Defects, Scientific Computation and the Scientific Method

Les Hatton

CIS, Kingston University, UK
lesh@oakcomp.co.uk, http://leshatton.org/

Abstract. Computation has rapidly grown in the last 50 years so that in many scientific areas it is the dominant partner in the practice of science. Unfortunately, unlike the experimental sciences, it does not adhere well to the principles of the scientific method as espoused by for example, the philosopher Karl Popper. Such principles are built around the notions of deniability and reproducibility.
Although much research effort has been spent on measuring the density of software defects, much less has been spent on the more difficult problem of measuring their effect on the output of a program. This paper explores these issues with numerous examples suggesting how this situation might be improved to match the demands of modern science.
Finally it develops a theoretical model based on Shannon information which suggests that software systems have strong implementation independent behaviour and presents supporting evidence.

Keywords: Scientific method, reproducibility, unquantifiable computation

1 Introduction

The thesis of this paper is that many scientific computations are tainted by the presence of unquantifiable software defects. To understand how this has come to pass, it is important to realise two things:-

- Computer science is historically not a particularly critical discipline. In experimental terms, it appears to be considerably less mature than the natural sciences as for example was demonstrated by [1], [2] when assessing the degree to which experiment played a part in typical computer science publications.
- The majority of the empirical research carried out into software defects has concerned itself with quantifying the density of such defects rather than the much more difficult problem of quantifying the *effects* those defects have on the output of scientific computations. For a thorough review, see [3]. The end product of this research suggests that typical residual defect densities in released software seem to be between 1 and 10 per thousand lines of code. Some very good systems may be as good as 0.1 per thousand lines of code,

[4], although it is not always clear if like is being compared with like, (for example, there are numerous ways of measuring lines of code - source with or without comment, or executable lines - and it is rarely clear which one is in use).

1.1 A small diversion on lines of code

I mentioned above that the use of the phrase "line of code" is problematic. It occurs in a number of guises. The simplest way of counting them is to use the number of newlines giving a value known as *SLOC* (Source Line of Code). This is normally shown in text editors and can be counted very simply indeed.

The presence of comments and language pre-processors complicates this leading to alternative measures such as *PPSLOC*, (Pre-Processed Source Line of Code) and *XLOC*, (Executable Lines of Code), neither of which are readily available when code is compiled and require either special tools or hand-coded tools to measure. As a result, most lines of code measured are SLOC. It is possible to understand the relationships between them by correlating them for a given population of code. As a simple example, Figure 1 illustrates SLOC v. XLOC and also bytes for a typical C application. Repeating on larger populations reveals similar relationships allowing us to move between SLOC, PPSLOC, XLOC and bytes with relative ease normalising defect densities as appropriate.

However, as I will show later in a token-based development using Hartley-Shannon Information Theory as eloquently described in [5], lines of code is too crude a measure.

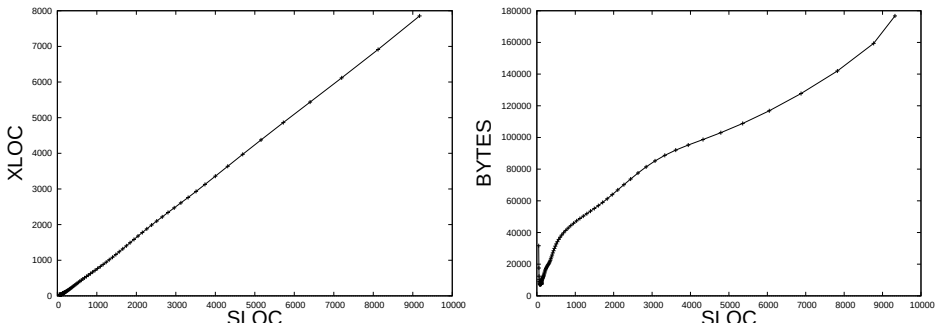

Fig. 1. The left hand diagram is a plot of SLOC count against XLOC count for a typical C application of around 140,000 SLOC in total. The right hand diagram shows the SLOC count against the object code size in bytes generated by compilation with the GNU C compiler. For this application, 1 XLOC = 0.8 SLOC very accurately and 1 SLOC = 25 +/- 3 bytes.

1.2 Software testing and deniability

Finally, it is also worth stating the central tenets of Popperian deniability here cast into a software context.

- Truth cannot be verified by software testing, it can only be falsified,
- Falsification requires quantification of computational modelling error,
- Deniability is at the heart of progress in scientific modelling. We are always seeking to deny the truth of a result and a continued failure to deny such truth simply adds weight to a result but not verification,

It will become clear that scientific source code plays a key part in this process.

2 Quantification of defect

I have distinguished above between the relative success of quantifying defect density, and the much more difficult problem of quantifying their effects. I will now expand on this.

2.1 Defect density and Static Program properties

Even though calculating defect density has been more successful, teasing out any relationships with statically measurable software properties such as the numerous software metrics which have been described in the literature, [6], [7], [8] has been rather less successful.

Complexity For a long time, a considerable amount of hope has been pinned on using statically measured structural properties of a program to predict the occurrence of defect after release, with probably the earliest and most well known being cyclomatic complexity, [9]. Whilst it has value because of its relationship with the number of test cases, [6], there remain difficulties and its originally suggested relationship as a predictor of defect seems illusory at best as can be seen in a study carried out by [10] on the NAG Fortran library. Figure 2 illustrates.

Programming Language Programming language definitions historically reflect the continuing tension between performance and verifiability. Simultaneously, they embody elements of fashion in the form of a need to present the latest features and paradigms to the end-user, even when those features are perhaps not well understood in terms of their capability for injecting defects. A perfect example is the inclusion of object-orientated features into virtually all programming languages in the last twenty years.

The effect of these, coupled with long-term difficulties in removing features of dubious benefit from internationally-standardised languages because of the need to preserve backwards compatibility, has resulted in programming languages

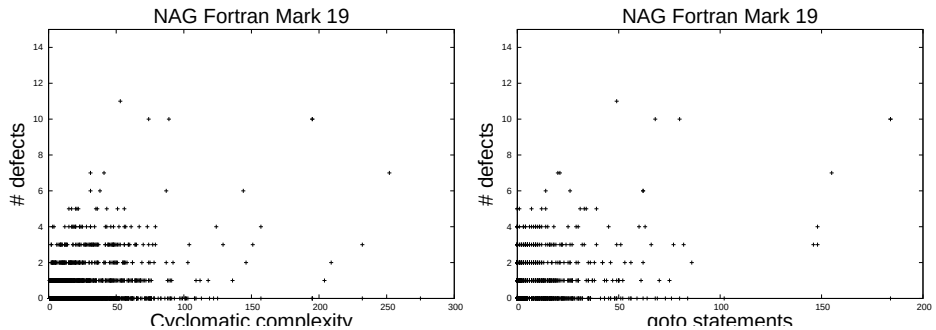

Fig. 2. The left hand diagram is a plot of historical defect against cyclomatic complexity for approximately 20 years history of the NAG Fortran library leading up to Mark 19 shown here. The right hand plot shows the same defects plotted against occurrences of the eponymous goto statement. Neither plot presents any significant statistical correlation of any dependability.

which have grown dramatically in size. Furthermore, they are often punctuated with significant numbers of features which have no defined behaviour and for which there is no requirement for compiler writers to diagnose. Examples include the 191 undefined features of ISO/IEC 9899:1999 (C99), (one of the few languages which actually bothers to list them as an appendix). In addition to these, languages contain features which often lead to erroneous behaviour as exemplified in C by [11] for example.

Although to my knowledge, there has been no published modern attempt to quantify the occurrence of these in released code, [12] demonstrated occurrence rates of around 8 per KSLOC in a study of several MSLOC several years ago, with a number of these packages still in use, whilst [13] demonstrated that these failed with some frequency by measuring an air-traffic control system over several years.

On top of these static fault modes, there are enduring problems with implementations of floating point arithmetic, [14], [15]. These are of fundamental importance to scientists as floating point arithmetic is at the very heart of scientific computation due to the enormous scale over which physical phenomena manifest themselves.

2.2 Quantification of the effect of defect

Whilst we have been fairly successful at understanding the density of such fault modes, (if not preventing them), little progress has been made in quantifying their effect on the computational results themselves, because the problem appears difficult. Several factors contribute to this.

Delayed defect discovery A surprisingly large number of defects take an extraordinarily long time to appear for the first time. In a definitive study,

Adams [16] demonstrated in an analysis of faults and failures in a number of IBM products, that around a third of all faults *took longer than 5,000 executable years to fail for the first time.* This immediately compromises the possible effectiveness of dynamic testing. Based on the kinds of product he analysed, Adams states:-

> "It may well be that as software engineering techniques improve, the population of DEs (Design Errors) will balance at a lower level; but absent development methods that generate truly error-free code, the same sort of error rate distribution may well persist in future large products"

This was written almost thirty years ago and we are certainly still "absent methods that generate truly error-free code".

Unknown answers In many if not most areas of scientific computation, we don't know what the answer is except perhaps in the broadest terms. This is particularly a problem in remote sensing where corroborating physical experiments on the target phenomena cannot actually be carried out at all because they are simply inaccessible, either temporally (for example in back-casting numerical climate models) or spatially, (seismological data). This latter will be the topic of an experiment I will describe shortly. In such cases, rough order of magnitude estimates may be all that is available and as will be seen, this is insufficient to diagnose significant long-present defects.

Access to source code It is only relatively recently, since the real advent of open source, that source code has been widely available in any area. However, in spite of the fact that there is very significant evidence of its pivotal part in defect discovery, it is still not a requirement to parcel up the source code with the algorithmic research, the data and the means to reproduce the results, the very essence of the scientific method. Some research groups, for example, [17] have led the way but progress is slow and even prestigious journals such as Nature remain ambivalent, [18] stating:-

> "Nature does not require authors to make code available, but we do expect a description detailed enough to allow others to write their own code to do similar analysis"

Software testing Software testing remains the Cinderella profession in Computing. It is not usually a significant part of the CS curriculum in universities, [19] and it is unclear whether this deficiency is ever addressed successfully in organisations.

N-Version One methodology which at least casts some light on the magnitude of errors in computation is known as N-version or back-to-back testing. In this approach, the same program specifications are given to N different groups who develop one version each independently, sometimes in different programming languages. These N versions are then given the same input data and any differences in the outputs must be explained. There are two significant disadvantages.

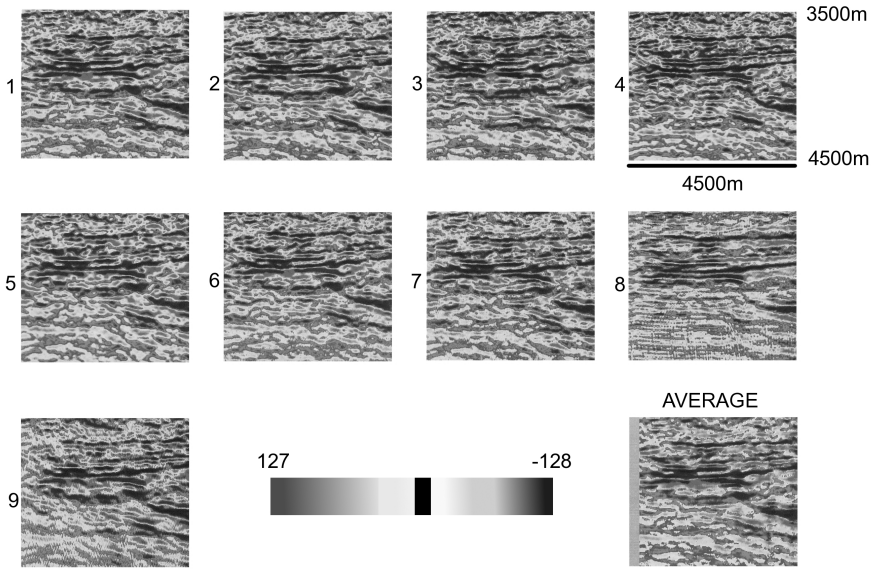

Fig. 3. A comparison of nine independently developed packages in the same programming language on the same input seismological data shown by [22]. The y-axis is depth of burial in the earth and the x-axis is distance along the surface of the earth. The outputs vary in the second and sometimes first significant figure. The data needs about three significant figures of accuracy to resolve the geological features (in this case an unconformity trap for a gas field in the North Sea) sufficiently accurately for statistically reliable positioning of a well.

- Cost. Since they must be independently developed, there are no economies of scale so the cost of development is effectively N times the cost of a single version.
- Independence. Important experiments such as those of [20] and [21] have demonstrated that there are dependent failures even in packages developed completely independently.

In spite of these deficiencies, N-version experiments have demonstrated their value in flushing out very long-lived defects which had evaded any other technique. In [22], nine different seismic data processing packages which had evolved independently in a commercial environment to very well-specified standard algorithms were tested by giving them an identical set of 32 bit floating point input data. After an identical processing sequence, the individual results differed in the 2nd and sometimes 1st significant figure. The results can be seen in Figure 3.

Amongst other things, the paper concluded

- The differences were due to previously undiscovered software faults, in some cases remaining hidden for many execution years.
- The initial 6 significant figures of agreement had shrunk to 1-2 by the time the data was passed to the scientist end-user for interpretation.
- The differences in the final datasets were non-random and therefore more likely to mislead.
- Each software fault which was identified and corrected caused the differences to reduce, so there was convergence although of course it is not possible to say what it was converging to as this is a remote sensing environment with the end product effectively inaccessible. (Drilling a gas well does not validate data as the act of drilling itself interferes with the lithology.)

Although conducted almost twenty years ago, the language used by all participants is still widely used in one form or another (Fortran), the software and test processes used by the participants are also still used and software engineers haven't changed. In other words, it seems likely that the lessons of this experiment are just as valid today.

Open Source It is believed that open source has a beneficial ameliorating effect on defect, [23], [24], [25] and numerous other authors. This is simply an extension of the quoted effectiveness of code inspections, [26] and [27] amongst many. Although in some senses obvious, the mechanisms are not clear although it may be a simple analogue of N-version experiments where there is one version but N independent sets of eyes rather than N independent versions. This is coupled in the open source world with a form of Darwinian overturn whereby the same feature set may appear many times but the best ones are adopted by the community and further strengthened. As in nature, the unsuccessful ones simply disappear.

Whatever model we ascribe to this process, there seems little doubt of its effectiveness. I have included it under the topic of quantifying the effects of defect as it is also commonly associated with a very close relationship between development and testing as occurs in the Linux kernel[1].

3 A theory of defect

One of the things engineers often note about software systems is that the same things occur again and again, [28]. To take one particular example, it is very often observed that defects appear to cluster, [29], [28], independently of either programming language or application area. Following on from [30], I will investigate this using an information theoretic model to avoid the straitjacket of dependence on line of code measures. This does require the development of tools to extract the tokens so is rather more effort than extracting SLOC but that effort proves to be important.

[1] http://www.ibm.com/developerworks/linux/library/l-stress/index.html, accessed 18-Oct-2011

All languages are specified by such tokens, which are extracted at the lexical analysis stage of a language compiler or interpreter. In this sense a token of a programming language takes one of two forms:-

Fixed token Fixed tokens of programming languages are those tokens specified by the language designer whose form cannot be altered - the programmer either uses them or not. Examples include language keywords such as **if**, **then**, **while**; structural tokens such as [,] and operators such as +, **-**, ***** and so on.

Variable token These are the user-specified tokens invented by the programmer in order to implement an algorithm. Examples include identifier names, constants such as 3.14159265 and strings. Apart from some mild lexical constraints such as limiting the length of an identifier to 31 characters and its starting character to be alphabetic, the programmer has complete freedom to invent what he or she chooses.

From this token model, all algorithms in all programming languages are constructed.

3.1 An information theoretic model

Suppose a software system is split up into M pieces, with the i^{th} piece containing t_i tokens altogether from a unique alphabet a_i tokens. I will refer to the pieces as *components*. In simple procedural languages such as Fortran, these would correspond to a function or a subroutine. In an OO language, it would be the outer class. No finer granularity will be used as the mathematical development considers only one level.

Following the discussion above, the unique alphabet can be decomposed as

$$a_i = a_f + a_v(i) \tag{1}$$

where a_f is the alphabet of fixed tokens and $a_v(i)$ is the alphabet of variable tokens and is clearly dependent on i, since programmers are free to create them as and when desired.

The number of ways of arranging the tokens of this alphabet in the i^{th} component is therefore $a_i^{t_i}$. Following Hartley, the quantity of information in the i^{th} component I_i will therefore be defined as

$$I_i = log(a_i)^{t_i} = t_i log a_i \tag{2}$$

We can then see that the total amount of information in a system I, can be written as

$$I = \sum_{i=1}^{M} I_i = \sum_{i=1}^{M} t_i(\frac{I_i}{t_i}) \equiv \sum_{i=1}^{M} t_i I_i' \tag{3}$$

where I_i' is the information density in the i^{th} component. We will see the reason for this transformation shortly.

We can also see that the total system size T is given by

$$T = \sum_{i=1}^{M} t_i \tag{4}$$

We can envisage a software system as a fixed level of functionality within some fixed size. Now functionality is intimately related to choice which as Cherry points out [5], is itself intimately related to Hartley-Shannon information. *It therefore makes sense to find the most likely way in which tokens can be arranged in components subject to the twin constraints that total size and total amount of information are fixed.* This can be solved using basic principles from statistical mechanics as follows.

The total number of different ways of distributing tokens amongst the components is given by:-

$$W = \frac{T!}{t_1!t_2!..t_M!} \tag{5}$$

We will now suppose that the information density of the i^{th} component is externally imposed by the nature of the algorithm and therefore in common with variational principles is kept constant during variation.

Using the method of Lagrangian multipliers, the most likely distribution satisfying equation (5) subject to the constraints in equations (4) and (3) will be found. This is equivalent to maximising the following variational

$$logW = TlogT - \sum_{i=1}^{M} t_i log(t_i) + \lambda\{T - \sum_{i=1}^{M} t_i\} + \beta\{I - \sum_{i=1}^{M} t_i I_i'\} \tag{6}$$

where λ and β are the Lagrange multipliers. Setting $\delta(logW) = 0$ and using the reasonable assumption that the $t_i \gg 1$ leads to

$$0 = -\sum_{i=1}^{M} \delta t_i\{log(t_i) + \alpha + \beta I_i'\} \tag{7}$$

where $\alpha = 1 + \lambda$. This must be true for all variations δt_i and so

$$log(t_i) = -\alpha - \beta I_i' \tag{8}$$

Using (4) to replace α, and defining $p_i = \frac{t_i}{T}$ using (3), p_i can be interpreted as the probability that a component is found with a share of I equal to I_i' and is given by

$$p_i \equiv \frac{t_i}{T} = \frac{e^{-\beta I_i'}}{\sum_{i=1}^{M} e^{-\beta I_i'}} \tag{9}$$

In other words, the probability of finding a component with a large amount of I_i' is correspondingly small. Given the assumed externally imposed nature of I_i',

p_i can then be taken to be the probability that a component of t_i tokens actually occurs.

Using (3) and (9), we can finally write

$$p_i = \frac{e^{-\beta \frac{I_i}{t_i}}}{Q(\beta)} \tag{10}$$

where

$$Q(\beta) = \sum_{i=1}^{M} e^{-\beta \frac{I_i}{t_i}} \tag{11}$$

Combining (10) and (2) then gives

$$p_i = \frac{e^{-\beta log a_i}}{Q(\beta)} \tag{12}$$

This of course is power-law behaviour

$$p_i = \frac{(a_i)^{-\beta}}{Q(\beta)} \tag{13}$$

So far this is a similar development to that followed in [31] and [32] for example, although it generalises the argument by using tokens of programming languages, which are the natural currency of information theory.

Note that this overall process does not care about the tokens themselves - all individual microstates are equally likely. It simply says that if total size and choice in the Hartley-Shannon sense is conserved during the process of distributing the tokens, (and programming is all about choices), then power-law distribution of component size in tokens is overwhelmingly likely to emerge since it occupies the vast majority of the microstates. As will be seen in the data analysis, the specific contribution made by the fact that choice is being made from programming language tokens is represented by the behaviour implicit in (1). This contrasts nicely with monkeys pounding on keyboards as eloquently described by [33]. The ergodic nature of (13) simply accumulates all possible programmers pounding on keyboards. Although not shown here, it also works well with much smaller numbers, i.e. individual systems, a characteristic of classical statistical mechanics.

Finally, I will observe that every language has a *fixed token overhead* in order to implement even the simplest of algorithms. In other words, smaller components must use a higher proportion of fixed tokens than variable tokens. In contrast, larger components use a higher proportion of user-specified tokens because the finite fixed token alphabet quickly stabilises. This can easily be measured. In the very large amount of data reported shortly, the $a_f/a_v(i)$ ratio is typically around 0.2 for smaller components and at least 5 for large components.

It turns out that computing p_i is fundamentally noisy in the tail of power-law distributions and [34] recommends using the equivalent cumulative density function c_i instead. We can then anticipate the final shape of (13) as follows.

Combining (1) and (13) gives

$$c_i \sim (a_f + a_v(i))^{-\beta+1} \tag{14}$$

For small components, as has been seen, it is reasonable to assume that the number of fixed tokens will tend to dominate the total number of tokens. In other words, $a_f \gg a_v(i)$. (14) can then be written

$$c_i \sim (a_f)^{-\beta+1}(1 + \frac{a_v(i)}{a_f})^{-\beta+1} \tag{15}$$

In other words,

$$c_i \sim (a_f)^{-\beta+1} \tag{16}$$

which implies that c_i will be tend to a constant for small components on a log-log plot.

For large components, using the same arguments,

$$c_i \sim (a_v(i))^{-\beta+1} \tag{17}$$

The generic shape of the predicted curve on a log-log plot is shown in Figure 4.

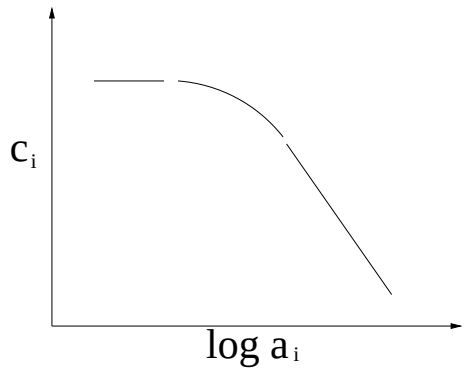

Fig. 4. The predicted cdf using the model described in this paper. The cdf is predicted to be approximately constant for small components and power-law for large ones with a merging zone between.

3.2 Results

To give a sufficiently broad analysis, many systems were analysed in multiple languages, (Java, C, C++, Ada, Fortran, Tcl-Tk). A generic token extractor was developed for each and calibrated against existing parsing engines in Fortran and

C which I had developed in previous projects and which had been tested against the appropriate validation suites, (FCVS and FIPS160 respectively). 75 systems totalling 34 million lines of code (around half a billion tokens) were analysed and the results are shown in Figure 5.

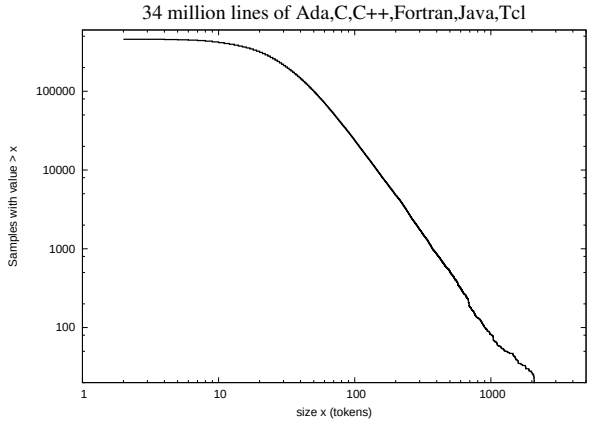

Fig. 5. The measured cdf for 75 systems combining 34 MSLOC into one super-system. This comprises around 15% Java, 15% C++, 15% Fortran, Ada and Tcl combined and around 55% C. This very roughly reflects the amount of each language freely available under open source.

Although the tail of the distribution shown in Figure 5 looks decidely linear, this was confirmed using the linear modelling function (lm()) in the widely-used R statistical package, (http://www.r-project.org/) which reported a very high degree of linearity with a linear-fit correlation of 0.998 between token counts of 30 and 1500, a span of almost two decades. The same analysis reports a slope of -2.404 +/ 0.004, which is squarely in the range -2 $\rightarrow$ -3 reported for most natural phenomena by [34].

If we now use the simplest model of defect, that we make a mistake every N tokens on average, $d_i \sim t_i \sim a_i$ (using Zipf's law [35]), then

$$c_i \sim (a_i)^{-\beta+1} \sim (t_i)^{-\beta+1} \sim (d_i)^{-\beta+1} \tag{18}$$

So defects will also statistically be distributed as a power-law and should exhibit clustering. As discussed above, this has been widely observed, and also exploited, [36].

4 Conclusions

This paper gives a guide to some of the problems of quantifying defect in scientific computation. It also demonstrates that software systems appear to have

implementation independent properties in which power-laws strongly figure and suggests that defects might be fundamentally statistical in nature rather than predictive. The development gives theoretical support to the observation that defects cluster and this phenomenon can be exploited.

N-version experiments to measure difference are formidably expensive although can emphasise that we have a problem but perhaps the only real way forward is through open source and open data so that reproducibility can be consistently achieved as in other parts of science.

Perhaps I can best sum up this paper by the following aphorism:-

> *We make progress in science by peer review. To make progress in scientific computation we must extend this to code review.*

References

1. W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "Experimental evaluation in computer science: a quantitative study," *J. Syst. Softw.*, vol. 28, pp. 9–18, January 1995.
2. W. Tichy, "Should computer scientists experiment more ?," *IEEE Computer*, vol. 31, pp. 32–40, May 1998.
3. B. Boehm, H. Romback, and M. Zelkowitz, *Foundations of empirical software engineering: the legacy of Victor R. Basili*. Springer, 1st ed., 2005. ISBN 3-540-24547-2.
4. T. Keller, "Achieving error-free man-rated software," *Second International Software Testing, Analysis and Review Conference*, 1993. Monterey, USA.
5. C. Cherry, *On Human Communication*. John Wiley Science Editions, 1963. Library of Congress 56-9820.
6. N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. PWS, 2nd ed., 1997.
7. R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, pp. 297–310, April 2003.
8. M. J. van der Meulen and M. A. Revilla, "Correlations between internal software metrics and software dependability in a large population of small c/c++ programs," *Software Reliability Engineering, International Symposium on*, vol. 0, pp. 203–208, 2007.
9. T. McCabe, "A software complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.
10. T. Hopkins and L. Hatton, "Defect correlations in a major numerical library," *Submitted for publication*, 2008. Preprint available at http://www.leshatton.org/NAG01_01-08.html.
11. A. Koenig, *C Traps and Pitfalls*. Addison-Wesley, 1989. ISBN 0-201-17928-8.
12. L. Hatton, "The T experiments: Errors in scientific software," *IEEE Computational Science and Engineering*, vol. 4, pp. 27–38, April 1997.
13. S. Pfleeger and L. Hatton, "Do formal methods really work ?," *IEEE Computer*, vol. 30, no. 2, pp. p.33–43, 1997.
14. W. Kahan and J. Darcy, "How java's floating point hurts everyone everywhere," *Originally presented at ACM 1998 Workshop on Java for HighPerformance Network Computing*, July 2004.

15. W. Kahan, "Desperately needed remedies for the Undebuggability of Large Floating-Point Computations in Science and Engineering," in *IFIP / SIAM / NIST Working Conference on Uncertainty Quantification in Scientific Computing*, 2011.
16. E. Adams, "Optimising preventive service of software products," *IBM Journal of Research and Development*, vol. 1, no. 28, pp. 2–14, 1984.
17. D. Donoho, A. Maleki, I. Rahman, M. Shahram, and V. Stodden, "Reproducible research in computational harmonic analysis," *Computing in Science and Engineering*, vol. 8, no. 18, 2009.
18. Editorial, "Devil in the details," *Nature*, vol. 470, pp. 305–306, 2011.
19. E. Jones, "Software testing in the computer science curriculum – a holistic approach," in *Proceeding ACSE '00 Proceedings of the Australasian conference on Computing education*, (New York, NY, USA), ACM, 2000.
20. J. Knight and N. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 96–109, 1986.
21. M. van der Meulen and M. A. Revilla, "The effectiveness of software diversity in a large population of programs," *IEEE Trans. Software Eng.*, vol. 34, no. 6, pp. 753–764, 2008.
22. L. Hatton and A. Roberts, "How accurate is scientific software ?," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, 1994.
23. A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, (New York, NY, USA), pp. 263–272, ACM, 2000.
24. E. S. Raymond, *The cathedral and the bazaar*. O'Reilly, February 2001.
25. A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 309–346, July 2002.
26. M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 2, pp. 182–211, 1976.
27. W. Humphrey, *A discipline of software engineering*. Addison-Wesley, 1995. ISBN 0-201-54610-8.
28. B. Boehm and V. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.
29. J. Tian and J. Troster, "A comparison of measurement and defect characteristics of new and legacy software systems," *Journal of Systems and Software*, vol. 44, no. 2, pp. 135 – 146, 1998.
30. L. Hatton, "Power-laws, persistence and the distribution of information in software systems," *preprint available at http://www.leshatton.org/variations_2010.html*, January 2010.
31. P. Rawlings, D. Reguera, and H. Reiss, "Entropic basis of the pareto law," *Physica A*, vol. 343, pp. 643–652, July 2004.
32. L. Hatton, "Power-law distributions of component sizes in general software systems," *IEEE Transactions on Software Engineering*, July/August 2009.
33. M. Mitzenmacher, "A brief history of generative models for power-law and lognormal distributions," *Internet Mathematics*, vol. 1, no. 2, pp. 226–251, 2003.
34. M. E. J. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, 2006.
35. M. Shooman, *Software Engineering*. McGraw-Hill, 2nd ed., 1985.
36. A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Softw. Engg.*, vol. 13, no. 5, pp. 473–498, 2008.