Functionality, power-laws and defect evolution in software systems

Les Hatton
CISM, University of Kingston[*]

September 15, 2009

Abstract

In a previous paper, an intimate link between power-law distribution of component sizes and defect growth in maturing software systems, independently of their representation language, was revealed by the use of a variational method built on statistical mechanical arguments.

Building on the above work, this paper first of all demonstrates experimentally that power-law behaviour in component sizes appears to be a persistent property in that it is present from the earliest release of software systems. It then goes on to develop a theoretical model which is able to predict this *a priori* appearance of power-law behaviour in component sizes in developing systems using another variational principle linking functionality with a modification of the Hartley/Shannon information content. This therefore unifies the observed phenomena under twin variational principles, one appropriate to the development phase and one to the release phase and provides a theoretical model for life-cycle component size distributions as well as defect growth in maturing systems.

`Keywords: Defects, Macroscopic system behaviour, Component size distribution, Power-law`

[*] L.Hatton@kingston.ac.uk, lesh@oakcomp.co.uk

1 Background

In [8], a model based on statistical mechanics was presented which showed that, in a software system subject only to the twin constraints of total size and number of defects present *a priori*, a component size distribution exhibiting power-law behaviour in medium to large components (i.e. greater than about 20 executable lines of code (XLOC)) was inextricably linked with a growth in defects within a component of xlogx where x is the number of XLOC. Numerous systems in several different languages were also analysed to show that such power-law behaviour in component size was indeed present. The paper concluded by saying that it was not however clear which if either of these two phenomena was the driver.

The current paper builds on this work and makes a number of further contributions.

- First it rationalises the observations of power-law behaviour in [8] with similar observations in other work.
- Second, it demonstrates using further experimental evidence that power-law behaviour of component sizes in disparate software systems is a *persistent* property in that it appears to be present from the earliest days of a released software system. In other words, it appears to evolve naturally during development driven perhaps by some deeper principle.
- Third, it provides a mechanism which explains the natural appearance of power-law size distributions during development, independently of any representation. The resulting model is a novel modification of the Hartley / Shannon information content within a variational context and attempts to define what might be meant by functionality.
- Fourth, it then unifies this model with that presented in [8]. In this unified model, linearity of information content with component size appears to play a central role in determining first, the appearance of power-law behaviour in component size and from that, a defect behaviour in the maturing system depending on xlogx, where x is the number of executable lines of code. This model is independent of any implementation detail such as programming language or design methodology.

1.1 Power-law behaviour

For reference, power-law behaviour can be represented by the probability p(s) of a certain size s appearing being given by a relationship like:-

$$p(s) = \frac{k}{s^{\alpha}} \tag{1}$$

where k is a constant, which on a log p - log s scale is a straight line with negative slope. The size s will be measured here in executable lines of code.

Power-law behaviour has been studied in a very wide variety of environments, see for example [18] (economic systems) and the excellent review by [16]. In software systems there has been significant activity, much of it recent, [4], [15], [14], [2], [17], [1], [5] and [8] all discuss power-law behaviour but in rather different contexts. Of these, Mitzenmacher's study [14] is particularly relevant to the present work as it considers the distributions of file sizes in general filing systems. Mitzenmacher observed that such file sizes were typically distributed with a lognormal body and a Pareto (i.e. power-law) tail. This does not however conflict with the present work as one of the basic assumptions of the statistical mechanical development in [8] and extended here, is that file sizes are at least medium sized so this would correspond to the Pareto tail[1]. It should also be noted that his work dealt with general file systems rather than discrete but distinct software packages containing numerous strongly related files, together providing the functionality of the software package. This distinction may also be relevant but has not been pursued here.

Given the observed appearance of power-law behaviour in such component sizes reported in numerous software packages of very different provenance by [8], and its central part in the theoretical development both there and here, it is of particular interest to investigate why and when such behaviour would emerge, given its link with defect behaviour within components.

Newman in his comprehensive review of power-law behaviour [16], describes a number of mechanisms which lead to power-law behaviour.

- Inverses of quantities
- Random walks

[1] Medium sized in the context of this paper corresponds to components of $\gtrsim 20$ executable lines of code. This will also be referred to as the *tail* of the distribution.

- The Yule process
- Phase transitions
- Self-organised criticality
- Combinations of exponentials

Of these, the last named will prove particularly useful in the development which follows.

2 Empirical evidence

2.1 Some notes on averaging

Before presenting any evidence, it is worth mentioning in passing that software data is typically very noisy for a variety of reasons. Factors of 10 or more between developers in the same group are often noted by researchers in measuring various parameters, for example fault detection capability during inspections, [7]. In comparisons of software components written independently to the same specification even in the same programming language, significant variations in program size measured in lines of code have similarly been reported, [12], [20]. This means that any kind of software measurement data will necessarily need significant smoothing to reveal any systematic patterns amongst the inevitable noise. Since this paper concerns the distributions of ensembles using statistical mechanical arguments, averaging has been used to reveal patterns of interest. The details of the smoothing will be given as appropriate with the datasets as they are presented but in essence, lines of code are averaged using trimmed means to give robust estimates of the mean.

2.2 Power-law behaviour in size distributions

In each of the cases here, the data will be presented in the log - log formation described earlier and straight line behaviour sought. Figure 1 shows averaged data for the 21 systems described in [8]. These systems are highly disparate in size (5-250 KXLOC), language (C, Fortran, Tcl) and application area, (embedded systems, scientific libraries, geophysical modelling, communications and others).

In spite of this very considerable disparity, the linearity is striking for medium and larger component sizes, (and confirmed for the raw data by significance testing in [8]). Here the data is displayed differently with each

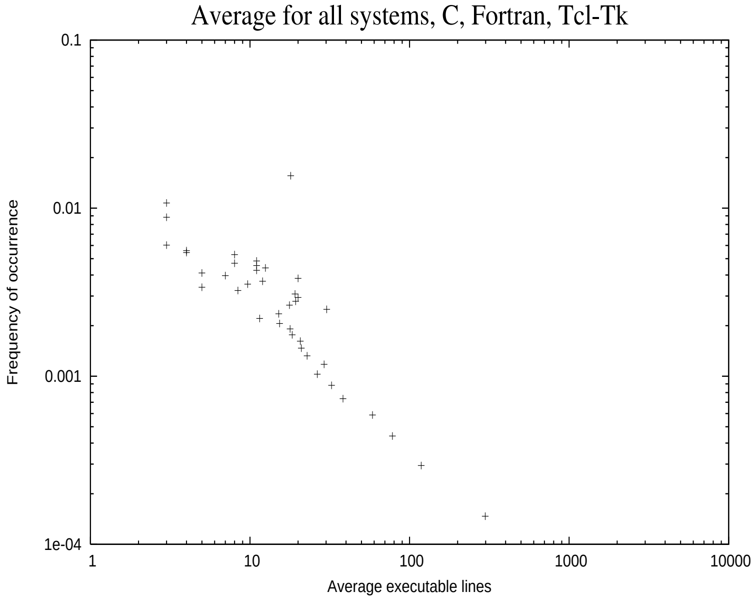

Figure 1: A plot of averaged data for the 21 systems described in [8]. Averaging has been done by computing the trimmed mean over all component sizes contributing to a particular frequency across all systems.

data point itself the average of a large number of component sizes which occurred with a particular frequency. As described above, the data has been averaged by computing the trimmed mean, (a standard technique to calculate the mean excluding outliers to improve the robustness of the estimate).

2.3 Persistence of power-law behaviour in size distributions

Given the strong power-law behaviour shown in Figure 1, it makes sense to ask wherever this information is available, if this behaviour is persistent from the time of the first release, or whether the power-law behaviour emerged subsequent to its first release as the system matured. If power-law behaviour is present from the earliest days of a released system presumably as a result of some underlying principle acting during development, then the mathematical argument in [8] predicts that defect growth in a component proportional to xlogx is most likely to take place as the system matures, where x is the number of executable lines of code. Indeed, some evidence for this functional defect growth was presented using mature systems data but the question remains open whether the power-law behaviour is indeed persistent.

Practical considerations suggest that it would be unusual to expect major changes in component size distribution as a system ages on the general grounds that engineers are reluctant to change working systems too much even as they adapt to changing requirements and other normal maintenance activities. By good fortune, several of the 21 systems analysed in [8] had source revision history, two from the very first release of 7-8 year life-cycles and one from around half way into its 25 year life-cycle from first release. The three systems are highly disparate. To reduce the noise in showing the component-size distributions for each release, the data are displayed in rank order in step format as described by [17]. Here the shape of the distribution is irrelevant, it is the *change* in shape across revisions which is of particular note. If this does not change substantially, then power-law behaviour is persistent, since in each case, it was present in the latest releases.

Figure 2 shows the component size distributions for each official release of a widely used numerical library (the NAG Fortran library) from release 12 through release 19, spanning around 12 years. The last release analysed, release 19, comprised some 370,000 executable lines of code. As expected, there is little substantial change across this time period. For general interest, the data is shown for all component sizes.

It remains possible that substantial change might have taken place in the releases prior to release 12, but the data were not available to confirm this. Accordingly two more systems were studied for which the full life-cycle behaviour was available. These had the benefit of being in different languages and of entirely different application domains. Figure 3 shows the development of a graphical user interface of approximately 43,000 lines of Tcl-Tk code used in geophysical modelling across all 44 revisions from its first appearance in 2002 to the present day, (only every 4th version is shown for clarity). The data is displayed again in rank order and step format, and it can be seen that there is again no substantial difference in component size distribution across the entire released life-cycle. The power-law behaviour in the tail was already present at first release, it is not an emergent property.

A third system is shown as Figure 4. Again this is in a different language, (this time C), a totally different application area (high-integrity C parsing tool) and is of considerable size, (in this case around 128,000 source lines). This system spans 27 separate releases across an 8 year life-cycle and in this figure every 3rd release is shown for clarity. Again, no substantial change is observed and again it must be concluded that the power-law behaviour in

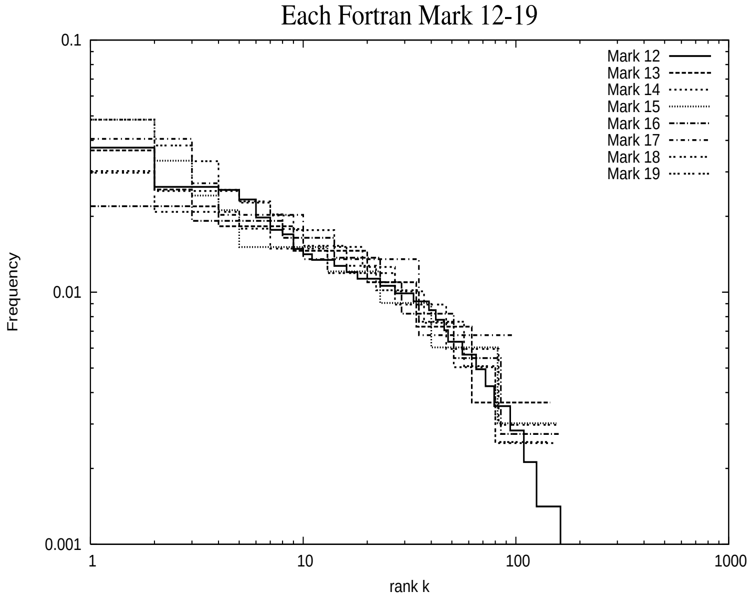

Figure 2: The distribution of component sizes displayed in rank order and step format for various versions of a major Fortran numerical library as reported by [9]. This data covers the development from about a half of the way into the life-cycle up to the present day. No substantial change in distribution is evident across this time-period.

the tail is persistent.

Given the very disparate nature of these three systems, it can be tentatively concluded that component size distributions do not appear to change substantially across long life-cycles of medium to large packages, (here the range is 43,000 - 370,000 executable lines of code in three different languages and very different application areas). Furthermore, all three are included in the population of systems which exhibit emphatic power-law behaviour in the tail of Figure 1, so it will be concluded that there is at least reasonable evidence that such power-law behaviour of component sizes is a persistent property and it does not emerge during the maintenance cycle.

Before returning to the nature of the resulting predictions made in [8], this paper will now address the question as to why power-law behaviour of this nature appears to be present from the earliest releases of general software systems.

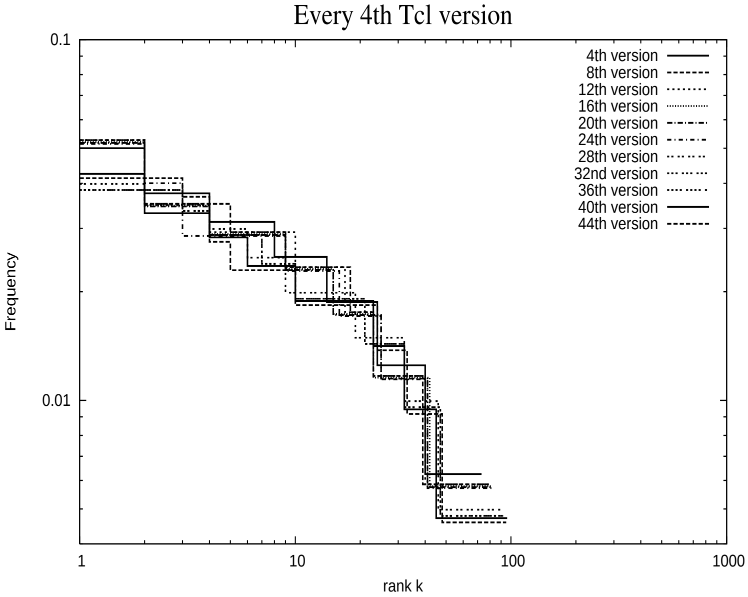

Figure 3: The distribution of component sizes displayed in rank order and step format for every 4th release in the first 44 versions of a GUI development written in Tcl-Tk and covering some 7 years of development. Across the entire life-cycle, there is no substantial change in component size distribution.

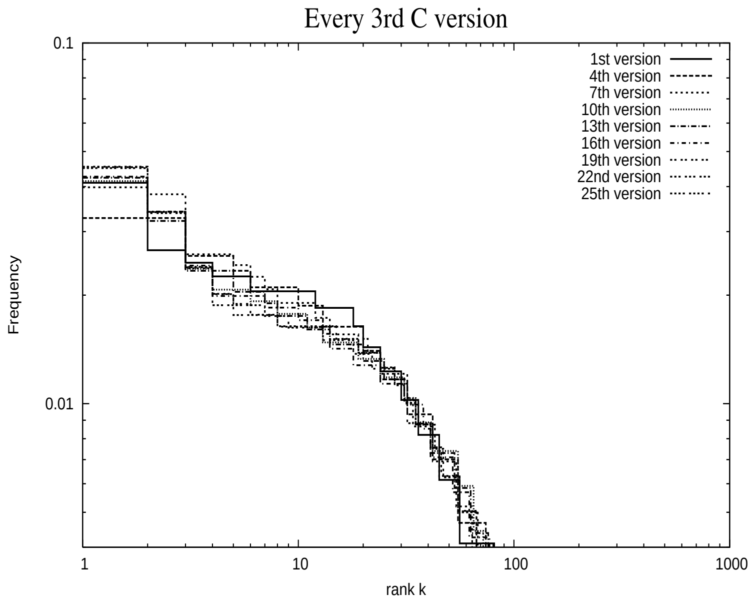

Figure 4: The distribution of component sizes displayed in rank order and step format for every 3rd release in the first 27 versions of a high-integrity parsing engine written in C and covering some 8 years of development. Across the entire life-cycle, there is no substantial change in component size distribution.

3 Functionality in software systems

To be present in the first release, it is obvious that any such power-law behaviour must evolve naturally as the functionality of a system is implemented in a software context during development and must therefore be intimately related to that functionality in some sense and this will be pursued here.

Historically, functionality has proven to be an elusive goal to quantify for computer scientists as evidenced by the fact that the number of lines of code still appears to be the dominant measure of *size*, and by common association, functionality, even though a *line of code* is itself a somewhat arbitrary measure, strongly related to the implementation language and also the developer's personal taste. In addition, different alternatives present themselves, for example, in C and C++, the following are all used:-

- SLOC, (source lines of code). Simply a measure of the count of lines as seen by a text editor.
- PPLOC, (pre-processed lines of code). SLOC contain comments and some argue that they should not be counted. In C and C++, the pre-processor removes comment in a predictable way but also expands header files leading to a definition of PPLOC as a count of the number of pre-processed non-blank lines of code.
- XLOC, (executable lines of code). A count of those lines of source code which cause the compiler to generate executable code. This is the preferred measure here as it is rather less dependent on the nature of the programming language than either of the first two.

Although these are all different measures, they are usually very highly correlated as exemplified by Figure 5 which shows the correlation between SLOC and XLOC for a large population of C programs, so knowledge of any one can be used to predict the others with considerable accuracy. Note that although the correlation is very close, this is not a trivial comparison where comments have been simply removed. Instead for C (and also C++), SLOC first have to be pre-processed which removes comment but expands header files. The resulting code must then be correctly parsed to distinguish for example between a declaration with an initialiser which is considered executable, and one without which is not. There are many other subtle distinctions.

The relationship of any of the line of code measures with functionality is much harder to understand however.

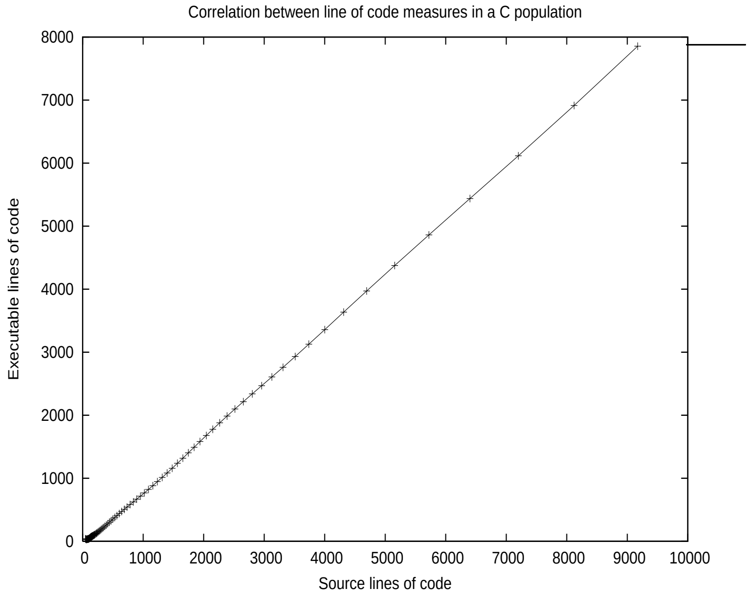

Figure 5: The correlation between SLOC and XLOC in a population of around 300,000 SLOC of C programs.

Because of these difficulties, other alternatives have been proposed to capture the amount of functionality in a system such as the *function point*. However, these also present significant difficulties as shown by [10] and [11].

Since there appears to be no obvious measure and since also this is a statistical approach, the number of lines of code in a component, (function, subroutine, procedure or whatever as described in [8]), will be considered a random variable with some distribution, whose expected value is a generic measure of size. This can best be visualised perhaps as taking the average size of an implementation for all the different implementations of the same specification in the same language which might be made. An example of an experiment in which this could be done is [12], where differences in size of factors of 3 were reported with the same specification and programming language.

3.1 A model for functionality evolution

First then, an abstract model of functionality will be derived based on the defect model development in [8] observing that functionality and defect must be intimately linked in some sense. Suppose that a system is made up of M components each of size n_i executable lines of code such that the total size is given by

$$N = \sum_{i=1}^{M} n_i \tag{2}$$

Suppose there is also some externally imposed entity ε_i associated with each line of component i whose total amount is therefore given by

$$U = \sum_{i=1}^{M} n_i \varepsilon_i \tag{3}$$

Using the method of Lagrangian multipliers, the following variational will be maximised

$$logW = NlogN - \sum_{i=1}^{M} n_i log(n_i) + \gamma\{N - \sum_{i=1}^{M} n_i\} + \beta\{U - \sum_{i=1}^{M} n_i \varepsilon_i\} \tag{4}$$

where γ and β are the multipliers. Setting $\delta(logW) = 0$ leads to

$$0 = -\sum_{i=1}^{M} \delta n_i \{log(n_i) + \alpha + \beta \varepsilon_i\} \tag{5}$$

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

$$log(n_i) = -\alpha - \beta \varepsilon_i \tag{6}$$

Using equation (2) to replace α, this can be manipulated into the most likely, i.e. the equilibrium distribution

$$n_i = \frac{N e^{-\beta \varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta \varepsilon_i}} \tag{7}$$

Following [18] and defining $p_i = \frac{n_i}{N}$ and referring to equation (3), p_i can be interpreted as the probability that a component is found with a share of U equal to ε_i. Manipulating equation (7) then yields

$$p_i = \frac{e^{-\beta \varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta \varepsilon_i}} \tag{8}$$

In other words, the probability of finding a component with a large amount of ε_i is correspondingly small. Given the externally imposed nature of ε_i, p_i can be taken to be the probability that a component of n_i lines actually occurs.

So far this is a completely standard development as followed in [18] and [8] for example.

Now the concept of *functionality* will be introduced. At this stage, it will be construed as having an obvious meaning related to the functional requirements of the system in the following sense. Suppose that the i^{th} component has associated with it a functionality f_i. The total functionality F is therefore given by

$$F = \sum_{i=1}^{M} f_i \tag{9}$$

The same computational device used in [8] will now be applied again. Noting that equation (9) can be written as

$$F = \sum_{i=1}^{M} n_i (\frac{f_i}{n_i}) \tag{10}$$

leads directly to the identification of ε_i with $(\frac{f_i}{n_i})$ in equation (7). In other words, each line of component i has a functionality density associated with it given by $(\frac{f_i}{n_i})$. Note that introducing this additional functional dependence of ε_i on n_i does not disrupt the development which led to equation as ε_i is fixed externally by assumption.

Equation (8) can then be written as

$$p_i = \frac{e^{-\beta \frac{f_i}{n_i}}}{Q(\beta)} \tag{11}$$

where

$$Q(\beta) = \sum_{i=1}^{M} e^{-\beta \frac{f_i}{n_i}} \tag{12}$$

3.2 Hartley-Shannon information content

Now it will be recalled from the initial discussion of power-law behaviour that Newman [16] gives a list of possible mechanisms for the evolution of power-law behaviour of which combination of exponentials turns out to be a fruitful avenue to pursue for software systems as will now be seen.

In essence if some quantity y has an exponential probability distribution

$$p(y) \sim e^{ay} \tag{13}$$

and some other quantity of interest x behaves like

$$x \sim e^{by} \tag{14}$$

Then the probability distribution of p(x) is given by

$$p(x) \sim x^{\frac{a}{b}-1} \tag{15}$$

which is power-law behaviour. The potential attraction of this for software systems is that it has been used before in a textual context by Miller [13] who applied it in order to attempt to explain the observed power-law distribution of the frequencies of words in texts. Miller's work was criticised in that it assumed randomly typing on a keyboard to generate valid words which of course, is far from the case in a coherent text. This led Hartley [6] to formulate ideas based on information content which were then developed into a theory of information transmission by Shannon [19] as described in Cherry's unifying book, [3]. This will therefore be used as a basis here for a model of functionality in software systems.

Hartley [6] showed that a message of N signs chosen from an *alphabet* or code book of S signs has S^N possibilities and that the *quantity of information* is most reasonably defined as the logarithm of the number of possibilities. This will be extended here by considering a component of size n_i lines as being built from a programming language with a total alphabet S(n_i) signs for that component. In the context of a programming language, a sign is a symbol of the language potentially containing multiple characters and is sometimes called a token.

The number of ways of arranging the signs of this alphabet is therefore S^S. Following Hartley, the quantity of information will therefore be defined as

$$I_S(n_i) = log(S(n_i))^{S(n_i)} = S(n_i)logS(n_i) \tag{16}$$

Here, S has been allowed to be a function of n_i as it is reasonable to expect that the alphabet of available signs may depend on the size of the component being built, (since programming languages allow user-defined signs).

Defining the functionality f_i of the ith component to be the information content of this alphabet gives

$$f_i \equiv I_S = log(S(n_i))^{S(n_i)} = S(n_i) log S(n_i) \tag{17}$$

Combining equations (11) and (17) then gives

$$p_i = \frac{e^{-\beta \frac{S(n_i)}{n_i} log S(n_i)}}{Q(\beta)} \tag{18}$$

Studying the form of equation (18), it is obvious that power-law behaviour will emerge if and only if the alphabet $S(n_i)$ is proportional to the size n_i so that

$$S(n_i) \propto n_i \tag{19}$$

In which case, equation (18) then leads directly to a prediction that component sizes will be distributed as a power-law

$$p_i = \propto n_i^{-\delta} \tag{20}$$

where δ is another constant.

These steps are worth summarising.

> *In the development of a software system of fixed size and fixed functionality where functionality is identified directly with the Hartley/Shannon information content of the alphabet used to construct each component, the most likely distribution of medium to large component sizes in the developed system will be as a power-law if and only if the size of that alphabet is linear with the component's size in lines.*

Now it has previously been noted as discussed by [3], p. 50 for example, that the concept of information based on alphabets as extended by Shannon and Wiener amongst others, *only relates to the symbols themselves* and not their *meaning*. This is clearly not satisfactory for the current model as it does not distinguish between pounding aimlessly on a keyboard and the careful construction of a computer program to specific intent.

To circumvent this criticism, the following alphabet will be explored.

> *The functionality of a component is defined to be the Hartley/Shannon information content of the alphabet S', of* ***non-redundant user-defined variable names used to construct that component****. (A non-redundant variable name is one which is actually used in the program.)*

This has an intuitive appeal in that the very fact that the developer has used particular variable names associates meaning with them in a fundamental way. The fact that the built-in symbols of a programming language are also excluded, for example keywords and other tokens such as *if*, *while* and so on, also reduces the dependence on stylistic attributes of a language, (there are often numerous equivalent ways of constructing the same code fragment even in the same programming language as can be seen by these functionally identical implementations of an *if* statement in C).

1. `max_of_xy = (y>x) ? y : x;`

2.
   ```
   max_of_xy = x;
   if (y>x) max_of_xy = y;
   ```

3.
   ```
   if (y>x) {
      max_of_xy = y;
   } else {
      max_of_xy = x;
   }
   ```

Given then that choosing an alphabet of non-redundant user-defined variable names was intended to circumvent the original criticism of Miller's work that it concerned only the symbols and not their meaning, an acid test of whether such a choice is sensible or not is to see if such linear behaviour is indeed present in the real systems described here. If it is not, it would be inconsistent with the presence of power-law behaviour according to the predictions of the model developed here.

3.3 Evidence for linearity

First consider the same library as was used in Figure 2. The raw data is somewhat unprepossessing as can be seen by inspecting Figure 6 which shows the relationship between the number of non-redundant user-defined variable names and number of executable lines of code in each component of the NAG Fortran scientific subroutine library as described in [9]. (The

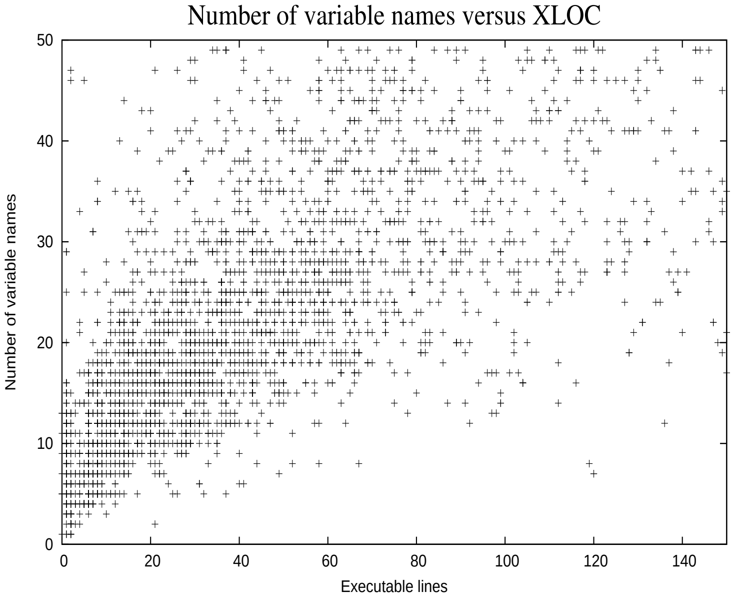

Figure 6: The raw data for the NAG Scientific Subroutine Library. Here the *non-redundant* variable name count for each component is plotted against the number of executable lines of code in that component.

largest 15% have been excluded as outliers as the bins defined there are not so well populated for large numbers of user-defined variable names.)

If the data of Figure 6 is averaged such that the number of non-redundant user-defined variables is gathered into bins in multiples of 5 variables and then plotted against the trimmed mean count of executable lines of code for that bin, then Figure 7 emerges. This shows that in this mature and very large library, the number of non-redundant user-defined variable names in a component is indeed *on average* closely linearly proportional to the count of executable lines of code, confirming the reasonableness of using non-redundant user-defined variable names as a measure of functionality in this system.

To widen the scope of this discussion, another system was analysed, this time for both redundant and non-redundant variable names. The rationale behind this is not only to see if the non-redundant names again follow the predicted linear pattern seen above, but also to see if the redundant names differ in behaviour, as would be hoped. If they do not, it would undermine the above argument that using non-redundant names as a measure of infor-

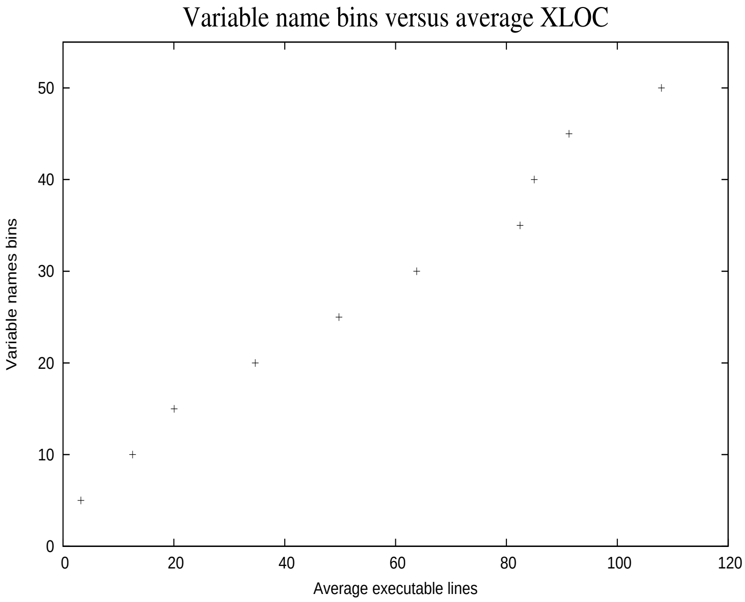

Figure 7: A markedly linear relationship emerges in the NAG Fortran Scientific Subroutine Library when the number of *non-redundant* user-defined variable names is binned in multiples of 5 with each bin plotted against the trimmed mean executable line of code count for that bin.

mation content overcomes the criticism that such content is describing only the symbols and not their meaning as described earlier.

This system is the one described earlier in Figure 4. It is written in a different language, C, and is in an entirely different application area, that of high-integrity language parsing. Analysing non-redundant variable names in C is a little more challenging than in Fortran due to the use of the #include construct of C and the common paradigm of defining variables for entire systems in such included files. To reduce the potential impact of this, only variables actually defined in the functions themselves were counted, (either as redundant if they were not used, or non-redundant if they were used).

Figure 8 shows the number of non-redundant variable names as a function of the trimmed mean of the number of lines of component in which they appear as before. Again, linearity is more than hinted at although there are insufficient points to make statistically significant statements, (only reasonably well populated bins are shown). In sharp contrast, when the redundant variable names are used, the data shows no obvious pattern as can be seen in Figure 9.

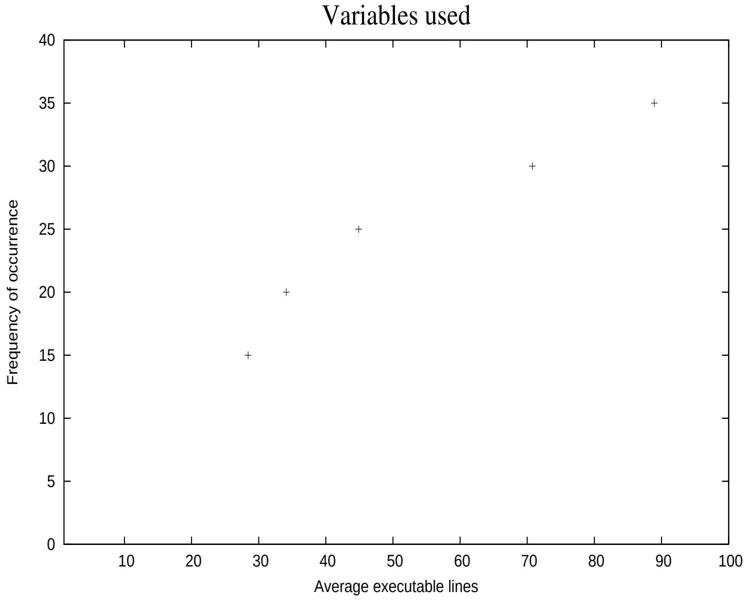

Figure 8: A relatively linear relationship emerges in this large C based system when the number of *non-redundant* user-defined variable names is binned in multiples of 5 with each bin plotted against the trimmed mean executable line of code count for that bin. Only bins with significant numbers of members are shown.

More analysis will be necessary to investigate this further but for the purposes of this paper, such a definition of functionality certainly looks reasonable in that it is both consistent with the information model used here and the constraints enforced upon it by the nature of the statistical mechanical argument.

4 Conclusions

It is appropriate to unite the conclusions from [8] and the current work as follows.

First of all, [8] was motivated by considering the role of defect first as it appears in an already developed system. It included examples of power-law behaviour being present in numerous released systems of different sizes, implementation languages and application areas and went on to show using an argument based on statistical mechanics, that the growth of defect with functional form xlogx where x is the number of XLOC, was intimately related

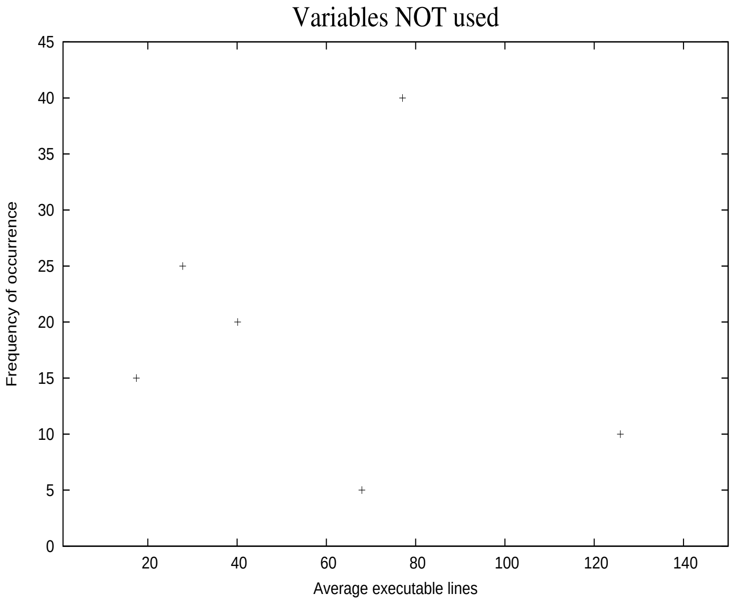

Figure 9: By contrast, a rather complex non-linear relationship emerges in this large C based system when the number of *redundant* user-defined variable names is binned in multiples of 5 with each bin plotted against the trimmed mean executable line of code count for that bin. Again only bins with significant numbers of members are shown.

with the appearance of power-law behaviour. Assumptions in the variational principles used limit this model to components greater than about 20 XLOC. It was not clear however which effect drives the other if any and left open a number of questions, the most important of which were:-

- Is power-law behaviour of component size a persistent property in that it is already present at the first release of a software system ?
- Is there an underlying mechanism for this ?

The current paper answers both of these questions at least in part. It answers the first by giving convincing evidence for power-law behaviour being persistent in three systems, entirely different in size, application area and programming language. It also shows that the power-law behaviour described here is not inconsistent with previous comparable studies such as that by Mitzenmacher [14].

It answers the second by demonstrating that a model of functionality based on the Hartley-Shannon information content of an alphabet of signs, coupled with a variational model similar to that used in [8], leads directly to the emergence of power-law behaviour for components greater than about 20 XLOC, *provided that the alphabet of signs used grows linearly with the component being built from those signs.*

Finally, it goes on to propose an intuitively reasonable alphabet based on non-redundant variable names which can be used with this model of functionality and demonstrates in two very disparate systems that the expected linear behaviour is indeed evident. It also shows that the use of a different alphabet based on redundant variable names does not lead to any obvious pattern. Taken together, this goes some way to justifying both the variational models used and their consistency with intuitively reasonable models of functionality.

The current paper and [8] therefore provide a tentative model for the development and release phases of a software system as follows:-

Development phase

> *In the development of a software system of fixed size and fixed functionality where functionality is identified directly with the Hartley/Shannon information content of the alphabet used to*

> *build each component, the most likely distribution of component sizes to emerge in the developed system will be as a power-law for components greater than about 20 XLOC if and only if the size of the alphabet is linearly proportional to the size of the component.*

It was demonstrated that an alphabet based on the non-redundant user-defined variable names of a component appears to have the right property of linearity.

Release phase When such a system is placed in the users' hands, it will contain a fixed (and generally unknown) number of defects. Then as shown by [8], with a matching variational principle and supporting empirical data,

> *After release, as a software system matures and gradually exhibits the defects inadvertently built in during its development, an underlying power-law distribution of component sizes will lead in a system of fixed size to a distribution in defect in each component of xlogx where x is the number of executable lines in that component. Again this is applicable to components greater than about 20 XLOC due to assumptions made in the variational model.*

This does not complete the story on information content in computer programs by any means as there may be other empirical measures of functionality which lead to the same power-law behaviour of component size, but this will be deferred for the moment to future study of these fascinating relationships. It should be noted that the current work makes no attempt to shed any light on why distributions of non-redundant variable names should be linear with the size of the component in which they appear. If a model for this emerged, it could be combined with the present work to show that such linearity directly drives both the power-law behaviour of component size and from that the xlogx defect behaviour.

Finally, it can be noted that the instincts of the long-suffering end user seem always to have been correct in associating bugs with features. The variational principles used in [8] and the current paper are entirely consistent if defects are linearly proportional to functionality as defined here. This can easily be seen by comparing the association of functionality density here, and the defect density in [8], with the entity ε_i in the unifying variational principle represented by equation (8).

With this view of information content in a software system therefore, defects are indistinguishable from functionality.

4.1 Threats

There are several threats to the developments presented here. Perhaps the most important ones, (which can be levelled at most attempts to obtain empirical evidence in software systems), are the lack of data and the noise inherent in such data as is available.

The first paper [8] demonstrated fairly convincingly the presence of power-law behaviour in the larger components of 21 separate systems of very different provenance. Furthermore, such behaviour is not inconsistent with other studies such as that by [14]. However in the current paper, only a handful of systems had sufficient source availability and control to search for deeper patterns. Having said that, these systems were very different indeed, in application area, size and programming language so their similarity in containing the sought patterns is at least encouraging. This is not enough but is a start in providing support for the powerful variational methods described in [8] and the current paper, which appear to underly software systems evolution independently of their implementation.

The data had to be considerably massaged. Trimmed means are a powerful and robust technique for dealing with outliers and potentially non-Gaussian distributions, but even then some of the bins were insufficiently populated to provide any reliable evidence. The only thing which can help this aspect is more good quality data.

5 Acknowledgements

The author would like to acknowledge useful discussions on power-law behaviour in general physical systems with his colleague Dr. Miro Novak and also the unknown reviewers of the first paper, who helped clarify the arguments in significant ways.

References

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. *OOPSLA '06*, 2006. http://doi.acm.org/10.1145/1167473.1167507.

[2] D. Challet and A. Lombardoni. Bug propagation and debugging in asymmetric software structures. *Physical Review E*, 70(046109), 2004.

[3] Colin Cherry. *On Human Communication.* John Wiley Science Editions, 1963. Library of Congress 56-9820.

[4] D. Clark and C. Green. An empirical study of list structures in lisp. *Communications of the ACM*, 20(2):78–87, 1977.

[5] G. Concas, M. Marchesi, S. Pinna, and N.Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Software Eng.*, 33(10):687–708, 2007.

[6] R.V.L. Hartley. Transmission of information. *Bell System Tech. Journal*, 7:535, 1928.

[7] L. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[8] L. Hatton. Power-law distributions of component sizes in general software systems. *IEEE Transactions on Software Engineering*, July/August 2009.

[9] T.R. 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.

[10] Barbara Kitchenham. Counterpoint: The problem with function points. *IEEE Software*, 14(2):29,31, 1997.

[11] Barbara Kitchenham, Shari Lawrence Pfleeger, Beth McColl, and Suzanne Eagan. An empirical study of maintenance and development estimation accuracy. *J. Syst. Softw.*, 64(1):57–77, 2002.

[12] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[13] G.A. Miller. Some effects of intermittent silence. *American Journal of Psychology*, 70:311–314, 1957.

[14] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–333, 2003.

[15] Christopher R. Myers. Software systems as complex networks: Structure, function and evolvability of software collaboration graphs. *Physical Review E*, 68(046116), 2003.

[16] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, 2006.

[17] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Comm. ACM.*, 48(5):99–103, May 2005.

[18] P.K. Rawlings, D. Reguera, and H. Reiss. Entropic basis of the pareto law. *Physica A*, 343:643–652, July 2004.

[19] C.E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27:379,423, July 1948.

[20] Meine J.P. van der Meulen. The effectiveness of software diversity. Ph.D. Thesis, City University, London, 2008.