Conservation of Information: Software's Hidden Clockwork ?

Les Hatton
Faculty of Science, Engineering and Computing, Kingston University, U.K.

Abstract—In this paper it is proposed that the Conservation of Hartley-Shannon Information (hereafter contracted to H-S Information) plays the same role in discrete systems as the Conservation of Energy does in physical systems. In particular, using a variational approach, it is shown that the symmetry of scale-invariance, power-laws and the Conservation of H-S Information are intimately related and lead to the prediction that the component sizes of any software system assembled from components made from discrete tokens always asymptote to a scale-free power-law distribution in the *unique* alphabet of tokens used to construct each component.

This is then validated to a very high degree of significance on some 100 million lines of software in seven different programming languages independently of how the software was produced, what it does, who produced it or what stage of maturity it has reached. A further implication of the theory presented here is that the average size of components depends only on their unique alphabet, independently of the package they appear in. This too is demonstrated on the main dataset and also on 24 additional Fortran 90 packages.

Keywords: Information conservation, Component size distribution, Power-law, software systems

1 Preliminaries

1.1 Statement of reproducibility

This paper adheres to the reproducibility principles espoused by [13] and includes references to all methods, source code and data necessary to reproduce the results presented. These are referred to here as the *reproducibility deliverables* and are available at http://leshatton.org/.

1.2 Conservation of Energy

The Conservation of Energy is one of a few principles which are at the very heart of all physical systems. The principle has been modified over the years notably to take account of the 4-vectors of relativity and mass-equivalence but it remains pivotal. In 1915, Emmy Noether proved a remarkable theorem, [20], which shows that any differentiable symmetry of the action of a physical system has a corresponding conservation law. For example, the principle of Conservation of Energy is a consequence of general invariance of systems with a Lagrangian under time translations; Conservation of Linear Momentum is a consequence of invariance under translation in space, and Conservation of Angular Momentum is a consequence of invariance under rotation.

The study of discrete systems is much younger and has only come of age in the digital era where we now routinely write millions of lines of source code to analyse terabytes of digital data. It is of great interest to see if there are similarly fundamental principles which apply to the evolution of discrete systems. This work extends that presented in [11], [12] to show that there is indeed an intimate relationship between symmetry, power-laws and conservation principles in discrete systems and verifies it to much higher levels of significance on an experimental dataset of around 100 hundred million lines of code incorporating some 500 million tokens. It is then shown that this behaviour leads to further interesting predictions which are themselves verified by experiment.

1.3 Power-laws

Power-law behaviour can be represented by the pdf (probability density function) p(s) of entities of a certain size s appearing in some process, being given by a relationship like:-

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

where k, γ are constants, which on a log p - log s scale is a straight line with negative slope $-\gamma$. It can easily be verified that the equivalent cdf (cumulative density function) c(s) derived by integrating (1), also obeys a power-law, (for $\gamma \neq 1$). For noisy data, the cdf form is most often used because of its fundamental property of reducing noise, as noted by [19] and it is this form which will primarily be used for significance testing.

Power-law behaviour has been studied in a very wide variety of environments, see for example [30] (linguistics), [23] (economic systems) and the excellent reviews by [16] and [19]. In software systems there has been significant activity, much of it recent, [5], [15], [18], [3], [8], [21], [2], [6], [14] and [11] all discussing power-law behaviour but in various contexts.

An early version of this paper was presented at 10th IFIP WG 2.5, Boulder, CO, 1-4 Aug. 2011. lesh@oakcomp.co.uk; leshatton.org

For example, Mitzenmacher [15] considers the distributions of file sizes in general filing systems and observed that such file sizes were typically distributed with a lognormal body and a Pareto (i.e. power-law) tail. Gorshenev and Pis'mak [8] studied the version control records of a number of open source systems with particular reference to the number of lines added and deleted at each revision cycle. Louridas et al [14] show that there is evidence that power laws appear in software at the class and function level and that distributions with long, fat tails in software are much more pervasive than previously established,

1.4 Systems of discrete choices

A system based on discrete choices is any system which is built from discrete pieces based on some available set of choices. Such choices will be referred to as an *alphabet*.The genome is a perfect example. This is an exceptionally complex system which has evolved over hundreds of millions of years, astonishingly from a set of only four choices, the four bases of the genetic alphabet, adenine, thymine, cytosine and guanine (ATCG). The human genome comprises some 3 billion such bases.

Computational science provides many more examples. In computational science, the source code of every computer program written by every programmer in pursuit of their computational results uses one or more programming languages.

The individual bases or alphabet of a programming language are called *tokens* and may take two forms; the *fixed* tokens of the language as provided by the language designers, and the *variable* tokens. Fixed tokens include (in the languages C and C++ for example) keywords such as *if*, *else*, *while*, {, }. These can not be changed, the programmer can only choose to use them or not. Variable tokens, with some small lexical restrictions (such as the common requirement for identifiers to begin with a letter), can be arbitrarily invented by the programmer whilst constructing their program. These might be names such as *numberOfCandidateCollisions* or *lengthOfGene* or constants such as 3.14159265. There are many programming languages but all obey the same principles and every form of software system evolves from such tokens.

It should be noted that classifying programs in terms of fixed and variable tokens is not new and appeared at least as early as 1977 in the influential work of Halstead who called them *operators* and *operands*, [9]. He developed his work to define various dependent concepts such as software *volume* and *effort* and tested them against programs of the time. This was further elaborated by Shooman in [28]. A different approach will be used here which borrows from the methods of variational calculus.

Computer programs are often very large. The software deployed in the search for the recently discovered Higg's boson comprises around 4 million lines of code [24]. At an average of around 5 tokens per line of code, this corresponds to some 20 million tokens, although this is still less than 1% of the human genome. The largest systems in use today appear to be around 100 million lines of source code [17], perhaps 15% of the number of tokens of the human genome. The (largely) open systems used to test the model described here total almost 100 million lines, (specifically 98,476,765 lines), totalling some 500 million tokens. (If 5 tokens per line seems a little low, it should be recalled that lines of code include comment lines here in line with common practice, whilst token counts do not.)

As an example of the nomenclature used here, consider the following simple sorting algorithm written in C, for example [25].

```
void bubble( int a[], int N )
{
  int i, j, t;
  for( i = N; i >= 1; i--)
  {
    for( j = 2; j <= i; j++)
    {
      if ( a[j-1] > a[j] )
      {
        t = a[j-1];a[j-1] = a[j];a[j] = t;
      }
    }
  }
}
```

This algorithm contains 94 tokens in all based on 18 of the fixed tokens of ISO C

```
void int ( ) [ ] { , ; for
= >= -- <= ++ if > -
```

and the 8 variable tokens (i.e. invented by the programmer)

```
bubble a N i j t 1 2
```

Although programming languages have a much richer alphabet of tokens than genes, they obey the same principles - some external process chooses tokens from the available alphabet. It will be argued here that this process is driven by a beautiful underlying clockwork, that of Conservation of H-S Information.

1.5 Information theory

Information theory has its roots in the work of Hartley [10] who showed that a message of N signs (i.e. tokens) 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 or choices. To gain a little insight into the reason why the logarithm makes sense, consider Figure 1. The number of choices necessary to reach any of the 16 possible targets is the number of levels which is log_2(number of possibilities). The base of the logarithm is not important here.

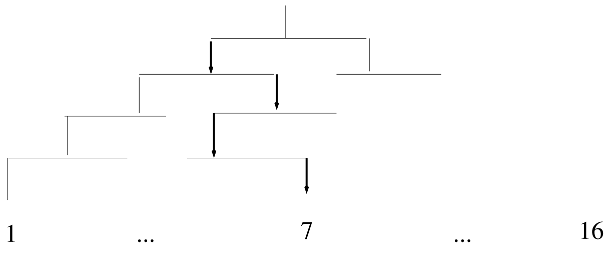

Fig. 1. A binary tree. Each level proceeding down can either go left or right. There are four levels leading down to one of 2^4 = 16 possibilities. Only 4 choices are needed to reach any of the possibilities. We note that $log_2(16) = 4$. Here the number 7 has been singled out by the choices left, right, left, right.

Information theory was developed very substantially by the pioneering work of Shannon [26], [27]. However it is important not to conflate information content with functionality or meaning and Cherry [4] specifically cautions against this noting 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*. Indeed, Hartley in his original work, defined *information* as the successive selection of signs, rejecting all meaning as a mere subjective factor. In the sense used here therefore, Conservation of H-S Information will be synonymous with Conservation of Choice, not meaning. This turns out to be enough to predict important system properties. In other words, those properties depend only on the alphabet and not on what combining tokens of the alphabet might mean in any human sense.

2 A VARIATIONAL MODEL OF A DISCRETE TOKENISED SYSTEM

Armed with these pieces of information, a variational model will be defined in which Conservation of H-S Information is a fundamental constraint following on from [12]. A general discrete tokenised system will be considered here as T tokens distributed in some way amongst M non-nested *components*, each containing t_i tokens where i = 1, ... ,M. In software systems, a component might be a subroutine (Fortran), function (C) or procedure (Tcl-Tk). In OO languages, it might be a method[1]. In a genetic system, a component might be a gene.

Before beginning the mathematical development, the whole process can be visualised by imagining a large number of people each independently carrying out the same project in parallel. They each must take the same large fixed number of beads, coloured so they can be distinguished, and distribute them over their own large set of boxes so that the total H-S information for their set of beads is conserved in a manner which is described below. The total number of beads must be the same for each person, (although the colour distribution need not be), and the total number of boxes must also be the same. There are no other rules. When they have all completed this task, an adjudicator measures the distribution of beads in each participant's boxes. If the numbers are large enough, the adjudicator will find that the vast majority of these distributions will obey a power-law in the unique count of different coloured beads in each set of boxes. This result is independent of the number of beads, participants or the number of boxes, providing they are large.

1. Strictly speaking methods can be and usually are nested in OO systems although when compiled they are simply treated as a function with some context and so remain relevant to this model of M non-nested components. Proper source analysis requires them to be treated the same way.

To prove this, the following variational methodology is borrowed from the world of statistical physics, [29] (p.217-) and for an excellent introduction, [7]. In the kinetic theory of gases, a standard application is to find the most common arrangement of molecules in a gas subject to various constraints such as a fixed total number of molecules and fixed total energy. In this development, molecules will be replaced by tokens and energy by an as yet undetermined quantity ε.

For this system, the total number of ways of organising the tokens is given by:-

$$W = \frac{T!}{t_1!t_2!..t_M!} \tag{2}$$

where

$$T = \sum_{i=1}^{M} t_i \tag{3}$$

and where there is some externally imposed entity ε_i associated with each token of component i whose total amount is given by

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

In a physical system, U corresponds to the total internal energy and the variational method to follow constrains this value to be fixed, i.e. solutions are sought in which energy is conserved.

Using the method of Lagrangian multipliers and Stirling's approximation as described in [11], the most likely distribution satisfying equation (2) subject to the constraints in equations (3) and (4) will be found. This is equivalent to maximising the following variational derived by taking the log of (2). Just as in maximum likelihood theory, taking the log dramatically simplifies the proceedings, in this case the factorials and allows the use of Stirling's theorem for large numbers and since it is monotonic, a maximum in log W is coincident with a maximum in W. This leads to

$$logW = TlogT - \sum_{i=1}^{M} t_i log(t_i) + \lambda\{T - \sum_{i=1}^{M} t_i\} + \beta\{U - \sum_{i=1}^{M} t_i\varepsilon_i\} \quad (5)$$

where λ and β are the multipliers and log is the natural logarithm. In essence, the variational process envisages varying the contents t_i of each of the components until a maximum of log W is found. This is indicated by taking $\delta(logW) = 0$, (analogous to finding maxima in differential calculus). Noting that the variational operator δ acts on pure constants such as TlogT, λT and βU to produce zero just as when differentiating a constant; $\delta(t_i log(t_i)) = \delta t_i log(t_i) + t_i \delta(log(t_i) = \delta t_i(1 + log(t_i))$; ε_i is independent of the variation by assumption and that T and the t_i are $\gg 1$ (to satisfy Stirling's theorem), leads to

$$0 = -\sum_{i=1}^{M} \delta t_i\{log(t_i) + \alpha + \beta\varepsilon_i\} \quad (6)$$

where $\alpha = 1 + \lambda$. (Further elaboration of this standard technique can be found in Glazer and Wark [7].)

Finally, (6) must be true for all variations to the occupancies δt_i and therefore implies

$$log(t_i) = -\alpha - \beta\varepsilon_i \quad (7)$$

Using equation (3) to replace α, this can be manipulated into the most likely, i.e. the equilibrium distribution of tokens amongst the M components.

$$t_i = \frac{Te^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \quad (8)$$

Defining $p_i = \frac{t_i}{T}$, equation (8) then yields

$$p_i = \frac{e^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \quad (9)$$

Following [23] and referring to equation (4), p_i can then be interpreted as a pdf and specifically as the probability that a component of size t_i tokens is found is exponentially related to ε_i. The larger ε_i for example, the less likely such a component is to appear. This interpretation will be clarified shortly.

Hinting at what is to come, *we can see immediately that in any discrete system in which (3) and (4) are conserved, and $\varepsilon_i = log\phi_i$, is the logarithm of some quantity ϕ_i, then the resulting size distribution is overwhelmingly likely to be power-law in ϕ_i since $exp(-\beta log\phi_i) = \phi_i^{-\beta}$. Furthermore if ϕ_i is external in the sense of not being dependent on t_i, then relationship (9) is scale invariant and links conservation quantities and scale-invariance with a power-law.*

2.1 Merging with information theory

Extending [12], suppose now that the *unique* alphabet of the i^{th} component contains a_i unique tokens and as defined above, t_i tokens in all. The number of ways of arranging the tokens of this alphabet in component i is therefore $a_i^{t_i}$. Following Hartley, the quantity of information in component i will therefore be defined as

$$I_i = log(a_i)^{t_i} = t_i log a_i \quad (10)$$

The total amount of information I, is then given by

$$I = \sum_{i=1}^{M} I_i = \sum_{i=1}^{M} t_i log a_i \quad (11)$$

Considering information as an intrinsic property and identifying U with I, it is immediately obvious that (11) is identical with (4) if the unique alphabet a_i is identified with the function ϕ_i. Note that this emphasises the intimate relationship between variational solutions which conserve energy (U) in a physical system and variational solutions which conserve H-S information (I).

This identification also satisfies the requirements of scale-invariance if we consider a_i and t_i as independent. This allows (9) to be written as

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

where $Q(\beta)$ is just a scale factor which guarantees that p_i is a pdf and is given by

$$Q(\beta) = \sum_{i=1}^{M} e^{-\beta log a_i} = \sum_{i=1}^{M} (a_i)^{-\beta} \quad (13)$$

Summarising, subject to the twin constraints that the total number of tokens T is fixed

$$T = \sum_{i=1}^{M} t_i \quad (14)$$

and the total Hartley / Shannon information content I is also fixed

$$I = \sum_{i=1}^{M} I_i \quad (15)$$

, then it is overwhelming likely that the distribution of component sizes will be a power-law obeying the scale-free pdf

$$p_i = \frac{(a_i)^{-\beta}}{Q(\beta)} \sim (a_i)^{-\beta} \quad (16)$$

With regard to clarifying the interpretation of p_i as a pdf, we can derive two simple but testable results in the form of a marginal and a conditional density function. To do this we will temporarily use continuous distribution theory to clarify the development.

2.1.1 Marginal distribution of the unique alphabet

Since each component consists of t_i tokens made up from a_i unique tokens, (16) will be treated as a joint distribution in the random variables $\overline{T}$ and $\overline{A}$, defined as follows:-

$$p(\overline{T} = t_i, \overline{A} = a_i) = \frac{1}{Q(\beta)}(a_i)^{-\beta}; t_i \geq a_i \quad (17)$$

$$= 0; t_i < a_i \quad (18)$$

since the probability of finding a component with less total tokens than its unique alphabet is clearly zero.

The marginal probability $p_{\overline{A}}(a_i)$ is then defined by

$$p_{\overline{A}}(a_i) \equiv \int_{a_i}^{T} p(\overline{T} = t_i, \overline{A} = a_i) dt_i = \int_{a_i}^{T} \frac{1}{Q(\beta)}(a_i)^{-\beta} dt_i \quad (19)$$

where the support of $\overline{T}$ is $[a_i, T]$ since the joint probability is zero for $t_i < a_i$, and no component can be greater than the total size of all components. From this,

$$p_{\overline{A}}(a_i) = \frac{(T - a_i)}{Q(\beta)}(a_i)^{-\beta} \approx \frac{T}{Q(\beta)}(a_i)^{-\beta} \quad (20)$$

since $T \gg a_i$.

This can be interpreted as implying that the marginal probability distribution, $p_{\overline{A}}$, for the distribution of the a_i over all t_i asymptotes to a scale-free power law in the unique alphabet a_i.

2.1.2 Conditional distribution of component sizes for fixed alphabet

By definition the conditional probability distribution of the component sizes, for a given alphabet is

$$p_{\overline{T}|\overline{A}}(\overline{T}|\overline{A} = a_i) \equiv \frac{p(\overline{T} = t_i, \overline{A} = a_i)}{p_{\overline{A}}(a_i)} \quad (21)$$

giving

$$p_{\overline{T}|\overline{A}}(\overline{T}|\overline{A} = a_i) = \frac{1}{Q(\beta)}(a_i)^{-\beta} \frac{Q(\beta)}{(T - a_i)(a_i)^{-\beta}} \sim k \quad (22)$$

where k is a constant, for $T \gg a_i$.

This can be interpreted as implying that the conditional probability distribution, $p_{\overline{T}|\overline{A}}$, for the size of a component given a fixed alphabet is asymptotically uniform within its defined support. Finally we note that the original variational method used to derive (16), [11] requires $T, t_i \gg 1$. The support of the marginal distribution in (19) therefore also requires that $a_i \gg 1$, so we will test these predictions in the regime $a_i \geq 10$.

Summarising, (16) then unifies the three concepts of power-law, the symmetry of scale-invariance and a conservation principle. This is not quite the analogue of Noether's theorem for physical systems because, that is exact and as far as is known, Conservation of Energy, and others such as linear momentum (corresponding to displacement symmetry) and angular momentum (corresponding to rotational symmetry) are always obeyed in all systems. For discrete systems however, a pdf is produced which is overwhelmingly likely to be obeyed on average although it might not be in specific instances. As will be seen, this is more than enough to manifest itself emphatically in real systems.

It is worth repeating that this overall process *does not care about the tokens themselves* - all individual microstates are equally likely in this variational method. It simply says that if total size and choice in the Hartley-Shannon sense are conserved during the process of distributing the tokens, then power-law distribution of component size in the unique alphabet of tokens used is overwhelmingly likely to emerge since it occupies the vast majority of the microstates at any scale. In short, as programmers assemble a software system from the constituent tokens, no matter how or why they choose them, they are overwhelmingly likely to finish up with a system which obeys (16), and therefore (20) and (22).

Some other comments are useful.

1) The variational method assumes that t_i, a_i and T are $\gg 1$. This turns out to be a very good approximation for nearly all the data here. Components with $a_i < 10$ are relatively rare in software systems because of the token overhead described shortly.
2) The variational method *enforces that the total size T and total H-S information I are kept constant* whilst the most likely solution is found. It should be noted that these are not the actual size and information at any point in time, but their eventual values defined by their intended functionality in an ergodic sense. In other words, if the same system was produced many times independently, then for a particular (T,I), the most likely distribution would be given by (16).
3) Systems should *evolve* to preserve (16) since if a system represented by (T,I) is modified and becomes (T',I'), it will still inhabit a landscape in which (16) is overwhelmingly likely to be true, since it is true at any scale. This will also be demonstrated shortly on real data.
4) It is the very fact that the overall process does not care about token meaning which leads to the ubiquity of the behaviour reported here.

3 Application to Software Systems

3.1 Software components and tokens

Following the earlier discussion of the sorting algorithm, we can write the unique alphabet of a_i tokens in the i^{th} component of a software system of M components as

$$a_i = a_f + a_v(i) \quad (23)$$

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

It will be noted that a_f is taken as independent of component whereas $a_v(i)$ is dependent on the component. To flesh this out a little, it is worthwhile introducing a highly relevant property of programming languages at this point, borne out by the data. *Smaller components tend to be dominated by tokens fixed by the programming language and larger components tend to be dominated by tokens invented by the programmer, for example constants and identifier names.* The reasons for this are first, the fixed tokens of a language are limited in number and a significant number of these are very rarely used, (for example, the 10 *trigraphs* or the *goto* in ISO C). Second, there is a certain token overhead which must be paid in order to produce the simplest of syntactically-viable components. As the component size grows, it is observed that the fixed token alphabet rapidly stabilises whilst the invented token alphabet grows without any such limits. It is therefore a reasonable assumption to consider the alphabet of fixed tokens as approximately constant across components.

To support this conclusion, throughout these studies the (variable/fixed) token ratio was found to be typically 0.4 or less for the small components (the value is 8/18 = 0.44 for the sorting algorithm described earlier) and typically greater than 5 for large components. In addition, on average, the fixed token population does not vary significantly with component size - linear regression of a_f against t_i on the components extracted in this study revealed a gradient of around 7.0×10^{-4}, in other words it is effectively zero.

In other words, as the component size grows, the fixed token alphabet is relatively constant in this dataset whilst the variable token alphabet grows without any such limits. It was also observed here that more than 95% of the components analysed used less than 30 fixed tokens.

This affects the predicted shape of the distribution as will be seen.

3.2 Predicted shape of the size distribution

In anticipation of applying this to software data, as well as the pdf, it is conventional to consider the cdf (cumulative distribution function) as used by [19] because of its much more stable behaviour in the presence of noise. This is defined as the probability c(x) that x has a greater value than some specified value a_i and is given by integration of (20) as

$$c_i(a_i) \sim a_i^{-\beta+1} \tag{24}$$

for $\beta \neq 1$. It is then possible to anticipate the approximate predicted shape of the size distribution as follows. For smaller components, we have seen that it is reasonable to assume that the number of fixed tokens will tend to dominate the total number of tokens because

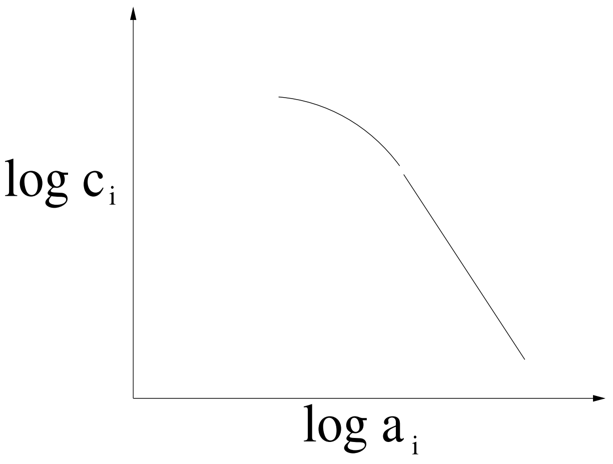

Fig. 2. The predicted cdf using the model described in this paper. The cdf is predicted to flatten for smaller components and to be power-law in a_i for large ones. Below $a_i \leq 10$, the approximations used break down.

of the fixed token overhead. In other words, $a_f \gg a_v(i)$. Equation (24) can then be written

$$c_i \sim (a_f)^{-\beta+1}(1 + \frac{a_v(i)}{a_f})^{-\beta+1} \tag{25}$$

In other words,

$$c_i \approx (a_f)^{-\beta+1} \tag{26}$$

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, the general power-law rule applies

$$c_i \sim (a_i)^{-\beta+1} \tag{27}$$

The generic shape of the resulting predicted curve on a log-log scale is shown in Figure 2.

In practice, this argument must be restricted to $a_i \geq 10$ or so to remain within the approximations used in the variational method ($T, t_i \gg 1$) and in the development of the marginal pdf over the support of $t_i \geq a_i$, (20).

3.3 Experimental verification

Such a profoundly simple but over-arching principle as (16) and its testable forms (20) and (22) invites validation to the highest possible degree. To cater for this, an unusually large number of systems were analysed across multiple languages in order to increase the statistical relevance. Open source has had many benefits but one of particular value to researchers is the enormous amount of source code which can be freely downloaded, often with excellent development history. In this study, 7 languages were chosen, Ada, C, C++, Fortran 77/90 (these could reasonably be treated as two languages given their fundamental differences but are counted as one here), Java, Matlab and Tcl-Tk. This covered a very wide variety of implementation areas and paradigms. In these languages, around 115 'packages' (package and system are essentially synonymous in this context) were downloaded by mining git archives, comprising just under 100 MSLOC (million source lines of code) over a bewildering number of development areas and around half a billion language tokens in all. These include for example, the whole of the Linux kernel, openBSD, PHP, X11R7, Postgresql, MySQL, R, the GNU distribution (counted as one package but containing 1427 sub-packages of very diverse nature) and Perl, all in C, the Ada validation system (Ada), the KDE desktop (C++ also counted as one package, but containing 311 sub-packages), and the Java Virtual machine (Java).

As well as these, the author had access to some commercial systems in Tcl, Fortran and C but these only totalled around 1% of the total code analysed. Individual package sizes spanned 4 orders of magnitude varying from around 3 KSLOC (thousands of source lines of code) to some 24 MSLOC (millions of source lines of code) in the GNU distribution. It is believed that this comprehensively samples the astonishingly varied nature of software development. Parenthetically, since the packages were extracted at random, it illustrates the continuing popularity of C which accounts for 85% of all the code analysed with Java 7%, C++ 3%, Ada 2% and the remaining languages measured, 3% between them.

3.3.1 Lexical analysis

The extraction of tokens from a program is not a trivial process and requires the development of tools which mimic the front-end of compilers [1]. The minimum requirements for a lexical analyser for each language considered here, were

- The ability to extract tokens and to distinguish between the two token forms, fixed and variable.
- The ability to recognise the start and end of a component. This is simpler for non-OO languages than OO languages because the latter admit nested components or methods. In this analysis a useful approximation is that nested methods can be ignored. This approximation is supported by the data.

The resulting generic tokeniser was written in C for optimal performance to handle the seven languages considered and also to exploit the well-known *lex* tool for generating lexical analysers[2]. It comprises around 2000 lines of C and 1300 lines of lex and is included with the reproducibility deliverables mentioned earlier. There are certain difficult tokens in some languages which are simply ignored by this generic analyser. This excluded only a tiny fraction of components from the analysis however. As a quality control check, the C and Fortran tokenisers were checked against and found to agree closely with existing full parsers written by the author some years ago, both of which parsed the relevant compiler validation suites correctly, (FIPS160 in the case of ISO C90 and the ACVS in case of Fortran 77). In

2. , http://flex.sourceforge.net/manual/

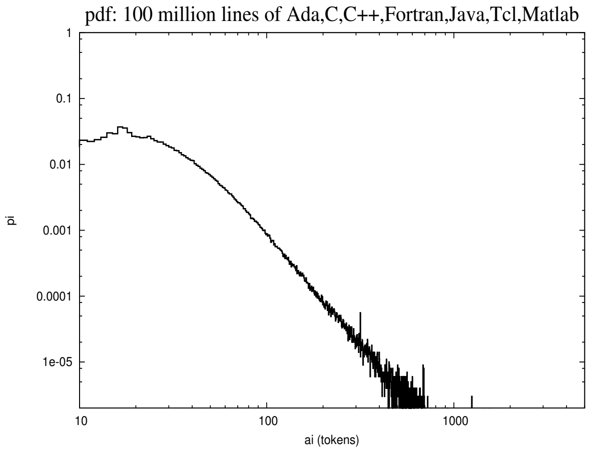

Fig. 3. The measured pdf for all the systems analysed here combined together.

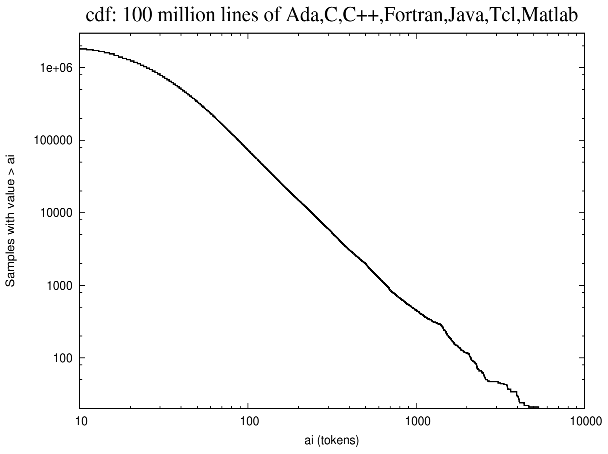

Fig. 4. The measured cdf for all the systems analysed here combined together.

addition, a small number of test programs were checked by hand. The resulting generic tokeniser is extremely fast and can extract tokens at around the rate of 100,000 lines a second on a typical Linux desktop allowing the analysis of the very large amounts of source code considered here.

3.3.2 Results

Since the predicted marginal distribution as tested (20) is scale-invariant and independent of any meaning, combining all the data together into a single half billion token super-system satisfactorily simulates the ergodic nature of the prediction. The resulting pdf and cdf are shown as Figures 3 and 4 respectively.

Figure 3 clearly shows qualitative agreement with the predicted power-law behaviour in the larger values of a_i tending to flatten out at the lower values. It also demonstrates the particular value of computing the cdf as suggested by [19] in reducing the noise as can be seen in Figure 4. This latter offers very satisfying support for the model proposed here and will be used to measure the slope in the power-law tail.

The predicted linearity of the power-law tail of Figure 4 was subjected to a standard test for significance using the linear modelling function lm() in the widely-used R statistical package [22]. This reported a very high degree of linearity with a linear-fit correlation of 0.998 between unique alphabet counts of 30 and 3000, a span of two decades. The same analysis reports a slope of -2.164 +/- 0.003, which is in the range -2 $\rightarrow$ -3 reported for most natural phenomena by [19], (corresponding to β = -3.164 in (20)). The associated p-value, (the probability of finding a dataset more unlikely than this one by chance) is $< 2.2 \times e^{-16}$, an extremely emphatic result. The corresponding output from R is shown in Table 1.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-0.1411	-0.0619	-0.0240	0.0504	0.4459	0.3033

Slope	Std. error	Adj R^2	F	DF	p
-2.164	0.003	0.998	$(5.55) \times e^5$	1107	$< (2.2) \times e^{-16}$

TABLE 1
Residuals (row 1) and fit statistics (row 2) on Figure 4 between a_i = 30 and 3000.

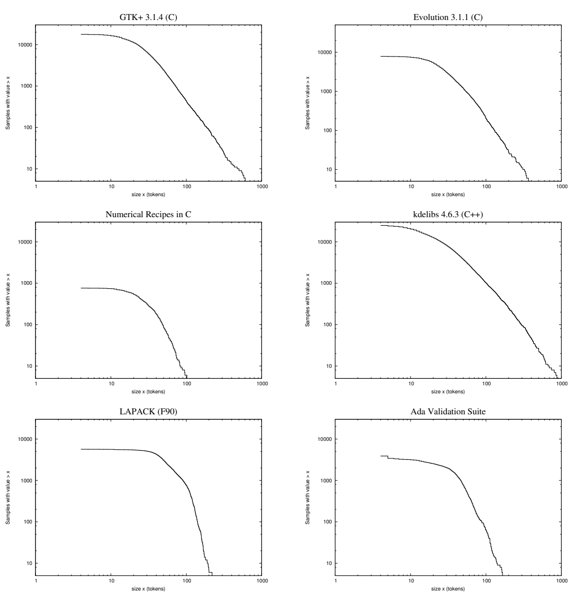

Fig. 5. cdfs of individual packages showing the broad adherence to (16) across language and size. Top row, GTK (C - large), evolution (C - medium); second row, Numerical Recipes (C - small), KDE library (C++ - large); Bottom row, LAPACK (Fortran - medium), Ada Validation (Ada - large).

It can be concluded that this experiment very strongly supports (20). The resulting behaviour implicit in Figure 4 contrasts nicely with the pure straight line predicted for monkeys pounding on keyboards as described by [16]. The ergodic nature of (20) simply accumulates all possible programmers pounding on keyboards, but with constraints on which keys can be used set by the relevant programming language. As can be seen by studying the animation included with the reproducibility deliverables, the generic shape of (20) appears early on, certainly within the first 1% of the total data represented by Figure 4.

3.3.3 Individual packages and persistence

The scale invariant nature of (20) also works well with individual packages as would be hoped providing token counts are large enough for the requirements of the variational method to be satisfied. To exemplify, Figure 5 shows a collage of the cdfs of some individual packages. This includes small (8 - 90 KSLOC) , medium (150 - 400 KSLOC) and large (0.5 - 1.0 MSLOC) C packages and medium and large C++, Fortran and Ada packages. Each of these packages clearly shows the nascent signature of the behaviour described by (20).

The behaviour is also persistent within a single package as is shown in Figure 6 for a package which doubled in size in its life-cycle over several years from its first release. Persistence is again expected from the scale-invariant nature of (20) which is preserved as a system moves from one power-law dominated landscape (T,I) to another (T′,I′) during maintenance.

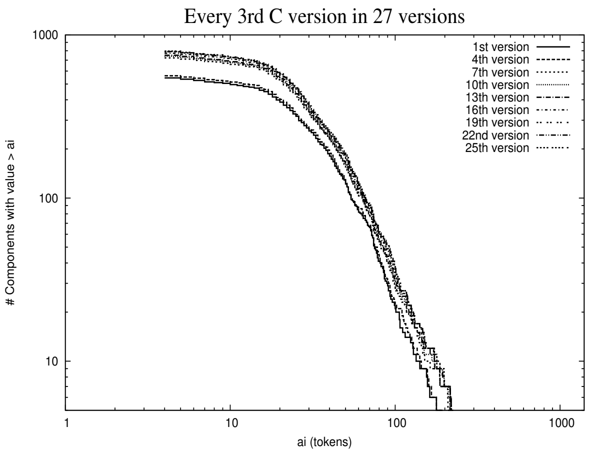

Fig. 6. Although difficult to see in monochrome, power-law behaviour is persistent. The first version of this system is on the inside of the family of curves and then versions appear monotonically out to the last version shown on the outside.

3.3.4 Constant average component length

So far, (20) has been derived and then demonstrated emphatically on a very large dataset. However, the *unique* alphabet a_i is not easily visible to a casual observer as it requires relatively sophisticated parsing of a language, (and is the reason for the use of the word 'hidden' in the title). The *total* number of tokens is much more accessible as it is an unequivocal measure of the overall size of a component and is very closely related to other commonly used measures such as the line of code. It

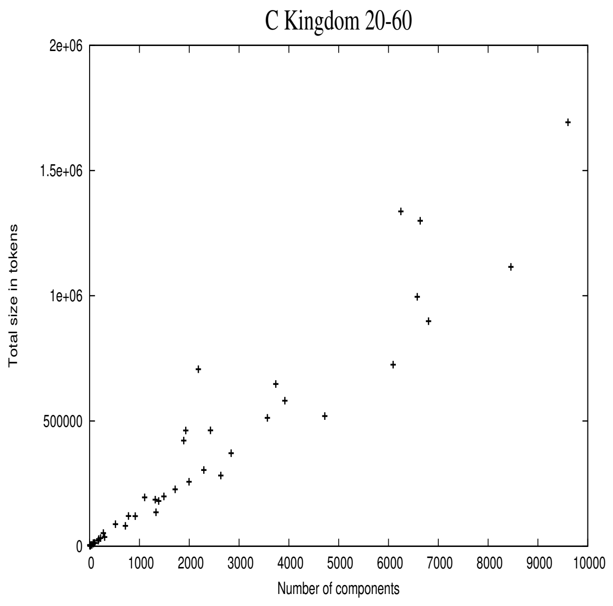

Fig. 7. Illustrating the predicted linearity of number of components (x-axis) against total number of tokens (y-axis) for $a_i \in [20, 60]$ for 51 C packages totalling some 50 MSLOC.

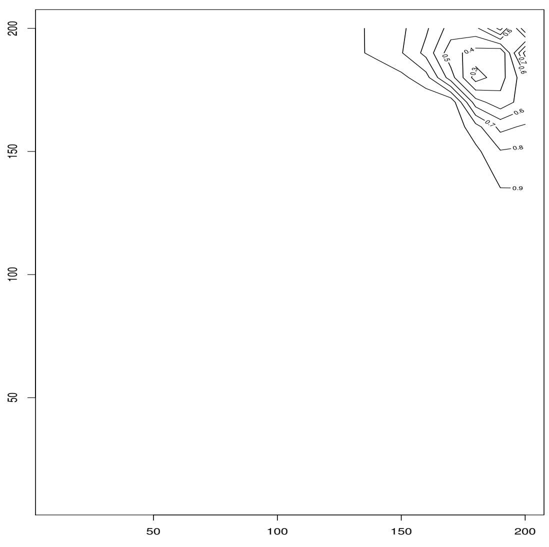

Fig. 8. A contour plot of the adjusted R-squared fit value produced by R as a function of the lower (x-axis) and upper (y-axis) values of the unique alphabet a_i. Figure 7 corresponds to the data point (20,60). The plot is dominated by a white area where R-squared > 0.99

is useful therefore to test (22) which predicts that total component lengths are uniformly distributed for a fixed unique alphabet. In other words, *components of any length with that unique alphabet are equally likely.*. This in turn implies that the average length of a component with that unique alphabet will be constant so *the total number of tokens should be linear with the total number of components with that alphabet for any package.*

This is far from an obvious inference but again it can be tested. 51 C packages totalling some 50 million lines of C code from the main experiment were subjected to the following repeated analysis. For a specified range, $a_i \in [a_{min}, a_{max}]$, all components were extracted for each package and the total number of components and the corresponding total number of tokens for each package calculated as a single data point. As an example Figure 7 shows all 51 data points calculated for the range $a_i \in [20, 60]$. Each data point corresponds to a package, for example the Apache package appears as the data point (6248,1336715).

As predicted, Figure 7 exhibits strong linearity. This was again tested using the lm() function of the R package and the results shown as Table 2.

Min.	1st Qu.	Median	3rd Qu.	Max.
-426964	-122636	-104656	59782	1205887

Slope	Adj R^2	F	DF	p
123.900 +/- 0.089	0.9974	$1.91 \times e^4$	49	$< (2.2) \times e^{-16}$

TABLE 2
Residuals (row 1) and fit statistics (row 2) on Figure 7.

This calculation was then extended out for values of $a_{min} = 10, .., 200, a_{max} = a_{min}, .., 200$ in increments of 10 and the adjusted R-squared value for the lm() linear fit function of R extracted for each (a_{min}, a_{max}) pair. The range $[10, 200]$ was chosen as being representative of the low-noise part of Figure 3 whilst incorporating both the flatter part and the power-law tail - it must be stressed that the linear behaviour for average component size predicted here is present whatever the actual values of a_i.

The result is shown in contoured form as Figure 8. Note that since the lower bound must be less than the upper bound, the half of the diagram below the diagonal is filled in by symmetry for convenience.

Figure 8 shows that over the vast majority of the ranges of a_i computed on a dataset of some 250 million tokens, that the adjusted R-squared value is exceptionally close to 1. This very strongly supports the prediction above that component size for a fixed range of a_i is distributed uniformly and that therefore the average component size only depends on the range of a_i and is completely independent of the package, what it does, or how big it is.

As a further test, 24 additional Fortran 90 packages were downloaded and analysed which did not form part of the main study. These included several CALGO[3] algorithms. Extracting the total number of components and total number of tokens for each package gave the plot shown as Figure 9. Again, the result demonstrates the expected approximate linearity and is shown in Table

3. http://calgo.acm.org/

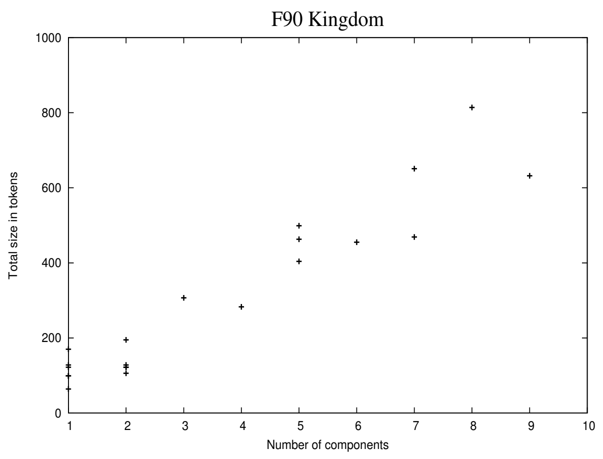

Fig. 9. The total length in tokens against the number of components in a number of software packages all written in the Fortran 90 programming language. The x-axis represents the number of small components in various packages and the y-axis is the total size of those small components in the package in tokens.

3.

Min.	1st Qu.	Median	3rd Qu.	Max.
-226.46	-49.84	-12.44	40.83	422.84
Slope	Adj R^2	F	DF	p
37.735 +/- 1.868	0.947	408.3	22	$< (1.076) \times e^{-15}$

TABLE 3
Residuals (row 1) and fit statistics (row 2) on Figure 9.

Even though there are less than half the data points of Figure 7, the fit is still a good one with adjusted R-squared value of 0.95.

4 Conclusions

This paper presents several contributions:

- Extending the argument suggested in [11], [12] to incorporate the principles of scale-invariance and the Conservation of H-S Information, it is confirmed that the probability p_i of a component appearing with t_i tokens in any software system, whatever its implementation details, should asymptote to the following scale-invariant marginal distribution with respect to the size of its unique alphabet of tokens a_i,
$$p_{\overline{A}}(a_i) \sim (a_i)^{-\beta} \qquad (28)$$
- Experimental evidence for this is greatly extended to cover almost 100 million lines of source code written in seven programming languages. The associated p-value matching the power-law tail to the prediction is $< (2.2) \times e^{-16}$ over two decades, with an adjusted R-squared value of 0.998, a very emphatic result indeed. The resulting value of β is -3.164 ± 0.003.
- The variational method now makes clear the intimate relationship between conservation principles, power-laws and the symmetry of scale-free behaviour in discrete systems. It also suggests strongly that the discrete system analogue of Conservation of Energy in physical systems is Conservation of H-S Information through the direct association of U with I. The analogy is not exact however. In physical systems, as [20] showed *inter alia* in her ground-breaking paper, energy is conserved in *all* systems with a Lagrangian which have the symmetry of time invariance. In the case of H-S information, the relationship between the conservation principle (H-S information) and the symmetry, (scale invariance), is in the form of a power-law probability distribution. Systems can exist which flout it but averaging over many systems, the power-law behaviour in the unique alphabet emerges emphatically as seen here.
- Another implication of Conservation of H-S Information is that the average component size in a package depends only on the unique alphabet a_i and not on package size, package functionality or the implementation language. This was tested on a large subset of the C systems analysed in the main study and also 24 additional Fortran 90 packages not in the main study with excellent statistical support and adjusted R-squared values generally very close to 1.

The main impact of the general principle of Conservation of H-S Information or Choice presented here is that it provides a theoretical underpinning for observed properties which appear to be system independent and indeed overwhelmingly likely. That such properties even exist in a discipline such as software engineering which encompasses a bewildering array of programming languages, design methodologies and implementation tools is of itself very encouraging. In addition, it stresses the role that the unique alphabet of a component may play in exempting some of its properties from continual technological overturn. For example, it is hoped that this will extend to a better understanding of the role of defects as touched upon in [11].

These findings have interesting implications for other kinds of system in which Conservation of H-S Information appears to play a fundamental role such as in the genome which has a fixed alphabet of unique tokens, but this will necessarily be considered as the subject of a separate paper.

Finally, it will be re-iterated that the Conservation of H-S Information and in particular its independence of meaning appears to provide a vast scale-free underlying power-law clockwork which the component size distributions of software systems at least, adhere to very closely, independently of what they do, how big they are, how they were produced or what technology was deployed.

5 Acknowledgements

The author would like to thank his colleagues Michiel van Genuchten and particularly Diomidis Spinellis who in a parallel joint project illustrated just how easy it was to automate the mining of vast amounts of open source software which rapidly trebled the experimental dataset used to verify the theory presented here. The author would also like to thank the anonymous reviewers and the Assistant Editor for very useful feedback in making the mathematical development as self-evident as possible.

References

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[2] 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.

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

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

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

[6] 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.

[7] A.M. Glazer and J.S. Wark. *Statistical Mechanics. A survival guide*. OUP, 2001.

[8] A.A. Gorshenev and Yu. M. Pis'mak. Punctuated equilibrium in software evolution. *Physical Review E*, 70:067103–1,4, 2004.

[9] M. Halstead. *Elements of Software Science*. Elsevier, 1977. ISBN 0-07-707640-0.

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

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

[12] L. Hatton. Scientific computation and the scientific method: a tentative road map for convergence. In *IFIP / SIAM / NIST Working Conference on Uncertainty Quantification in Scientific Computing*, pages 123–38, August 2011.

[13] D.C. Ince, L. Hatton, and J. Graham-Cumming. The case for open program code. *Nature*, 482:485–488, February 2012. doi:10.1038/nature10836.

[14] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, October 2008.

[15] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(1):305–333, 2002.

[16] Michael Mitzenmacher. A brief history of generative models for power-law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003.

[17] J. Mössinger. Software in automotive systems. *IEEE Software*, 27(2):2–4, March/April 2010.

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

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

[20] E. Noether. Invariante variationsprobleme. *Nachr. D. KÃ¶nig. Gesellsch. D. Wiss. Zu GÃ¶ttingen, Math-phys. Klasse 1918*, pages 235–257, 1918.

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

[22] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2011. ISBN 3-900051-07-0.

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

[24] D. Rousseau. The Software behind the Higgs Search. *IEEE Software*, 29(5):p. 11–15, 2012.

[25] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

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

[27] C.E. Shannon. Communication in the presence of noise. *Proc. I. R. E.*, 37:10, 1949.

[28] M.L. Shooman. *Software Engineering*. McGraw-Hill, 2nd edition, 1985.

[29] A. Sommerfeld. *Thermodynamics and Statistical Mechanics*. Academic Press, 1956.

[30] G.K. Zipf. *Psycho-Biology of Languages*. Houghton-Miflin, 1935.