the chance of error (over the choice of r_1, r_2) is $\leq \frac{4}{100}$. Moreover observe that, if we compute several corrected outputs $y_c^{(1)}, \ldots, y_c^{(t)}$ (using new random values of r_1, r_2 for each $y_c^{(i)}$) and pick the majority answer as our final output, we thereby make the chance of error arbitrarily small. The price we pay for this enhanced reliability is a multiplication of $\mathbf{P}$'s usual run-time by a factor of $4t$. This performance loss will be further considered below.

A final note on self-correcting. Above we speak of needing an assurance that $\mathbf{P}$ is correct on all but a small fraction of inputs. Such language implies an assumption that $\mathbf{P}$'s behavior is fixed (except for possible randomization) prior to the self-correcting process: i.e., that $\mathbf{P}$ is not an adversary which can adapt in response to the corrector. This assumption is necessary, as an adaptable adversary could easily fool a corrector such as the above.[3] A checker, in contrast, is required to be secure against even an adaptable adversary.

One may also define a **complex checker** [Blum 1988], which outputs an ACCEPT/REJECT correctness-check similar to that of a simple checker, but which, like a self-corrector, is allowed to poll $\mathbf{P}$ at several locations (and so, like a self-corrector, tends to be less efficient than a simple checker). In general, a complex checker seeks to determine the correctness of $\mathbf{P}(x)$ by comparing $\mathbf{P}(x)$ to several of $\mathbf{P}$'s other outputs. More formally, a **complex checker for** $\boldsymbol{f}$ is defined as follows:

[3][Gemmell et al. 1991] however demonstrates that self-testing/correcting may be possible even when a program has limited adversarial power.

—**Input:** x (an input to f), along with **P**, a program which may compute f on all, some, or no inputs. The checker is allowed to call **P** repeatedly, using it as a subroutine.

—**Correct output:** If **P** is correct both on x and on all the other outputs sampled by the checker, must ACCEPT. If $\mathbf{P}(x)$ is incorrect, must REJECT.

—**Reliability:** For all $\langle x, \mathbf{P}\rangle$: on input $\langle x, \mathbf{P}\rangle$ the checker must return correct output with probability (over internal randomization) $\geq p_c$, for p_c a constant close to 1.

—**Time-bound:** The checker's time-bound, including subroutine calls to **P**, must be limited to a constant multiple of **P**'s time-bound; the checker's time-bound, counting each subroutine call to **P** as just one step, must be $o(T(n))$ (where $T(n)$ is again the smallest possible time to compute f).

Observe a necessary weakness of this definition: in the case that $\mathbf{P}(x)$ is correct but at least one of **P**'s other outputs sampled by the checker is incorrect, the checker may either ACCEPT or REJECT. This weakness is necessary (if we are to allow algorithms which go beyond simple checking) because, in the case that **P** returns only junk answers, we cannot expect the checker to figure out whether or not $\mathbf{P}(x)$ is correct. In spite of this weakness, the complex checker tells us what we need to know: if the checker ACCEPTS, we know that $\mathbf{P}(x)$ is correct; if the checker REJECTS, we know that **P** failed on at least one of a small, known set of inputs.

An example complex checker (for graph isomorphism) is described in Section 1.7. For more information on result-checking, refer to the annotated bibliography.

1.4 Debugging via Checkers

We will argue here that embedding checkers in software products may be a substantial aid in debugging those products and assuring their reliability. It is our hope that result-checking may form the basis for a debugging methodology more rigorous than the testing suite and more pragmatic than verification.

Throughout the process of software testing, embedded checkers may be used to identify incorrect output. They will thereby serve as an efficient alternative to a conventional testing oracle. Furthermore, even after software is put into use, leaving the checkers in the code will allow for the effective detection of lingering bugs. We envision that, each time a checker finds a bug, an informative output will be written to a file; this file will then be periodically reviewed by software-maintenance engineers. Another possibility is an immediate warning message, so that the user of a critical system will not be duped into accepting erroneous output.

While it could be argued that buggy output would be noticed by the user in any case, an automatic system for identifying errors should in fact reduce the probability that bugs will be ignored or forgotten. Moreover, a checker can catch an error in a program component whose effects may not be readily apparent to the user. Thus a checker might well identify a bug in a critical system before it goes on to cause a catastrophic failure. Moreover, checkers may trigger automatic self-correction: potentially a method for the creation of extremely reliable systems.

By writing a checker, one group of programmers may assure themselves that another group's code is not undermining theirs by passing along erroneous output. Similarly, the software engineer who creates the specifications for a program component can write a checker to verify that the programmers have truly understood and provided the functionality he required. Since checkers depend only on the I/O specifications of a computational task, one may check even a program component whose internals are not available for examination, such as a library of utilities running on a remote machine. Thus checkers facilitate the discovery of those misunderstandings at the "seams" of programs which so often underlie a software failure. Evidently, all of this applies well to object-oriented paradigms.

Unlike verification, checking reveals incorrect output originating in any cause—whether due to software bugs, hardware bugs, or transient run-time errors. Moreover, since a checker concerns itself only with the I/O specifications of a computational task, the introduction of a potentially unreliable new algorithm for solving an old task may not require substantial change to the associated checker. And so, as a program is modified throughout its lifetime—often without an adequate repeat of the testing process—its checkers will continue to guard against errors.

1.5 Checker-Based Debugging Concerns

• **Performance loss:** Simple checkers are inherently efficient: due to the little-o rule, a simple checker should not significantly slow the program **P** that it checks. Self-correctors, on the other hand, require multiple calls to **P**, and so multiply execution-time by a small constant (or, alternatively, require additional parallel hardware). In a real-time computer system, such a slowdown may not be acceptable.

Hence it is desirable to check a given program by implementing both a simple checker and a self-corrector. The simple checker may then be run on each output, while only in the (hopefully rare) case that the output is incorrect need the self-corrector be run as well; hence the self-corrector is kept off the common-case path. Alternatively, in Section 1.6 we will consider a change to our definitions which will allow for certain self-correcting algorithms to run at real-time speeds.

• **Real-number issues** (see [Gemmell et al. 1991; Ar et al. 1993]): Traditional result-checking algorithms, such as the example self-corrector from Section 1.3, often rely on the orderly properties of finite fields. In many programming tasks, we are more likely to encounter real numbers—i.e., approximate reals represented by a fixed number of bits.

Such numbers will be limited to a legal subdomain of $\Re$. In Section 2.4, we will see how to modify a traditional self-correcting methodology to account for this. Moreover, limited precision will likely result in round-off errors within **P**. We must then determine how much arithmetic error may allowably occur within **P**, and must check correctness only up to the limit of this error-delta. In Section 2, we will see that limited precision calculations, while a substantial challenge, are by no means fatal to our checking project.

• **Testing checkers:** A checker, like any software component, must be carefully debugged. Otherwise, a degenerate checker **C** may indiscriminately report program **P** to be correct, thereby giving a false sense of assurance.

The problem of constructing a suitable testing suite for a checker is non-trivial. Evidently testing a checker on correct program output is insufficient: we must also test the checker's ability to recognize bugs—and not just flagrant, random bugs, but various and subtle ones. One possibility would be to employ a mutation model of software error: i.e., to test **C** on the output from mutated variants of **P**.

• **Buggy checkers:** What if we nevertheless fail to debug our checker? Indeed, it may be objected that result-checking begs the question of software reliability: for checking may fail due to the checker's code being as buggy as the program's. To this objection we respond with three arguments: arguments which are, however, heuristic rather than rigorous.

First, checkers can often be simpler than the programs they check, and so, presumably, less likely to have bugs. For example, modern computers may use intricate, arithmetically unstable algorithms to compute matrix multiplication in time, say, $O(n^{2.38})$, rather than the standard $O(n^3)$. And yet Freivalds's $O(n^2)$-time checker for matrix multiplication [Freivalds 1979] uses only the simplest of multiplication algorithms, and so is seemingly likely to be bug-free.

Second, observe that one of the following four conditions must hold each time a (possibly buggy) simple checker **C** checks the output of a program **P**:

(I). $\mathbf{P}(x)$ is correct; **C** correctly accepts it.
(II). $\mathbf{P}(x)$ is incorrect; **C** correctly rejects it.
(III). "False alarm": $\mathbf{P}(x)$ is correct; **C** incorrectly rejects it.
(IV). "Missed bug": $\mathbf{P}(x)$ is incorrect; **C** incorrectly accepts it.

Only a "missed bug" is a truly bad outcome: for a "false alarm," while annoying, at least draws our attention to a bug in **C**.

To reduce the likelihood of missed bugs, it suffices to rule out a strong correlation between x values at which **P** fails and x values at which $\mathbf{C}(x, \mathbf{P}(x))$ fails. It is our heuristic contention that such a correlation is unlikely. Indeed, recall that one effect of the "little-o rule" is that **C** doesn't have sufficient time to merely reduplicate the computation performed by **P**. Thus we claim (heuristically) that **C** must be doing something *essentially different* from what **P** does, and so, if buggy, may reasonably be expected to make *different errors.* But then we would expect few correlated errors; moreover, we would expect more uncorrelated than correlated errors, so that a given bug could well be identified and fixed before it goes on to generate *any* correlated errors.[4]

Finally, say that correlated errors nonetheless occur. This might in particular be expected due to faults in hardware or software components on which both **P** and **C** depend. We argue that, even in this case, a "missed bug" may not result. Indeed, consider our checker for a sorting program (Section 1.2), which determines whether or not $\vec{y}$ is a permutation of $\vec{x}$ by comparing $h(x_1) + \cdots + h(x_n)$ with $h(y_1)+\cdots+h(y_n)$: assuming that the sorting program has failed, $h(x_1)+\cdots+h(x_n)$ and $h(y_1) + \cdots + h(y_n)$ will not match (with probability $\geq \frac{1}{2}$). If the checker

[4]It could be argued that the little-o rule only prohibits **C** from duplicating the most time-consuming of **P**'s components; **P**'s simpler components may be trivially duplicated in **C**, and so, if buggy, might produce correlated errors. To this argument we respond that **C**'s primary function is to check the central algorithm of **P**; smaller components needed as subroutines by both **P** and **C** should, if necessary, be checked separately.

As an illustration of a related issue, consider a standard MAX-FLOW program, which proceeds by successively adjusting flows until it finds a configuration which achieves optimal flow across a particular cut. A checker for this program need do little more than verify that optimal flow is indeed achieved across the cut—which is what the program itself does in its final stage. Hence checking this sort of program seems trivial and ineffective. Our response is simply that such a program indeed requires no *separate* checker, because it is naturally self-checking.

simultaneously fails, it may miscalculate $h(x_1) + \cdots + h(x_n)$ and/or $h(y_1) + \cdots + h(y_n)$; but, we argue, it seemingly is not likely to fail in exactly such a manner as to get values for the two sums which happen to match each other.

Many checkers proceed in this way, calculating two quantities and declaring an error if they do not match. Degenerate checker behavior seems unlikely to yield matching quantities; and so we argue that checkers, even when buggy themselves, have a natural resistance to the bad case of missing a program bug.

1.6 Stored Randomness

We here introduce an extension to traditional checker definitions: we suggest that randomized checkers, rather than having to generate fresh random bits for each check, should be allowed to use preprocessed **stored randomness**. That is, prior to run-time—while the program is idle, or during boot-up, or even during software development—we generate random bits and do some (perhaps lengthy) preprocessing; these random bits, together with preprocessed values dependent on them, form one or more **random packages.** These packages are stored; then, at run-time, each check requires such a package as an additional input.[5]

Stored randomness has several potential advantages. First, it allows for much or all of a checker's randomness to be generated before run-time. This could be useful particularly in doing very quick checks of low-level functionalities such as microprocessor arithmetic (as in [Blum and Wasserman 1996]): in this context, having to generate random bits at run-time could be an inconvenience. Second, we shall see that stored randomness allows for the use of checking algorithms which would otherwise require too much computation at run-time: by paying a time-cost of $\Omega(T(N))$ in preprocessing, we "cheat" some work out of the way, so that each run-time check can then be completed in time $o(T(N))$.

For a trivial example of stored randomness, we can reconsider the multiplication self-corrector of Section 1.3. If in a preprocessing stage r_1, r_2 are generated, $\mathbf{P}(r_1, r_2)$ is calculated, and package $R = \langle r_1, r_2, \mathbf{P}(r_1, r_2)\rangle$ is stored, this leaves only three calls to $\mathbf{P}$ at run-time, rather than four. However, this improvement is not particularly impressive. Stored randomness will be further explicated through more substantial examples: in Sections 2.1 and 2.5, we will present a simple checker which is possible only if one allows stored randomness, and a self-corrector which, if one allows stored randomness, can run at real-time speeds.

How much randomness must be stored? Let N be our fixed input length, and let ℓ be the number of checks we expect to carry out at run-time. One approach would be to preprocess ℓ packages. Then each check would have its own "fresh" randomness, which would certainly be good from the point of view of reliability. However, we might expect ℓ to be $\gg N$ or even unknown during preprocessing; so this trivial approach could have too high a cost in preprocessing time and storage space.

An opposite approach would be to generate a single random package and use it for every check. But would the repeated use of a "stale" random package reduce

[5]This method requires input length N to be fixed prior to run-time; equivalently, new preprocessing stages will be required as input length increases. This use of stored information is analogous to that in the P/poly model of computation.

the reliability of our checker? Sections 2.1 and 2.5 initially choose this approach; and Lemmas 4 and 10 argue that the consequent reduction in reliability is not fatal.

Moreover, in Sections 2.1 and 2.5 we go on to suggest an intermediate approach, which requires a number of stored packages that is $O(N)$ and independent of ℓ. Lemmas 5 and 11 argue that the resulting stored-random checkers are then highly reliable.

1.7 Variations on the Checking Paradigm

It may still be objected that most programming tasks are less amenable to checking than are the clean mathematical problems with which theoreticians usually deal. Nevertheless, it is our experience that many such tasks may be subjected to interesting checks. The following is a list of checking ideas which may suggest to the reader how definitions can be generalized to make checking more suitable as a tool for software debugging.

• **Partial checks:** One may find it sufficient to check only certain aspects of a computational output. Programmers may focus on properties they think particularly likely to reveal bugs—or particularly critical to the success of the system. Certain checkers might only be capable of identifying outputs incorrect by a very large error-delta. Even just checking inputs and outputs separately to verify that they fall in a legal domain may prove useful.

• **Timing checks:** One might wish to monitor the execution-time of a software component; an unexpectedly large (or small) time could then reveal a bug. For instance, a programmer's expectation that a particular subroutine should take time $\approx 10n^2$ on input of variable length n could be checked against actual performance.

• **Checking via interactive proofs:** There exist **interactive proofs** of correctness for certain mathematical computations; and many such interactive proofs may equivalently be regarded as complex checkers. For example, it follows from the IP method of [Shamir 1992] that, if program **P** claims to solve a PSPACE-complete problem, then there is a checker for **P** which requires polynomial time plus a polynomial number of calls to **P**.

Similarly, the following interactive-proof method, due to [Goldreich et al. 1991], may equivalently be regarded as a complex checker. Say that **P** claims to decide graph isomorphism. If **P** says that $A \cong B$, we can check this by running **P** on successively reduced versions of A and B, thereby forcing **P** to reveal a particular isomorphism. Less trivially, if **P** claims that $A \ncong B$, we repeat several times the following check: generate C, a random isomorphism either of A (with probability $\frac{1}{2}$) or of B (with probability $\frac{1}{2}$); then ask **P** to tell us which one of A or B is isomorphic to C. If $A \cong B$, **P** can only guess which of the two we permuted.

For more on interactive proof, refer to the bibliography.

• **Changing I/O specifications:** In Section 1.2, we saw how an easy augmentation to the output of a sorting program could make the program easier to check. Similarly, consider the problem of checking a $\Theta(\log n)$-time binary-search task. As traditionally stated (see [Bentley 1986, p. 35], where it features in a relevant discussion of the difficulty of writing correct programs), binary search has as input a key k and a numerical array $a[1], \ldots, a[n]$, where $a[1] \leq \cdots \leq a[n]$, and as output i such that $a[i] = k$, or 0 if k is not in the array.

Note that the problem as stated is uncheckable: for, if the output is 0, it will take $\Theta(\log n)$ steps to confirm that k is indeed not in the array. But say that we modify the output specification for our binary-search task to read: output i such that $a[i] = k$, or, if k is not in the array, $\langle j, j+1 \rangle$ such that $a[j] < k < a[j+1]$. Any natural binary-search program can easily be modified to give such output, and completing an $O(1)$-time check is then straightforward.

• **Weak or occasional checks** [Blum, A. 1994]: Certain pseudo-checkers have only a small probability of noticing a bug. For example, if a search program, given key k and array $a[1], \ldots, a[n]$, claims that k does not occur in a, we could check this by selecting an element of a at random and verifying that it is not equal to k. This weak checker has probability $\frac{1}{n}$ of identifying an incorrect claim.

Alternatively, to save time a lengthy check might be employed only on occasional I/O pairs. Given that many bugs cause frequent errors over the lifetime of a program, such weak or occasional checks may well be useful.

• **Batch checks** [Rubinfeld 1992]: For certain computational tasks, if one stores up a number of I/O pairs, one may check them all at once more quickly than one could have checked them separately. While it may be too late to correct output after completing such a retroactive check, this method can when applicable provide a very time-efficient identification of bugs.

• **Inefficient auto-correcting:** When a checker identifies a program error, it could correct by, for example, loading and running an older, slower, but hopefully more reliable version of the program. This should be necessary only on rare occasions, so the overall loss of speed should not be unreasonable. For example, a trivial $O(n^3)$-time matrix-multiplication program can be kept in reserve in case a more sophisticated $O(n^{2.38})$-time program fails.

2. CASE STUDY: CHECKING REAL-NUMBER LINEAR TRANSFORMATIONS

We will now consider the problem of checking and correcting a general linear transformation, defined as follows: input to program $\mathbf{P}$ is n-length vector $\vec{x}$ of real numbers; correct output is n-length vector $\vec{y} = \vec{x}A$, where A is a fixed $n \times n$ matrix of real coefficients. Extensions of our results to more general cases—e.g., $\vec{y}$ of different length than $\vec{x}$, or components complex rather than real—are straightforward. However, we do require n and A to be fixed prior to run-time.

Evidently, any such transformation may be computed in $O(n^2)$ arithmetic operations. We assume that, for non-trivial transformations A, the smallest possible time to compute the transformation may range from $\Theta(n)$ to $\Theta(n^2)$. Our simple checker will take time $O(n)$ (as will our self-corrector, not counting one or more calls to $\mathbf{P}$), so the little-o rule is satisfied for all but the $\Theta(n)$-time transformations.[6] For example, the Fourier Transform is linear and may be computed in time $\Theta(n \log n)$ via the Fast Fourier Transform; if one makes the reasonable assumption that an $O(n)$-time Fourier Transform is not possible, then our algorithms qualify as checkers for this transform.

Assume now that program $\mathbf{P}$ is intended to carry out the $\vec{x} \mapsto \vec{x}A$ transformation

[6]In the more general case that $\vec{y}$ has length m, any non-trivial linear transformation will have smallest possible computation-time on range $\Theta(\max\{m,n\})$ to $\Theta(mn)$. It is easily verified that our simple checker and our self-corrector each take time $O(\max\{m,n\})$; and so, once again, the little-o rule is satisfied for all but the fastest of transformations.

on a computer with a limited-accuracy, fixed-point representation of real numbers. **P**'s I/O specifications might then be formulated as follows:

—**Input:** $\vec{x} = (x_1, \ldots, x_n)$, where each x_i is a fixed-point real number and so is limited to a legal subdomain of $\Re$: say, to domain $[-1, 1]$.

—**Output:** $\mathbf{P}(\vec{x}) = \vec{y} = (y_1, \ldots, y_n)$, where each y_i is a fixed-point real.

—Let $\delta \in \Re^+$ be a constant; let $\vec{\Delta} = (\Delta_1, \ldots, \Delta_n)$ be the "error vector" $\vec{y} - \vec{x}A$. Then:
  —I/O pair $\langle \vec{x}, \vec{y} \rangle$ is **definitely correct** iff, for all i, $|\Delta_i| \leq \delta$.
  —I/O pair $\langle \vec{x}, \vec{y} \rangle$ is **definitely incorrect** iff there exists i such that $|\Delta_i| \geq 6\sqrt{n}\,\delta$.

Note that, due to arithmetic round-off errors within **P**, a small error-delta is to be regarded as acceptable: specifically, we model **P**'s expected error as a flat error of at most $\pm\delta$ in each component of $\vec{y}$. Also note that we will only require the simple checker of Section 2.1 to accept I/O which is *definitely correct* and to reject I/O which is *definitely incorrect.* It is seemingly unavoidable that there is an intermediate region of I/O pairs $\langle \vec{x}, \vec{y} \rangle$ which could allowably be either accepted or rejected. However, this intermediate region currently appears to be altogether too large (i.e., condition "exists i such that $|\Delta_i| \geq 6\sqrt{n}\,\delta$" is too strong). It seems that we should be able to do better; and, in Section 2.2, we shall suggest an improvement.

2.1 A Simple Checker

Motivation: To develop a simple checker for **P**, we will take the approach of generating a randomized vector $\vec{r}$ and trying to determine whether or not $\vec{y} \approx \vec{x}A$ by calculating whether or not $\vec{y} \cdot \vec{r} \approx (\vec{x}A) \cdot \vec{r}$. This method is a variant of that in [Ar et al. 1993, Section 4.1.1].

To facilitate the calculation of $(\vec{x}A) \cdot \vec{r}$, we will employ the method of *stored randomness* (Section 1.6): by generating $\vec{r}$ and preprocessing it together with A prior to run-time, we will then be able to complete the calculation of $(\vec{x}A) \cdot \vec{r}$ with just $O(n)$ run-time arithmetic operations. Thus we will achieve the strong result of checking a $\Theta(n^2)$-time computation in time $O(n)$.

ALGORITHM 2. *Our checker's preprocessing stage, to be completed prior to run-time, is specified as follows:*

—*For* $k := 1$ *to 10:*
  —*Generate and store* $\vec{r}^{(k)}$*, an* n*-length vector of form* $(\pm 1, \pm 1, \ldots, \pm 1)$*, where each* $\pm$ *is chosen positive or negative with independent 50/50 probability.*
  —*Calculate and store* $\vec{v}^{(k)} := \vec{r}^{(k)} A^T$ *(where* A^T *denotes the transpose of* A*).*[7]

Then, to check an I/O pair $\langle \vec{x}, \vec{y} \rangle$ *at run-time, we employ this* $O(n)$*-time check:*

—*For* $k := 1$ *to 10:*
  —*Calculate* $D^{(k)} := \vec{y} \cdot \vec{r}^{(k)} - \vec{x} \cdot \vec{v}^{(k)}$.
  —*If* $|D^{(k)}| \geq 6\sqrt{n}\,\delta$, REJECT.
—*If all 10 tests are passed,* ACCEPT.

[7] Since this calculation need be done only a few times, and when time is not at a premium, its correctness can be assured, e.g., by employing simple brute-force methods, or by checking the result by hand.

Why does this work? First note that

$$\begin{aligned} D^{(k)} &= \vec{y} \cdot \vec{r}^{\,(k)} - \vec{x} \cdot \vec{v}^{\,(k)} \\ &= \vec{y} \cdot \vec{r}^{\,(k)} - \vec{x} \cdot (\vec{r}^{\,(k)} A^T) \\ &= \vec{y} \cdot \vec{r}^{\,(k)} - (\vec{x} A) \cdot \vec{r}^{\,(k)} \\ &= (\vec{y} - \vec{x} A) \cdot \vec{r}^{\,(k)} \\ &= \vec{\Delta} \cdot \vec{r}^{\,(k)} \\ &= \pm \Delta_1 \pm \cdots \pm \Delta_n, \end{aligned}$$

where each $\pm$ is positive or negative with independent 50/50 probability.[8]

LEMMA 1. *For any definitely correct* $\langle \vec{x}, \vec{y}\, \rangle$,

$$\Pr_{\vec{r}^{\,(k)}} \left[|D^{(k)}| \geq 6\sqrt{n}\,\delta \right] \leq \frac{1}{10,000,000}.$$

PROOF. Each $|\Delta_i| \leq \delta$. Thus, by a Chernoff Bound [Alon et al. 1992, Theorem A.16],

$$\begin{aligned} \Pr\left[| \pm \Delta_1 \pm \cdots \pm \Delta_n | \geq 6\sqrt{n}\,\delta \right] &\leq 2e^{-(6\sqrt{n})^2/2n} \\ &< \frac{1}{10,000,000}. \quad \square \end{aligned}$$

LEMMA 2. *For any definitely incorrect* $\langle \vec{x}, \vec{y}\, \rangle$,

$$\Pr_{\vec{r}^{\,(k)}} \left[|D^{(k)}| < 6\sqrt{n}\,\delta \right] \leq \frac{1}{2}.$$

PROOF. We know there exists i such that $|\Delta_i| \geq 6\sqrt{n}\,\delta$. Fix all components of $\vec{r}^{\,(k)}$ except for $r_i^{(k)}$. Then observe that, for at least one of the two possible values $\{\pm 1\}$ of $r_i^{(k)}$, $|D^{(k)}| = |\Delta_1 r_1^{(k)} + \cdots + \Delta_n r_n^{(k)}|$ must be $\geq |\Delta_i| \geq 6\sqrt{n}\,\delta$. Thus $|D^{(k)}| < 6\sqrt{n}\,\delta$ for at most $\frac{1}{2}$ of the equally likely choices of random vector $\vec{r}^{\,(k)}$. $\square$

LEMMA 3. **(I).** *For any definitely correct* $\langle \vec{x}, \vec{y}\, \rangle$,

$$\Pr_{\vec{r}^{\,(1)},\ldots,\vec{r}^{\,(10)}} [\textit{checker mistakenly rejects}\ ] \leq \frac{1}{1,000,000}.$$

(II). *For any definitely incorrect* $\langle \vec{x}, \vec{y}\, \rangle$,

$$\Pr_{\vec{r}^{\,(1)},\ldots,\vec{r}^{\,(10)}} [\textit{checker mistakenly accepts}\ ] \leq 2^{-10} < \frac{1}{1,000}.$$

PROOF. **(I).** By Lemma 1, the probability of a mistaken reject for each $\vec{r}^{\,(k)}$ is $\leq \frac{1}{10,000,000}$. Thus, the probability of a mistaken reject on *any* of the 10 tries is $\leq 10 \cdot \frac{1}{10,000,000}$.

(II). By Lemma 2, the probability of failing to reject at each $\vec{r}^{\,(k)}$ is $\leq \frac{1}{2}$. Thus, the probability of rejecting on *none* of the 10 independent tries is $\leq 2^{-10}$. $\square$

[8] We have employed a linear-algebra identity: the dot-product of two vectors is equivalently their matrix-product once the second vector (i.e., row-matrix) has been transposed; thus $\vec{x} \cdot \left(\vec{r}^{\,(k)} A^T\right) = \vec{x}\left(\vec{r}^{\,(k)} A^T\right)^T = \vec{x} A \left(\vec{r}^{\,(k)}\right)^T = (\vec{x} A) \cdot \vec{r}^{\,(k)}$.

While Lemma 3 embodies our intuition of the checker's correctness, it does not deal with the implications of using the same stored "random package" $R = \langle \vec{r}^{(1)}, \ldots, \vec{r}^{(10)}, \vec{v}^{(1)}, \ldots, \vec{v}^{(10)} \rangle$ in each run-time check. The following claims suggest that this use of *stored randomness* does not seriously undermine the reliability of the checker:

LEMMA 4. *Assume that we preprocess random package R and use this package to check ℓ runs of $\mathbf{P}$. Then, for probabilities over the choice of R (where all results except (II) are independent of the value of ℓ):*

(I). *Assume that a bug causes at least one definitely incorrect I/O pair. Then the probability that our checker will miss this bug is $\leq \frac{1}{1,000}$.*

(II). *If $\ell \leq 10,000$, then the probability that* any *definitely correct I/O pair will be mistakenly rejected is $\leq \frac{1}{100}$.*

(III). *The probability that $\geq \frac{1}{2}$ of all definitely incorrect I/O pairs will be mistakenly accepted is $\leq \frac{1}{500}$.*

(IV). *The probability that $\geq \frac{1}{10,000}$ of all definitely correct I/O pairs will be mistakenly rejected is $\leq \frac{1}{100}$.*

PROOF. (I) and (II) follow readily from Lemma 3. To prove (III), note a consequence of Lemma 3 (II): if we select R randomly and $\langle \vec{x}, \vec{y} \rangle$ uniform-randomly from the list of all *definitely incorrect* I/O pairs out of our ℓ runs, then the probability that our checker accepts given input $\langle \vec{x}, \vec{y} \rangle$ and randomization R is $\leq \frac{1}{1,000}$. But if (III) were false, then this same probability would, on the contrary, be $> \frac{1}{500} \cdot \frac{1}{2} = \frac{1}{1,000}$. (IV) is proved analogously. $\square$

While Lemma 4 gives a reasonably strong assurance, it is not entirely satisfactory. Indeed, when stored randomness is used as described above, there is no run-time randomization, so that if, for example, the checker fails on an input $\langle \vec{x}, \vec{y} \rangle$ which occurs repeatedly in the ℓ runs, then it will fail on each repetition. This has several potentially bad consequences. First, if $\mathbf{P}$ is regarded as an adversary, it can defeat the checker by learning its "bad inputs." Second, observe that if fresh randomness could be used for each check, then each check would have *independent* probability of error $\leq \frac{1}{1,000}$; hence the probability of *many* checker errors during ℓ runs of $\mathbf{P}$ would be exponentially small. With stored randomness, the probability of many errors is small (per Lemma 4 (III,IV)) but not *exponentially* small. Note that frequent errors could result in system failure, if checker errors trigger lengthy self-correcting procedures and if we are dealing with a **soft real-time** system in which *average* run-time must be kept low.

Hence we now introduce a more sophisticated approach to stored randomness, one which allows stored-random checkers to be as reliable as checkers using fresh random bits.

Recall that $R = \langle \vec{r}^{(1)}, \ldots, \vec{r}^{(10)}, \vec{v}^{(1)}, \ldots, \vec{v}^{(10)} \rangle$ is the preprocessed "random package" needed by our checker, and that $\langle \vec{x}, \vec{y} \rangle$ is an I/O pair to be checked. Also recall that we are assuming input length to be fixed before run-time: specifically, let us assume that the value of $\langle \vec{x}, \vec{y} \rangle$ can be completely represented by N bits, so that there are at most 2^N possible values for $\langle \vec{x}, \vec{y} \rangle$.

ALGORITHM 3. *At preprocessing time, we create, not just a single random package R, but $4N$ random packages $R_1^*, \ldots, R_{4N}^*$. Then, at run-time, we pick $R_i^* \in_u$*

$\{R_1^*, \ldots, R_{4N}^*\}$, and use R_i^* to check $\langle \vec{x}, \vec{y} \rangle$. For a smaller probability of error, we can repeat this check with several more $R_i^* \in_u \{R_1^*, \ldots, R_{4N}^*\}$ and return the majority answer.

The following lemma argues that our improved stored-random checker has an *independent* probability of error at each run-time check which can be made arbitrarily small. Also observe that **P** cannot fool the checker even if **P** is an adversary which knows the values of $R_1^*, \ldots, R_{4N}^*$.

LEMMA 5. *With probability of failure $\leq 2^{-5N}$, $R_1^*, \ldots, R_{4N}^*$ will be such that, for* all *$\langle \vec{x}, \vec{y} \rangle$, at most $\frac{1}{4}$ of $R_1^*, \ldots, R_{4N}^*$ are bad for checking $\langle \vec{x}, \vec{y} \rangle$. (Hence, checking with t values of $R_i^* \in_u \{R_1^*, \ldots, R_{4N}^*\}$ and taking the majority answer results in a run-time error-probability which is exponentially small in t.)*

PROOF. Analogous to the standard proof that BPP $\subseteq$ P/poly:

Recall from Lemma 3 that, for any given $\langle \vec{x}, \vec{y} \rangle$, the probability (over the random choice of a package R) that the checker fails given input $\langle \vec{x}, \vec{y} \rangle$ and randomization R is $\leq 2^{-10}$.

Hence, for any given $\langle \vec{x}, \vec{y} \rangle$, the probability (over the random choice of packages $R_1, \ldots, R_N$) that *all* of $R_1, \ldots, R_N$ are bad for checking $\langle \vec{x}, \vec{y} \rangle$ is $\leq 2^{-10N}$.

Hence the probability (over the random choice of $R_1^*, \ldots, R_{4N}^*$) that there exists *any* $\langle \vec{x}, \vec{y} \rangle$ and *any* size-N subset $R_1, \ldots, R_N$ of $R_1^*, \ldots, R_{4N}^*$ such that all of $R_1, \ldots, R_N$ are bad for checking $\langle \vec{x}, \vec{y} \rangle$ is $\leq 2^N \cdot \binom{4N}{N} \cdot 2^{-10N} \leq 2^N \cdot 2^{4N} \cdot 2^{-10N} = 2^{-5N}$. $\square$

This is in fact a general method allowing any simple checker to make use of stored randomness without compromising its reliability. Only $O(\log N)$ bits of run-time randomness are then needed for each check.

2.2 A Stronger Simple Checker

Recall that $\vec{\Delta} = (\Delta_1, \ldots, \Delta_n)$ is the "error vector" $\vec{y} - \vec{x}A$. Also recall that, while the full value of vector $\vec{\Delta}$ is not calculable in time $O(n)$, we *can* use stored, preprocessed $\vec{r}^{(k)}$ and $\vec{v}^{(k)}$ to efficiently calculate randomized scalar value $\vec{r}^{(k)} \cdot \vec{\Delta}$.

The assurance that our simple checker will reject instances for which any $|\Delta_i| \geq 6\sqrt{n}\,\delta$ is not entirely satisfactory: it leaves too large a gap between instances which are reliably accepted and those which are reliably rejected. A stronger method would be to approximate

$$|\vec{\Delta}| = \sqrt{\Delta_1^2 + \cdots + \Delta_n^2}$$

and accept or reject by comparing this value to a suitable threshold. This method is particularly appropriate in that engineers often use this **root-mean-square measure** to represent overall error in calculations such as Fourier Transforms.

What we need then is an efficient method for approximating $|\vec{\Delta}|$. The lemmas below prove that it is indeed possible to efficiently approximate $|\vec{\Delta}|^2$.

LEMMA 6. *Let $\vec{r}^{(1)}, \ldots, \vec{r}^{(m)}$ be n-length vectors, each component ± 1 with independent 50/50 probability, and consider randomized quantity*

$$X = \left(\frac{1}{m}\right) \sum_{k=1}^{m} \left[\vec{r}^{(k)} \cdot \vec{\Delta}\right]^2 .$$

For any positive ϵ and p: if $m \geq \frac{2}{p\epsilon^2}$, then, with probability greater than $1-p$,

$$(1-\epsilon)|\vec{\Delta}|^2 \leq X \leq (1+\epsilon)|\vec{\Delta}|^2.$$

PROOF. Observe that

$$\begin{aligned}
\left[\vec{r}^{(k)} \cdot \vec{\Delta}\right]^2 &= \left[\sum_{i=1}^{n} r_i^{(k)} \Delta_i\right]^2 \\
&= \sum_{i=1}^{n} \Delta_i^2 + \sum_{i=1}^{n} \sum_{j \neq i} r_i^{(k)} r_j^{(k)} \Delta_i \Delta_j \\
&= |\vec{\Delta}|^2 + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} r_i^{(k)} r_j^{(k)} \Delta_i \Delta_j. \\
\Longrightarrow \quad X &= \left(\frac{1}{m}\right) \sum_{k=1}^{m} |\vec{\Delta}|^2 + \left(\frac{2}{m}\right) \sum_{k=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} r_i^{(k)} r_j^{(k)} \Delta_i \Delta_j \\
&= |\vec{\Delta}|^2 + \left(\frac{2}{m}\right) R,
\end{aligned}$$

where $\left(\frac{2}{m}\right)$ times $R := \sum_{k=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} r_i^{(k)} r_j^{(k)} \Delta_i \Delta_j$ is the randomized "error term" whose size we wish to bound.

Note that R is a sum of $\frac{mn(n-1)}{2}$ *pairwise independent* quantities. Thus:

$$\begin{aligned}
\operatorname{Var}[R] &= \sum_{k=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \operatorname{Var}\left[r_i^{(k)} r_j^{(k)} \Delta_i \Delta_j\right] \\
&= \sum_{k=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Delta_i^2 \Delta_j^2 \\
&\leq \sum_{k=1}^{m} \left(\frac{1}{2}\right) \sum_{i=1}^{n} \sum_{j=1}^{n} \Delta_i^2 \Delta_j^2 \\
&= \left(\frac{1}{2}\right) \sum_{k=1}^{m} \left[\left(\sum_{i=1}^{n} \Delta_i^2\right) \cdot \left(\sum_{j=1}^{n} \Delta_j^2\right)\right] \\
&= \left(\frac{1}{2}\right) \sum_{k=1}^{m} \left[|\vec{\Delta}|^2 \cdot |\vec{\Delta}|^2\right] \\
&= \frac{m|\vec{\Delta}|^4}{2}.
\end{aligned}$$

Thus the variance of "error term" $\left(\frac{2}{m}\right)R$ is $\leq \left(\frac{2}{m}\right)^2 \cdot \frac{m|\vec{\Delta}|^4}{2} = \frac{2|\vec{\Delta}|^4}{m}$. Since we are assuming $m \geq \frac{2}{p\epsilon^2}$, this is $\leq p\epsilon^2|\vec{\Delta}|^4$.

But then the probability that $\left|\left(\frac{2}{m}\right)R\right| > \epsilon|\vec{\Delta}|^2$ must be less than p. This follows by Chebyshev's Inequality: i.e., if the contrary were true, then we would have $\operatorname{Var}\left[\left(\frac{2}{m}\right)R\right] > p\epsilon^2|\vec{\Delta}|^4$, which contradicts what we have proved above. $\square$

LEMMA 7. *Recall once again that, using preprocessed $\langle \vec{r}^{(k)}, \vec{v}^{(k)} \rangle$, we can calculate $\vec{r}^{(k)} \cdot \vec{\Delta}$ in time $O(n)$. Then: for any positive ϵ and p, we can, in time $O(n(\frac{1}{\epsilon})^2 \log(\frac{1}{p}))$, calculate a value Y such that, with probability greater than $1-p$,*

$$(1-\epsilon)|\vec{\Delta}|^2 \leq Y \leq (1+\epsilon)|\vec{\Delta}|^2.$$

PROOF. Let $t := \lceil 6 \ln(\frac{1}{p}) \rceil$ and $m := \lceil \frac{8}{\epsilon^2} \rceil$. We repeat $2t$ times the calculation of a randomized X-value as described in Lemma 6, using preprocessed $\langle \vec{r}^{(1)}, \ldots, \vec{r}^{(2tm)}, \vec{v}^{(1)}, \ldots, \vec{v}^{(2tm)} \rangle$. We then return the median of X-values $X_1, \ldots,$ X_{2t} as our value for Y.

Why does this work? By Lemma 6, each X_i approximates $|\vec{\Delta}|^2$ to within the desired range with independent probability of error less than $\frac{1}{4}$. And the median of the X-values can be out of range only if at least t of the $2t$ X-values are out of range. By a Chernoff Bound, the probability of this is $< e^{-t/6} \leq p$, as desired. □

2.3 Simple Checker—Variants

• A fuller statement of our algorithm would include consideration of arithmetic round-off error in the checker itself as well as in **P**. However, under the reasonable assumption (for a fixed-point system) that addition and negation do not generate round-off errors, our checker could generate error only in the n real-number multiplications needed to calculate $\vec{x} \cdot \vec{v}^{(k)}$. It is our heuristic expectation that such error would not be large enough to significantly affect the validity of our checker.

• Evidently, different values could be used for the constants in our algorithm of Section 2.1. Our goal was to particularize a pragmatic checker—one which satisfies reasonably well each of the following desirable conditions: small gap between the definitions of *definitely correct* and *definitely incorrect;* small chance of a mistaken ACCEPT; very small chance of a mistaken REJECT; and small run-time.

To generalize: let our definition of *definitely incorrect* be that there exists $|\Delta_i| \geq c\sqrt{n}\,\delta$. Let our algorithm calculate t values of $D^{(k)}$ and reject iff any $|D^{(k)}| \geq c\sqrt{n}\,\delta$. (We used $c = 6$, $t = 10$ above.) Then the error bound in Lemma 1 is $2e^{-c^2/2}$. The error bound in Lemma 3 (I) is $2te^{-c^2/2}$, and that in Lemma 3 (II) is $\left(\frac{1}{2}\right)^t$. Lemma 4 (III) generalizes: for any $q \in (0, 1]$, the probability that an at least q portion of all *definitely incorrect* I/O pairs will be mistakenly accepted is $\leq \frac{\rho}{q}$, where ρ is the error bound from Lemma 3 (II). Lemma 4 (IV) generalizes analogously. Lemma 5 applies to any simple checker which has probability of error $\leq 2^{-10}$ (and any checker can have its error-bound reduced to this level by repeating the check a small constant number of times).

2.4 A Self-Corrector

Motivation: To develop a self-corrector for **P**, we start with the traditional approach of adding a random displacement to the location at which we must poll **P**. However, to prevent the components of our input-vectors from going out of legal domain $[-1, 1]$, we here find it necessary to poll **P** at a location which is a weighted average of desired input $\vec{x}$ and a random input $\vec{r}$. If we weight the average so that $\vec{r}$ dominates, the resulting vector is near-uniform random, allowing us to prove the reliability of our self-corrector. This method may be compared to those employed in [Gemmell et al. 1991].

Assume that we have tested **P** on a large number of **random input vectors:** i.e., vectors (of fixed length n) generated by choosing each component uniform-randomly from the set of fixed-point real numbers on legal domain $[-1, 1]$. Assume we have verified that **P** is correct on these random input vectors ($\pm$ an allowable error-delta); a version of our simple checker might be employed to facilitate this verification. Through such a testing stage (or by use of a self-tester [Ergün 1995]), we can satisfy ourselves that (with high probability) the fraction of inputs on which **P** returns incorrect output is very small: say, $\leq \frac{1}{10{,}000{,}000}$.

Once we have this assurance, we can employ the following self-corrector for **P**, whose time-bound is two calls to **P** plus $O(n)$:

ALGORITHM 4. *On input $\vec{x}$,*

—*Generate random input vector $\vec{r}$.*

—*Call* **P** *to calculate* $\mathbf{P}(\vec{r})$.

—*Call* **P** *to calculate* $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$. *(Note that this is possible because, for any legal input vectors $\vec{x}$ and $\vec{r}$, $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ will also be legal—i.e., each component will be in legal domain $[-1, 1]$.)*

—*Return, as our corrected value for* $\mathbf{P}(\vec{x})$,

$$\vec{y}_c := n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot \mathbf{P}(\vec{r}).$$

LEMMA 8. *For any $\vec{x}$:*

$$\Pr_{\vec{r}}\left[\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) \textit{ is incorrect}\right] \leq \frac{3}{10{,}000{,}000}.$$

PROOF. Fix $\vec{x}$. As $\vec{r}$ varies over all legal input vectors, each component of vector $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ varies over a $\left(1-\frac{1}{n}\right)$ fraction of legal domain $[-1, 1]$. Thus, since $\vec{r}$ is a random input vector, the value of $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$ is distributed uniform-randomly[9] over a "neighborhood" of input vectors whose size, as a fraction of the space of all legal input vectors, is $\left(1-\frac{1}{n}\right)^n \approx \frac{1}{e} > \frac{1}{3}$.

So the probability of hitting an incorrect output at $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$ is at most three times the probability of hitting an incorrect output at a truly uniform-random location, which we know to be $\leq \frac{1}{10{,}000{,}000}$. □

LEMMA 9. *For any $\vec{x}$: $\vec{y}_c$ will (approximately) equal desired output $\vec{x}A$, with probability of error (over the choice of $\vec{r}$) $\leq \frac{4}{10{,}000{,}000}$.*

PROOF. Based on testing and on Lemma 8, we know that $\mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)$ and $\mathbf{P}(\vec{r})$ are both likely to be correct: chance of error is $\leq \frac{3}{10{,}000{,}000} + \frac{1}{10{,}000{,}000} = \frac{4}{10{,}000{,}000}$. And if they are indeed both correct, then (barring problems of round-off error, which will be discussed below),

$$\vec{y}_c \;=\; n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot \mathbf{P}(\vec{r})$$

[9] Arithmetic rounding must be done with care to assure this uniformity.

$$\begin{aligned}
&= n\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right)A - (n-1)\vec{r}A \\
&= n\left(\frac{1}{n}\right)\vec{x}A + n\left(1-\frac{1}{n}\right)\vec{r}A - (n-1)\vec{r}A \\
&= \vec{x}A. \quad \square
\end{aligned}$$

Unfortunately, our self-corrector reduces the arithmetic precision of $\mathbf{P}$. For, in calculating

$$\vec{y}_c := n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r}),$$

$\log n$ bits of information are lost when we divide $\vec{x}$ by n (assuming fixed-point arithmetic). Moreover, the final multiplications by n and $(n-1)$ will magnify the $\pm\delta$ errors in the values returned by $\mathbf{P}$. It would seem, then, that the self-corrected $\mathbf{P}$ may have poly(n) times the arithmetic error of the original $\mathbf{P}$. A more conclusive error-analysis would require knowledge of the internals of $\mathbf{P}$.

Thus we will have greater confidence in this corrector if our system allows for the use of more bits of arithmetic precision than are seemingly required for the ordinary operation of $\mathbf{P}$. In particular, $O(\log n)$ bits of additional precision might be expected to compensate for a poly(n)-factor increase in error. However, a more detailed error-analysis, requiring knowledge of the internals of $\mathbf{P}$, would be needed to determine whether or not this is in fact true.

Rather than delving further into error-analyses, we merely note that questions of the round-off errors which checkers and correctors should expect and will produce are important to real-number checking, and may be of interest to the numerical analysis community.

2.5 A Faster Self-Corrector

Another problem with our self-corrector is that it slows execution speed by a factor of at least 2. We can fix this problem by employing the method of *stored randomness* (Section 1.6):

ALGORITHM 5. *Prior to run-time, we generate and store a random input vector $\vec{r}$. We also calculate and store $(n-1)\mathbf{P}(\vec{r})$. By repeatedly using these stored values, we then need only one run-time call to $\mathbf{P}$ to calculate*

$$\vec{y}_c := n \cdot \mathbf{P}\left(\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}\right) - (n-1)\cdot\mathbf{P}(\vec{r}).$$

Observe that our modified self-corrector—like our simple checker—adds only $O(n)$ arithmetic operations to the execution-time of $\mathbf{P}$. Assuming that $\mathbf{P}$ is an $\omega(n)$-time program, the resulting loss of performance should be negligible.

Proceeding as in Lemma 4, we can readily derive the following results—results which suggest that the use of stored randomness does not seriously undermine the reliability of our corrector:

LEMMA 10. *Assume that we generate "random package" $R = \langle \vec{r}, (n-1)\mathbf{P}(\vec{r})\rangle$ and use this stored package to self-correct ℓ runs of $\mathbf{P}$. Then, for probabilities over the choice of R:*

(I). *If $\ell \leq 10{,}000$, then the probability that* any *corrected output will be erroneous is $\leq \frac{4}{1{,}000}$.*

(II). *For any ℓ, the probability that $\geq \frac{1}{10{,}000}$ of the corrected outputs will be erroneous is $\leq \frac{4}{1{,}000}$.*

Lemma 10, like Lemma 4, is not entirely satisfactory. As in Section 2.1, we can derive a stronger result by using a slightly more sophisticated method:

Fix input length: assume that N bits suffice to represent any legal input $\vec{x}$. Also fix the behavior of $\mathbf{P}$: i.e., assume (as is usual for a self-corrector) that $\mathbf{P}$ is not able to adapt in response to the corrector.

Algorithm 6. *At preprocessing time, we generate, not one random package $R = \langle \vec{r}, (n-1)\mathbf{P}(\vec{r}) \rangle$, but $4N$ packages $R_1^*, \ldots, R_{4N}^*$. Then, for each run-time self-correction, we use package $R_i^* \in_u \{R_1^*, \ldots, R_{4N}^*\}$.*

The following lemma argues that each run-time self-correction will then have a small independent probability of error. Proof is analogous to that of Lemma 5.

Lemma 11. *With probability of failure $\leq 2^{-5N}$, $R_1^*, \ldots, R_{4N}^*$ will be such that, for* all *$\vec{x}$, at most $\frac{1}{4}$ of $R_1^*, \ldots, R_{4N}^*$ are bad for correcting $\mathbf{P}(\vec{x})$. Moreover, by increasing the number of stored packages we can reduce error-bounds 2^{-5N} and $\frac{1}{4}$ as desired.*

This is a general method allowing *any* self-corrector to take advantage of stored randomness without compromising its reliability. However, a difference between this result and Lemma 5 should be noted: here $\mathbf{P}$ is not allowed to be an adversary which can adapt to the corrector. This seems a necessary requirement, as a program knowing the values of stored packages $R_1^*, \ldots, R_{4N}^*$ could readily make use of this knowledge to fool the corrector.

2.6 Self-Corrector—Variants

- To generalize: let p be an upper bound on the portion of input vectors $\vec{x}$ for which $\mathbf{P}(\vec{x})$ is incorrect. (We used $p = \frac{1}{10{,}000{,}000}$ above.) Then the error bound in Lemma 8 is ep, and that in Lemma 9 is $(e+1)p$. Lemma 10 (II) generalizes: for any $q \in (0,1]$, the probability that an at least q portion of the corrected outputs will be erroneous is $\leq \frac{(e+1)p}{q}$. Lemma 11 applies to any self-corrector which has probability of error $\leq 2^{-10}$.
- Having calculated corrected output $\vec{y}_c$ as described above, we could then employ a version of our simple checker to verify the correctness of $\vec{y}_c$. This modified form of $\mathbf{P}$ then provides both checking and self-correcting, while adding only $O(n)$ arithmetic operations to $\mathbf{P}$'s usual execution-time.
- The checking situation would be somewhat different if our computer used floating-point, rather than fixed-point, representations. Our model of $\mathbf{P}$'s arithmetic errors as a flat $\pm\delta$ in each component of output would need modification, and the problem of designing expressions such as $\left(\frac{1}{n}\right)\vec{x} + \left(1-\frac{1}{n}\right)\vec{r}$—i.e., randomized expressions near-uniformly distributed over the set of legal inputs—would be more complex. In [Blum and Wasserman 1996], we deal with floating-point arithmetic by checking only operations on normalized mantissas (for we regard such

operations as the most arduous and error-prone portion of microprocessor arithmetic); we thereby reduce, essentially, to the fixed-point case.

3. CONCLUSION

We have stated our belief that result-checking may prove a valuable tool for software debugging and for the creation of extremely reliable systems. Continuing work toward this goal advances on two fronts. First there is theoretical research: work which adapts checking methodologies to make them more easily applicable. This can include: making checking algorithms more efficient; extending checking to work with limited-accuracy real numbers; and developing philosophical/mathematical analyses of the potential reliability of checkers.

The current paper specifies an efficient linear-function checker (Sections 2.1–2.3) and corrector (Sections 2.4–2.6), using *stored randomness* as a "cheat" to save time. These algorithms also deal with issues of checking real-number computations, extending the work of [Gemmell et al. 1991; Ar et al. 1993]. Moreover, Section 1.5 gives some preliminary result-checking "philosophy." The reader may also refer to [Blum and Wasserman 1996], which considers the checking of symbolic mathematics libraries and of microprocessor arithmetic.

Future theoretical research might be expected to derive checking algorithms with powers analogous to those of the current paper, but applicable to broader classes of functions. Also interesting is the topic of *stored randomness*—and, in general, of saving time by cheating work into a preprocessing stage. The methods of Sections 2.1 and 2.5 seem to be the only checking algorithms so far to make essential use of preprocessing. More examples will perhaps be forthcoming.

Result-checking theory must be joined with applied research. With Hughes Aircraft we are conducting pilot studies of checker-based debugging [Boettcher and Mellema 1995]. In particular, Dr. David Shreve of Hughes has implemented the simple checker of Section 2.1 as a means of testing a Fourier Transform used in radar software. Through such studies, it must be determined how effective result-checkers are at flagging bugs, and whether creating checkers is an efficient use of programming man-hours. As mentioned in Section 1.5, the problem of debugging checkers is also intriguing.

Finally, an extensive topic for further study, bridging theory and application, is the creation of partial checkers for "messy" functionalities commonly found in real-world software. Applied result-checking will at last depend on the development of a rich, general family of heuristics for software checking.

The following bibliography of checking literature is only partial. It is not intended as a comprehensive survey.

REFERENCES

Alon, N., Spencer, J. H., and Erdős, P. 1992. *The Probabilistic Method.* John Wiley and Sons. Cited for a Chernoff Bound in Section 2.1.

Ar, S., Blum, M., Codenotti, B., and Gemmell, P. 1993. Checking approximate computations over the reals. In *Proc. 25th ACM Symp. Theory of Computing* (1993), pp. 786–795. Extends checking methodologies to computations on limited-accuracy real numbers. Examples: matrix multiplication, inversion, and determinant; solving systems of linear equations.

Arora, S. and Safra, S. 1992. Probabilistic checking of proofs; a new characterization of NP. In *Proc. 33rd IEEE Symp. Foundations of Computer Science* (1992), pp. 2–13. Equates NP languages with those in interactive-proof class PCP($\log n$, $\sqrt{\log n}$).

BABAI, L. AND FORTNOW, L. 1991. Arithmetization: a new method in structural complexity theory. *Computational Complexity 1*, 1, 41–46. Preliminary version: "A characterization of #P by arithmetic straight line programs," *Proc. 31st IEEE Symp. Foundations of Computer Science* (1990), pp. 26–34. Further develops a technique from [Babai, Fortnow, and Lund] of translating Boolean formulae into multivariate polynomials; hence allows for polynomial concepts to be applied to the study of certain complexity classes.

BABAI, L., FORTNOW, L., LEVIN, L., AND SZEGEDY, M. 1991. Checking computations in polylogarithmic time. In *Proc. 23rd ACM Symp. Theory of Computing* (1991), pp. 21–31. A variant of [Babai, Fortnow, and Lund] with lower time-bounds, this paper introduces an unusual type of very fast checker for NP computations. Such checkers could be regarded as "hardware checkers," in that they ensure that the hardware follows instructions correctly, but don't ensure that the software is correct.

BABAI, L., FORTNOW, L., AND LUND, C. 1991. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity 1*, 1, 3–40. Preliminary version: *Proc. 31st IEEE Symp. Foundations of Computer Science* (1990), pp. 16–25. Proves that NEXP = 2IP, and hence that NEXP-complete problems have complex checkers.

BEAVER, D. AND FEIGENBAUM, J. 1990. Hiding instances in multioracle queries. In *Proc. 7th Symp. Theoretical Aspects of Computer Science* (1990), pp. 37–48. Considers the question: can one get one or more servers to compute $f(x)$ without revealing the value of x to any server? Also proves that polynomials can be self-corrected.

BEIGEL, R. AND FEIGENBAUM, J. 1992. On being incoherent without being very hard. *Computational Complexity 2*, 1, 1–17. Response to questions of [Blum and Kannan 1995; Yao 1990], including a proof that all NP-complete languages are coherent.

BENTLEY, J. 1986. *Programming Pearls*. Addison-Wesley. A discussion here of the difficulty of writing a correct binary-search program is cited in Section 1.7.

BLUM, A. 1994. Personal communication. Idea of weak checking (Section 1.7).

BLUM, M. 1988. Designing programs to check their work. Technical Report TR-88-009, Int'l Computer Science Institute. Formal introduction of simple and complex checking (though anticipated by, e.g., [Freivalds 1979]). Considers checking of graph isomorphism, sorting, and several group-theoretic computations. See also [Blum and Kannan 1995].

BLUM, M., EVANS, W., GEMMELL, P., KANNAN, S., AND NAOR, M. 1994. Checking the correctness of memories. *Algorithmica 12*, 2/3 (Aug./Sept.), 225–244. Introduces data-structure checking. Demonstrates that, given a small, secure database, one may check the correctness of a large, adversarial database.

BLUM, M. AND KANNAN, S. 1995. Designing programs that check their work. *Journal of the ACM 42*, 1 (Jan.), 269–291. Preliminary version: *Proc. 21st ACM Symp. Theory of Computing* (1989), pp. 86–97. Closely related to [Blum 1988].

BLUM, M., LUBY, M., AND RUBINFELD, R. 1993. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences 47*, 3, 549–595. Preliminary version: *Proc. 22nd ACM Symp. Theory of Computing* (1990), pp. 73–83. Introduces self-testing and self-correcting, and gives applications to a variety of fundamental mathematical computations.

BLUM, M. AND WASSERMAN, H. 1996. Reflections on the pentium division bug. *IEEE Transactions on Computers 45*, 4 (April), 385–393. A companion-piece to the current paper; argues that checking and correcting may be used to enhance the reliability of microprocessor arithmetic.

BOETTCHER, C. AND MELLEMA, D. J. 1995. Program checkers: practical applications to real-time software. In *Test Facility Working Group Conf.* (1995). Progress report on a pilot study with Manuel Blum intended to test the usefulness of result-checking as a software development tool.

BUTLER, R. W. AND FINELLI, G. B. 1993. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering 19*, 1 (Jan.), 3–12. Demonstrates inherent limitations of conventional software testing and of **fault tolerance.**

Cleve, R. and Luby, M. 1990. A note on self-testing/correcting methods for trigonometric functions. Technical Report 90-032, Int'l Computer Science Institute. Applies the methods of [Blum et al. 1993] to trigonometric functions.

Ergün, F. 1995. Testing multivariate linear functions: overcoming the generator bottleneck. In *Proc. 27th ACM Symp. Theory of Computing* (1995), pp. 407–416. Extends self-testing to functions (e.g., the Fourier Transform) for which traditional self-testers prove inefficient. See also [Ravikumar and Sivakumar 1995].

Fortnow, L. and Sipser, M. 1988. Are there interactive protocols for co-NP languages? *Information Processing Letters 28*, 5 (Aug.), 249–251. Suggests that co-NP may not be contained in IP. [Lund et al. 1992] later proved the contrary.

Freivalds, R. 1979. Fast probabilistic algorithms. In *Mathematical Foundations of Computer Science*, Number 74 in Lecture Notes in Computer Science, pp. 57–69. Springer-Verlag. Includes a simple checker for matrix multiplication.

Gemmell, P., Lipton, R. J., Rubinfeld, R., Sudan, M., and Wigderson, A. 1991. Self-testing/correcting for polynomials and for approximate functions. In *Proc. 23rd ACM Symp. Theory of Computing* (1991), pp. 32–42. Extends self-testing/correcting of polynomials to several difficult cases: e.g., for domains other than finite fields, or for programs capable of (limited) adversarial adaptation. Also commences the extension of traditional checking methodologies to computations on limited-accuracy reals.

Gemmell, P. and Sudan, M. 1992. Highly resilient correctors for polynomials. *Information Processing Letters 43*, 4 (Sept.), 169–174. Gives near-optimal self-correctors for programs which claim to compute multivariate polynomials. As long as a program is correct on a $\frac{1}{2}+\delta$ fraction of inputs (for $\delta \in \Re^+$), self-correcting is possible.

Goldreich, O., Micali, S., and Wigderson, A. 1991. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM 38*, 3 (July), 691–729. Preliminary version: *Proc. 27th IEEE Symp. Foundations of Computer Science* (1986), pp. 174–187. Includes interactive-proof algorithms for problems including graph isomorphism.

Kannan, S. 1990. *Program Result-Checking with Applications.* Ph.D. thesis, Computer Science Division, University of California, Berkeley. Includes checkers for group-theory and linear-algebra problems.

Lipton, R. J. 1990. Efficient checking of computations. In *Proc. 7th Symp. Theoretical Aspects of Computer Science*, Number 415 in Lecture Notes in Computer Science, pp. 207–215. Springer-Verlag. Proves that logspace suffices to check many computations.

Lipton, R. J. 1991. New directions in testing. In *Distributed Computing and Cryptography*, Volume 2 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 191–202. American Mathematics Society. Independently paralleling [Blum et al. 1993], introduces what are essentially self-correctors (though under a different name). Proves that #P-complete problems have self-correctors.

Lund, C., Fortnow, L., Karloff, H., and Nisan, N. 1992. Algebraic methods for interactive proof systems. *Journal of the ACM 39*, 4 (Oct.), 859–868. Preliminary version: *Proc. 31st IEEE Symp. Foundations of Computer Science* (1990), pp. 2–10. Methodology for constructing an interactive proof system for any language in the polynomial-time hierarchy. Applications to [Shamir; Babai, Fortnow, and Lund].

Micali, S. 1992. CS proofs and error-detecting computation. Technical report, MIT Lab for Computer Science. Also: "CS proofs," Technical Report TM-510, MIT Lab for Computer Science, 1994. Gives result-checkers for NP-complete problems, subject to the assumptions that we have available a random oracle which can serve as a cryptographically secure hash-function, and that the program being checked has insufficient time to find collisions in this hash-function.

Nisan, N. 1989. Co-SAT has multi-prover interactive proofs. E-mail announcement. Initiated events leading to [Lund, Fortnow, Karloff, and Nisan; Shamir; Babai, Fortnow, and Lund].

Ravikumar, S. and Sivakumar, D. 1995. On self-testing without the generator bottleneck. In *Proc. 15th Conf. Foundations of Software Technology and Theoretical Computer Science*, Number 1026 in Lecture Notes in Computer Science, pp. 248–262. Springer-Verlag. Extends the results of [Ergün 1995].

Rubinfeld, R. 1990. *A Mathematical Theory of Self-Checking, Self-Testing, and Self-Correcting Programs.* Ph.D. thesis, Computer Science Division, University of California, Berkeley. Closely related to [Blum et al. 1993].

Rubinfeld, R. 1992. Batch checking with applications to linear functions. *Information Processing Letters 42*, 2 (May), 77–80. Introduces **batch checking**: i.e., collecting I/O pairs and checking them simultaneously. For certain functions it is possible for this combined check to be more efficient than separate checks.

Rubinfeld, R. 1994. On the robustness of functional equations. In *Proc. 35th IEEE Symp. Foundations of Computer Science* (1994), pp. 288–299. Shows that functions satisfying any of a broad class of **functional equations** (e.g., $f(x+y) = f(x) + f(y)$) may be self-tested and self-corrected. Hence specifies testers and correctors for many naturally occurring functions, such as $\tan x$, $1/(1+\cot x)$, and $\cosh x$.

Rubinfeld, R. and Sudan, M. 1992. Self-testing polynomial functions efficiently and over rational domains. In *Proc. 3rd ACM-SIAM Symp. Discrete Algorithms* (1992), pp. 23–32. Extends checking methodologies from finite fields to integer and rational domains.

Rubinfeld, R. and Sudan, M. 1996. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing 25*, 2 (April), 252–271. Demonstrates that polynomials have properties which allow them to be self-tested and self-corrected. See also [Rubinfeld 1994].

Schwartz, J. T. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (Oct.), 701–717. A fundamental result (credit for which is also given to Zippel and DeMillo–Lipton): to determine (with high probability) whether two polynomials are identical, it generally suffices to check their equality at a random location. Applications include: checking multiset equality; proving that two straight-line arithmetic programs compute the same function.

Shamir, A. 1992. IP = PSPACE. *Journal of the ACM 39*, 4 (Oct.), 869–877. Preliminary version: *Proc. 31st IEEE Symp. Foundations of Computer Science* (1990), pp. 11–15. It follows from this result that a program **P** which claims to solve a PSPACE-complete problem may be checked in polynomial time plus a polynomial number of calls to **P**.

Vainstein, F. S. 1991. Error detection and correction in numerical computations by algebraic methods. In *Proc. 9th Symp. Applied Algebra, Algebraic Algorithms and Error–Correcting Codes*, Number 539 in Lecture Notes in Computer Science, pp. 456–464. Springer-Verlag. See [Vainstein 1993].

Vainstein, F. S. 1993. *Algebraic Methods in Hardware/Software Testing.* Ph.D. thesis, EECS Department, Boston University. Uses the theory of algebraic and transcendental field-extensions to design partial complex checkers for all functions that may be constructed from x, e^x, $\sin(ax+b)$, and $\cos(ax+b)$ using operators $+ - \times \div$ and fractional exponentiation.

Valiant, L. G. 1979. The complexity of computing the permanent. *Theoretical Computer Science 8*, 2 (April), 189–201. Defines #P-completeness and proves that computing the permanent of a matrix is #P-complete. See also [Lipton 1991].

Wegman, M. N. and Carter, J. L. 1981. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences 22*, 3 (June), 265–279. Includes an idea for a simple check of multiset equality (completed in [Blum 1988]).

Yao, A. C. 1990. Coherent functions and program checkers. In *Proc. 22nd ACM Symp. Theory of Computing* (1990), pp. 84–94. Function f is **coherent** iff on input $\langle x, y\rangle$ one can determine whether or not $f(x)=y$ via a BPP^f algorithm which is not allowed to query f at x. Author proves the existence of incoherent (and thus uncheckable) functions in EXP. See also [Beigel and Feigenbaum 1992].