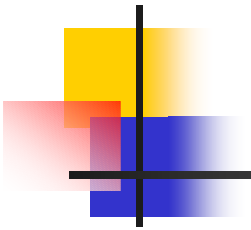
“The characteristic shape of software"

Les Hatton

Professor of Forensic Software Engineering
CISM, Kingston University
L.Hatton@kingston.ac.uk

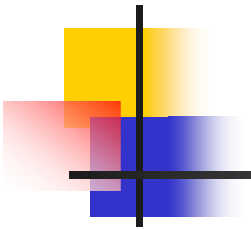
Version 1.1: 11/Nov/2010

Overview



- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

Building systems



- When we build a system we are making choices
  - Choices on functionality
  - Choices on architecture
  - Choices on programming language(s)

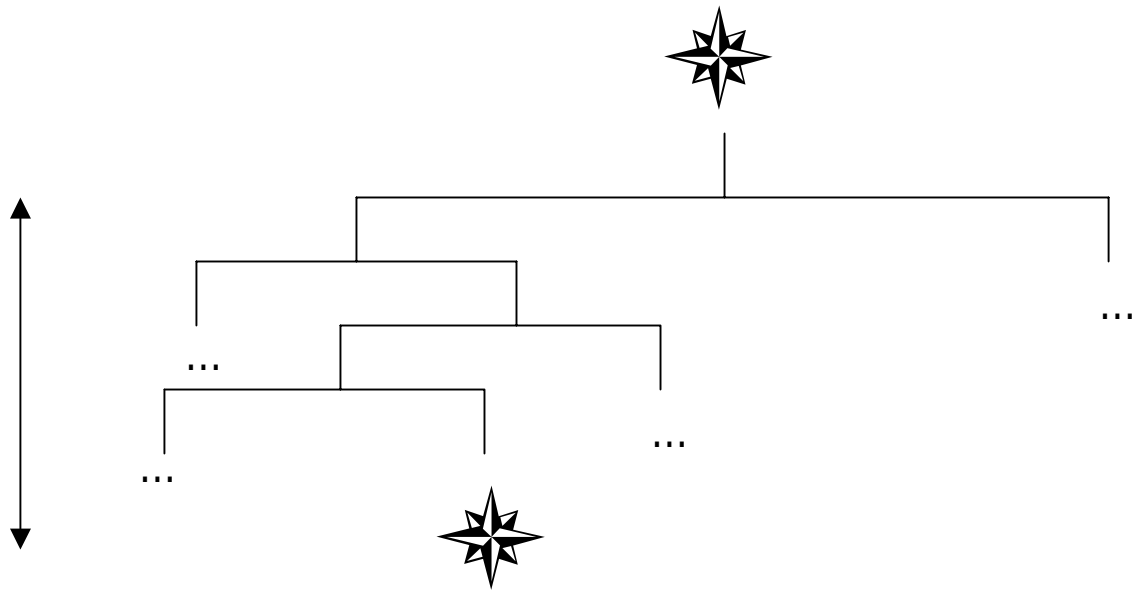
- There is a general theory of choice – Shannon information theory.

Building systems

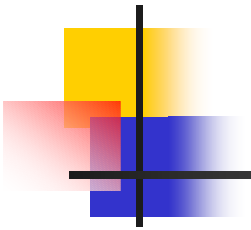
- Some notes on Shannon information theory
  - Based on assembling messages from non-divisible pieces, such as bits.
  - Suppose we have an N-bit stream of 1s and 0s. There are 2^N possible ways of arranging these.
  - Shannon was interested in representing the complexity or *information content* of these and realised following Hartley in 1928 that $\log_2 (2^N) = N$ was related to the number of choices need to find a particular combination

Building systems

Shannon information and decisions

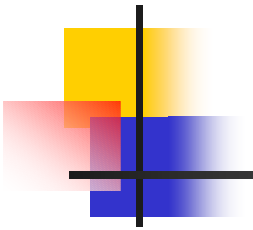


Building systems



- Software component size - approximate
  - *Number of lines of code*. This is quite dependent on the programming language, (consider the influence of the pre-processor in C and C++ for example).
- Software component size - better
  - Based on *tokens of a programming language*.

Building systems from tiny pieces

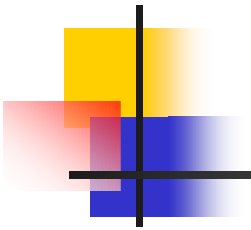


- Tokens of language
  - *Fixed tokens*. You have no choice in these. There are 49 operators and 32 keywords in ISO C90. Examples include the following in C, (but also in C++, PHP, Java, Perl …):

    { } [ ] ( ) if while * + *= == // / , ; :
  - *Variable tokens*. You can choose these. Examples include:-

    identifier names, constants, strings
- Every computer program is made up of combinations of these, (note also the Boehm-Jacopini theorem (1966)).

Overview



- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

What is scale-free behaviour ?

- In this context, scale-free behaviour refers to a phenomenon whose *frequency of occurrence* is given by a *power-law*.
- Consider word-counting in a document. If n is the total number of words in a document and n_i is the number of occurrences of word i, then ***it is observed*** (originally by Zipf (1949)), that for many texts,

$$f_i = \frac{c}{i^p}$$ where c, p are constants and $$f_i \equiv \frac{n_i}{n}$$

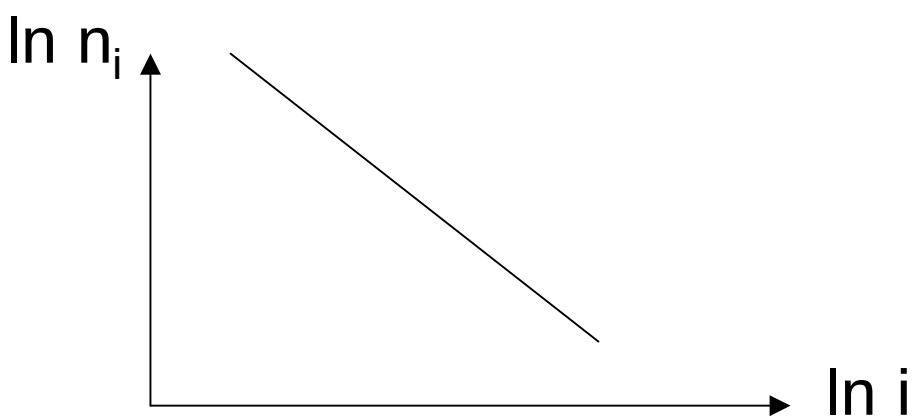
What is scale-free behaviour ?

Re-writing as $$n_i = \frac{nc}{i^p}$$

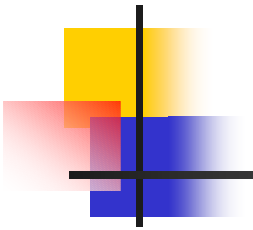
This is usually shown as

$$\ln n_i = \ln(nc) - p \ln i$$

which looks like

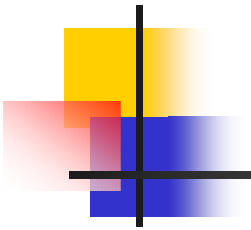


Examples from the real world



- Physics:- specific heat of spin glasses at low temperature, Caudron et al (1981)
- Biology: Protein family and fold occurrence in genomes, Qian et al. (2001)
- Biology: Evolutionary models, Fenser et al (2005)
- Economics: Income distributions, Rawlings et al (2004)
- Software systems: incoming and outgoing references and class sizes in OO systems, Potanin et al (2002)
- Many excellent examples in Newman (2006)
- Studies of C systems also reveal scale-free behaviour (Jones)
  - http://www.knosof.co.uk/cbook/cbook.html

Overview

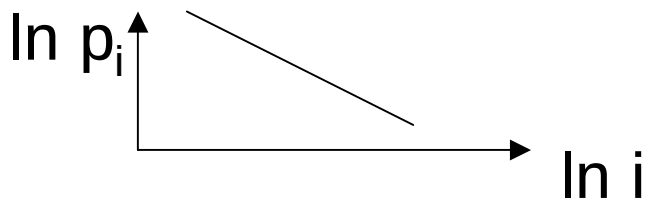


- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

Application to software systems

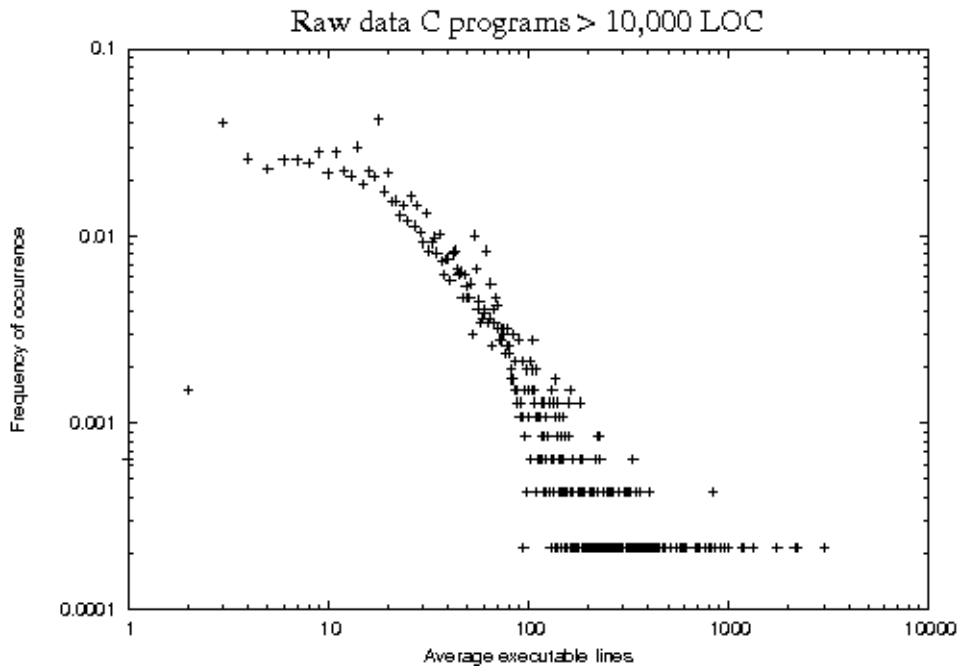
The following is a study of component sizes in 21 software systems in 3 different languages, Fortran, C and Tcl by Hatton (2009), IEEE TSE.

We are looking for .....



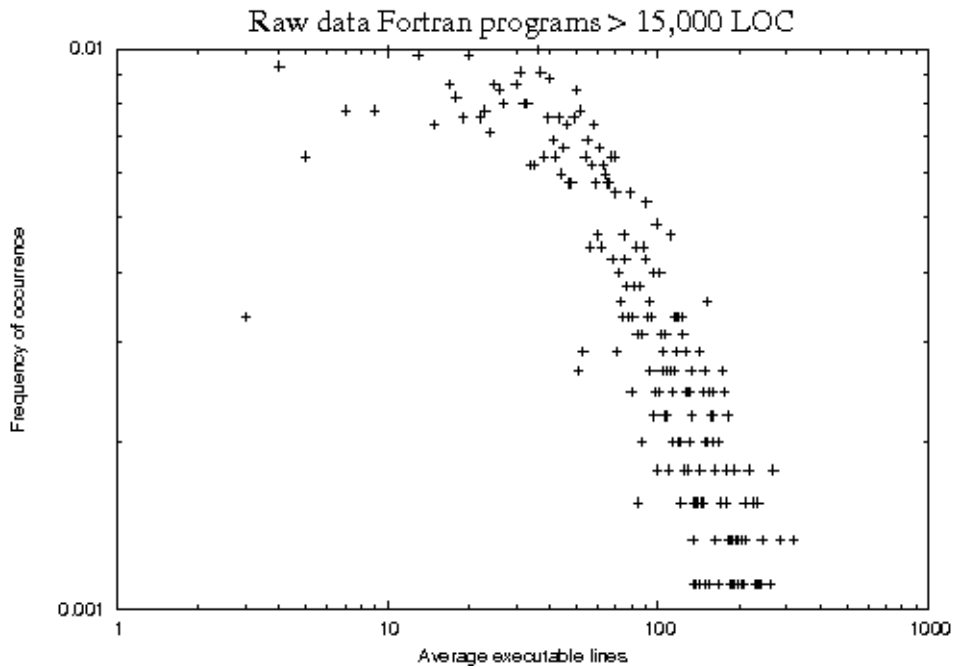
Application to software systems

Raw data for 9 systems in C > 10000 LOC



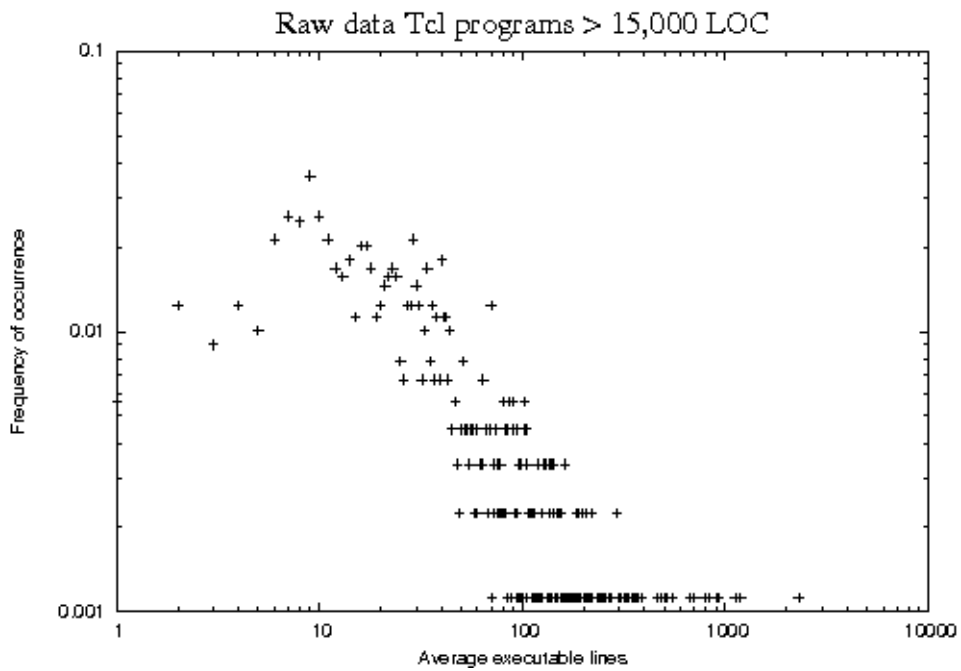
Application to software systems

Raw data for 10 Fortran systems > 15,000 – 250,000 LOC

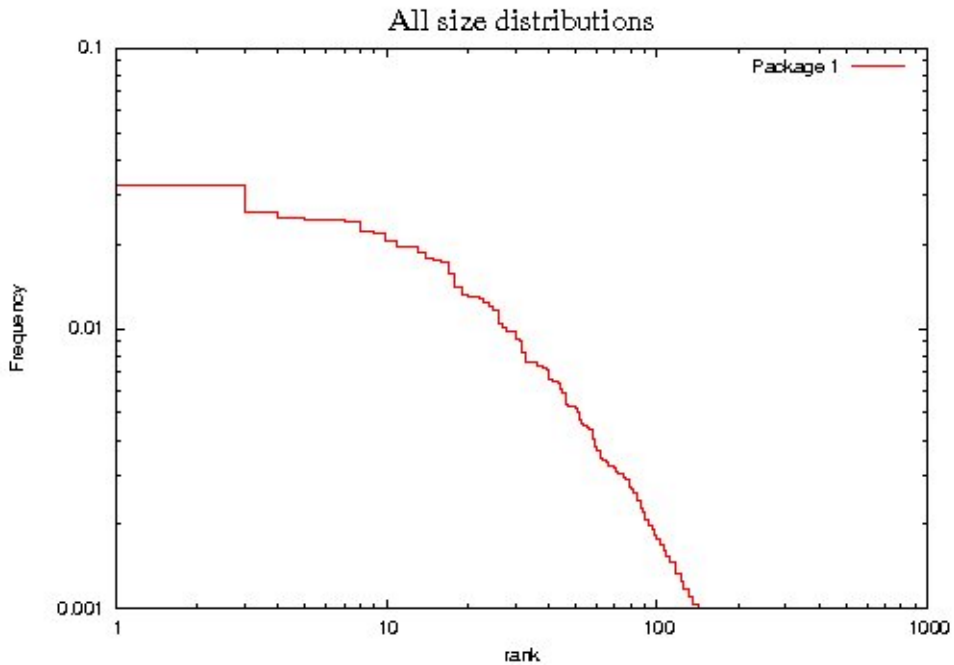


Application to software systems

Raw data for 2 Tcl/Tk systems

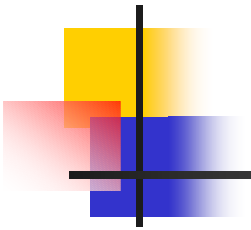


Application to software systems



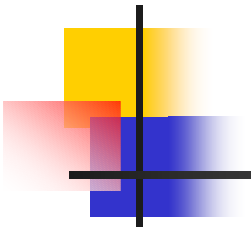
Smoothed (cdf) data for 21 systems, C, Tcl/Tk and Fortran, combining 603,559 lines of code distributed across 6,803 components.

Overview



- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

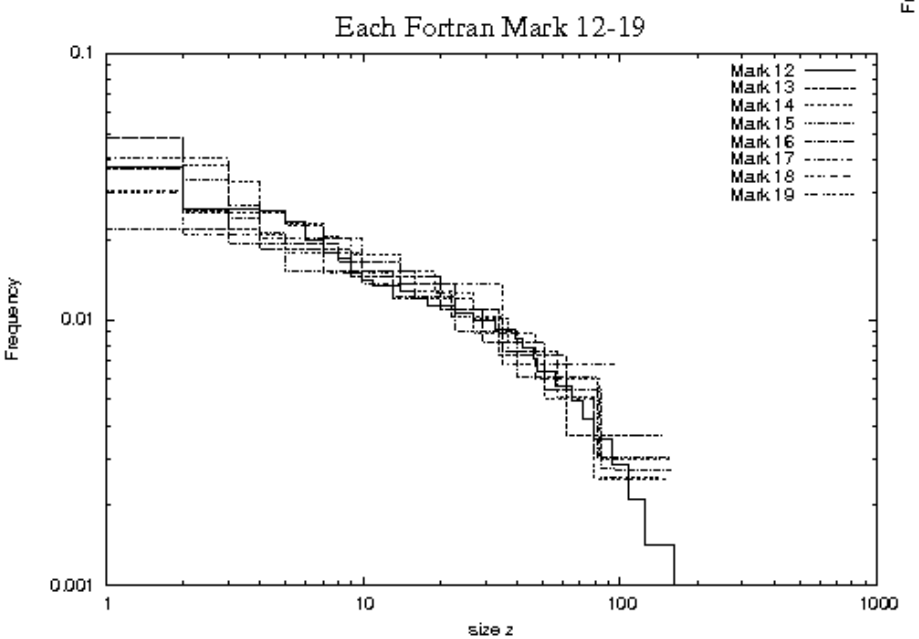
Is power-law behaviour persistent ?

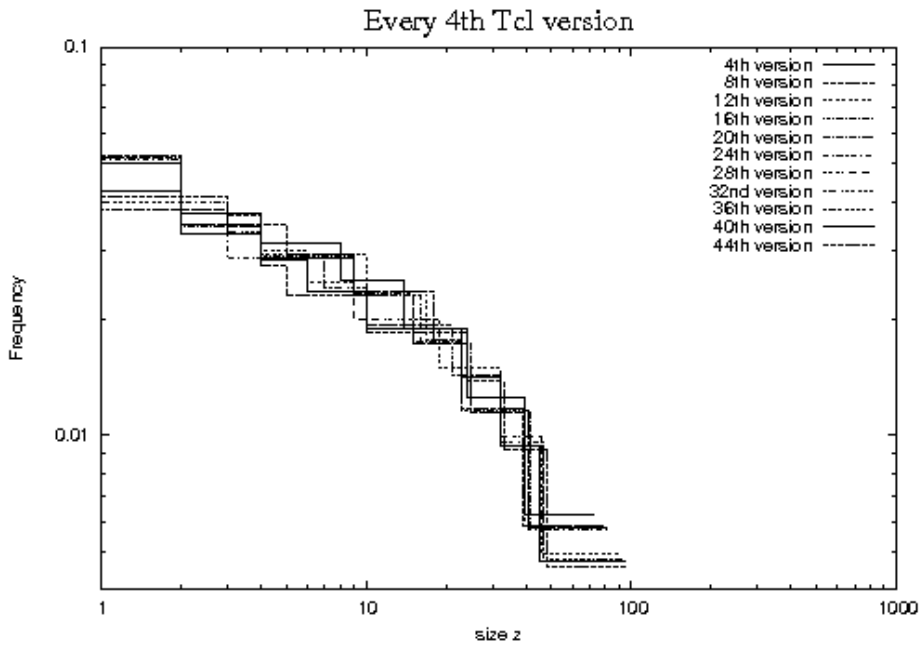


- Question: Does power-law behaviour in component size establish itself over time as a software system matures or is it present at the beginning ?

Size distributions in major systems as function of time

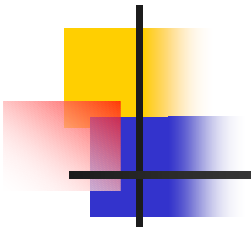
7 versions of the NAG Fortran library over 10 years





41 versions of a TCL commercial application over 6 years from birth to present

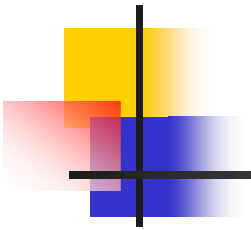
Is power-law behaviour persistent ?



- Answer: *Power-law behaviour in component size appears to be present at the beginning of the software life-cycle.*

- Given that this is independent of programming language, can we explain why ?

Overview



- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

Mechanisms leading to power-law behaviour, Newman (2006).

- Inverse of quantities
- Random walks
- Yule process
- Phase transitions
- Self-organised criticality
- Squinting closely at log-normal distributions.

- Combinations of exponentials, e.g. Miller, Mandlebrot, Shannon – the meaning of symbols

A model for emergent power-law size behaviour using Shannon entropy

A message of N tokens chosen from a code book of S unique tokens has S^N possibilities. Hartley (1928) showed that quantity of information is best defined as log (S^N).

The normal criticism of this use of information content is that Hartley(-Shannon) *only relates to the symbols and not their meaning*.

However as we saw earlier, *it's really to do with choice*.

A model for emergent power-law size behaviour using Shannon entropy

Suppose component i in a software system has t_i tokens in all constructed from an alphabet of a_i unique tokens.

First we note that

$$a_i = a_f + a_v(i)$$

Fixed tokens of a language, **{ } [ ] ; while ...**

Variable tokens, (id names and constants)

A model for emergent power-law size behaviour using Shannon entropy

An example from C:

```
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;
      }
    }
  }
}
```

Fixed (18)

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

+

Variable (8)

bubble a N i j t 1 2

Total (94)

A model for emergent power-law size behaviour using Shannon entropy

For an alphabet a_i the Hartley-Shannon information content density I'_i per token of component i is defined by

$$t_i I'_i \equiv I_i = \log(a_i a_i \ldots a_i) = \log(a_i^{t_i}) = t_i \log(a_i)$$

We think of I'_i as fixed by the nature of the algorithm we are implementing.

Consider now building a system as follows

Consider a general software system of T tokens divided into M pieces each with t_i tokens, each piece having an *externally imposed information content* density property I'_i associated with it.

1	2	3	….			
			t_i, I'_i			
					…	M

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

General mathematical treatment

The number of ways of organising this is:- $$W = \frac{T!}{t_1!t_2!...t_M!}$$

Stirling's approximation + logs as usual gives:-

$$\ln W = T \ln T - \sum_{i=1}^{M} t_i \ln(t_i)$$

In physical systems, we seek to find the most likely arrangement by maximising this subject to two constraints

$$T = \sum_{i=1}^{M} t_i \qquad \text{and } I \equiv \sum_{i=1}^{M} I_i = \sum_{i=1}^{M} t_i I'_i \equiv \sum_{i=1}^{M} t_i \left(\frac{I_i}{t_i} \right)$$

Assume fixed externally so not varied

General mathematical treatment

Using Lagrange multipliers and setting $\delta(\ln W) = 0$

leads to the most likely distribution being given by

$$p_i \equiv \frac{t_i}{T} = \frac{e^{-\beta I'_i}}{\sum_{i=1}^{M} e^{-\beta I'_i}}$$

where p_i can be considered *the probability of piece i occurring with a share e_i of U*. β is a constant.

General mathematical treatment

To summarise, for large T and t_i

The most likely distribution of the I'_i subject to the constraints of T and I held constant

$$T = \sum_{i=1}^{M} t_i \qquad \text{and} \qquad I = \sum_{i=1}^{M} t_i I'_i$$

is

$$p_i \equiv \frac{t_i}{T} = \frac{e^{-\beta I'_i}}{\sum_{i=1}^{M} e^{-\beta I'_i}}$$

General mathematical treatment

However $$I'_i = \left(\frac{I_i}{t_i} \right) = \left(\frac{t_i}{t_i} \log(a_i) \right) = \log(a_i)$$

Giving the general theorem $$p_i \sim \left(a_i \right)^{-\beta}$$

This states that in any software system, conservation of size and information (i.e. choice) is overwhelmingly likely to produce a power-law alphabet distribution. (Think ergodic here).

One last little bit of maths

- Note that for small components, the fixed token overhead is a much bigger proportion of all tokens, $a_f >> a_v(i)$, so

$$p_i = \frac{1}{Q(\beta)}\left(a_f + a_v(i)\right)^{-\beta} \approx \left(a_f\right)^{-\beta}\left(1+\frac{a_v(i)}{a_f}\right)^{-\beta} \approx \left(a_f\right)^{-\beta}$$

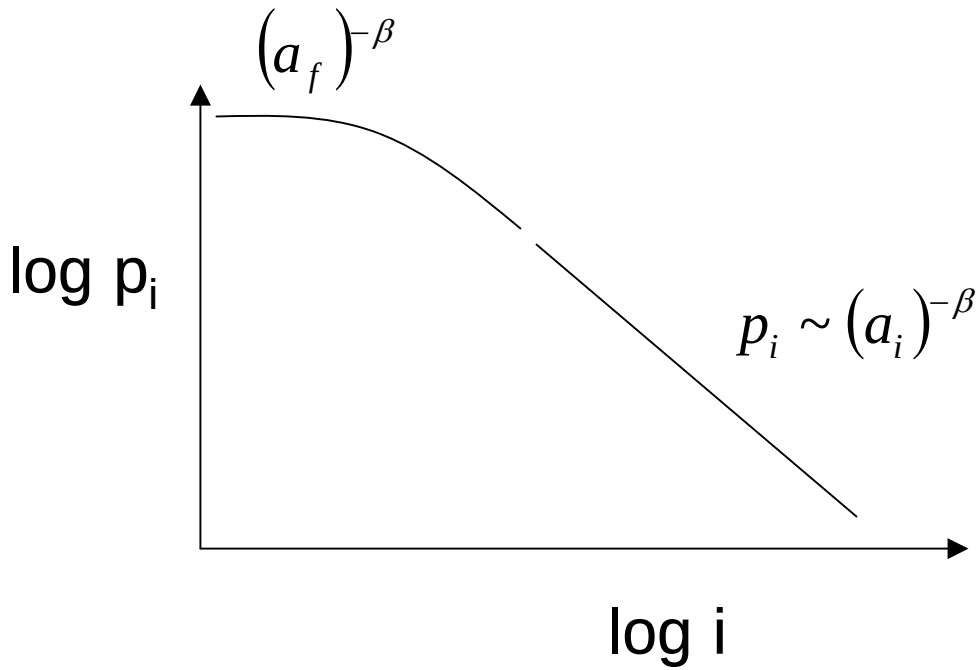
Constant

- For large components, the general rule takes over

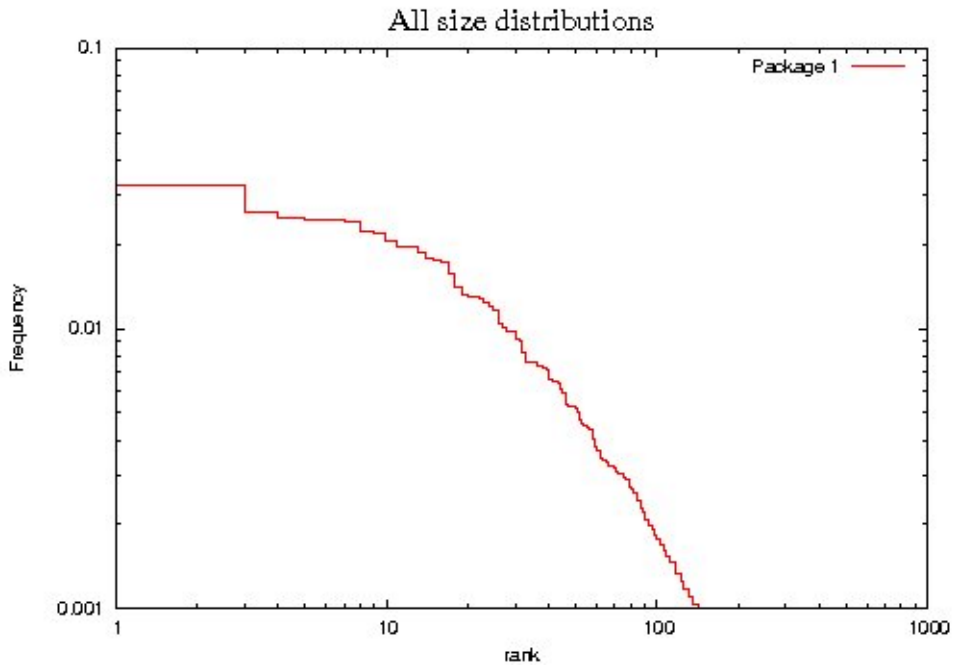
$$p_i \sim \left(a_i\right)^{-\beta}$$

Application to software systems

So we are looking for the following signature

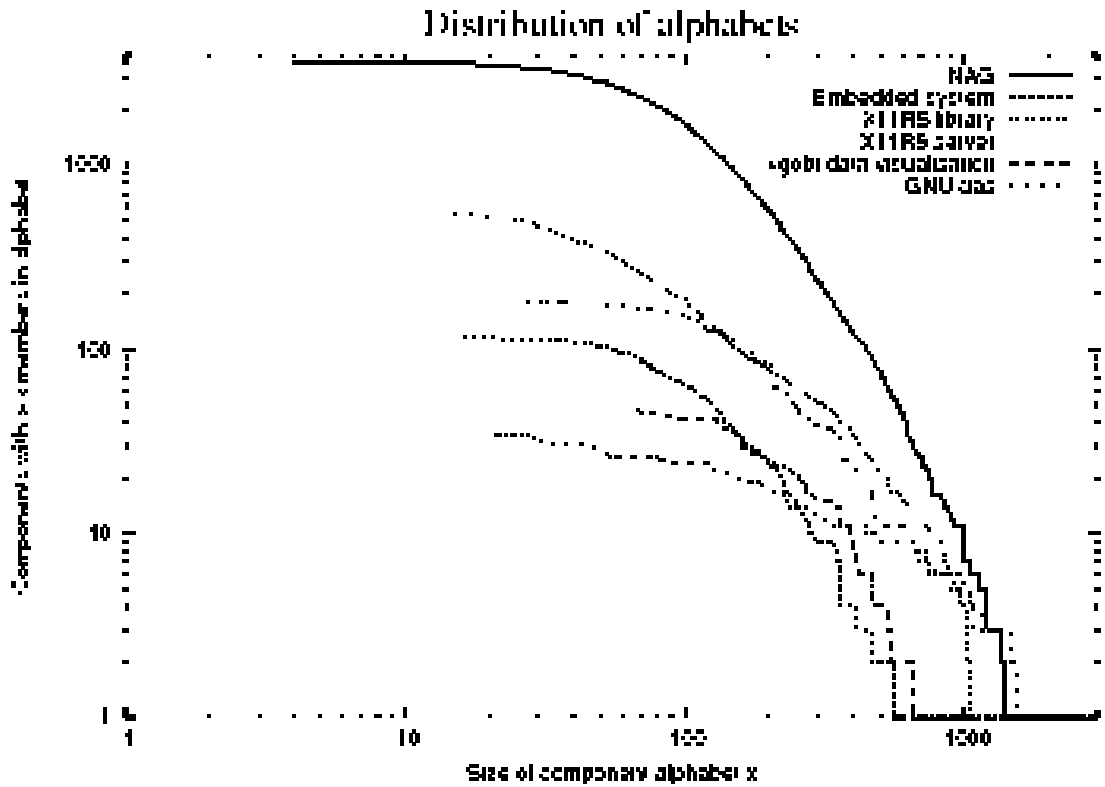


Application to software systems



Smoothed (cdf) data for 21 systems, C, Tcl/Tk and Fortran, combining 603,559 lines of code distributed across 6,803 components, Hatton (2009), IEEE TSE.

Application to software systems



Six randomly chosen individual systems

Some model predictions

- Suppose there is a constant probability P of making a mistake on any token. The total number of defects is then given by $d_i = P.t_i$ Then

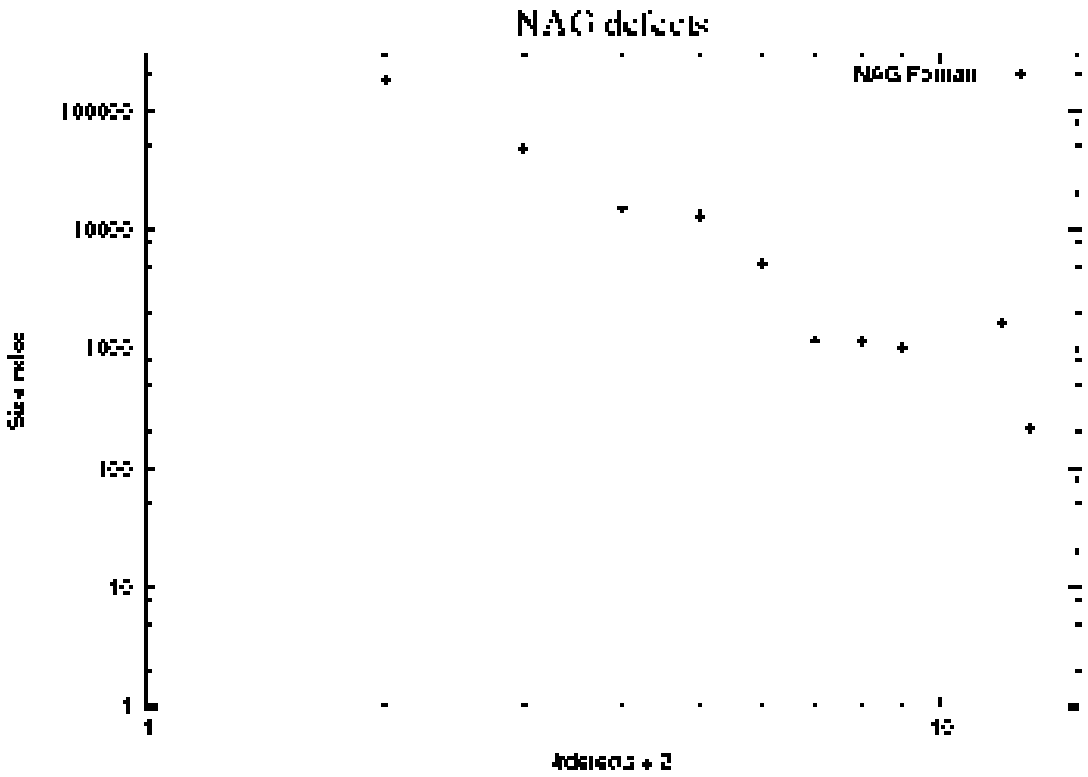
$$p_i = \frac{1}{Q(\beta)}(a_i)^{-\beta} \approx (t_i)^{-\beta} \approx (d_i)^{-\beta}$$

This step uses Zipf's law, Hatton (2009)

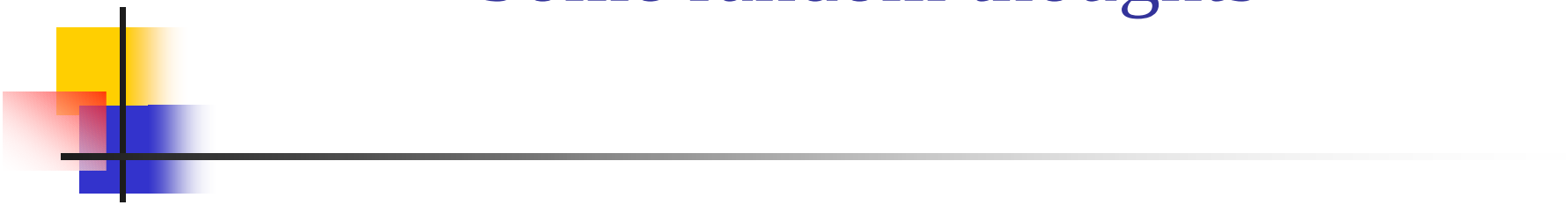
- So defects will also be distributed according to a power-law.

Defect clustering in the NAG Fortran library (over 25 years)

Defects	components	XLOC
0	2865	179947
1	530	47669
2	129	14963
3	82	13220
4	31	5084
5	10	1195
6	4	1153
7	3	1025
> 7	5	1867

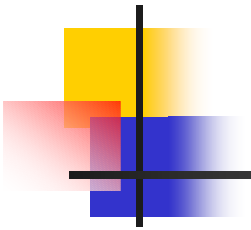


Some random thoughts



- I think it is useless to ask the question why so many components exhibit zero defect. It is a statistical side-effect and as useless as asking why somebody has won the lottery.
- Note that defect clustering strongly implies that if you find a defect, you should carry on looking as there is an increased probability of finding more.
- Although I won't prove it here, a natural extension of this shows that in a mature system, the total number of defects in a component of t tokens is ~ t log t, for which there is also some experimental evidence.

Overview



- Some thoughts about building systems
- What is scale-free behaviour ?
- Scale-free behaviour in component size
- Is scale-law behaviour persistent in software systems ?
- A tentative unifying principle
- Conclusions

Conclusions so far

- Observation suggests that power-law behaviour in component size is
  - *Independent of programming language*
  - *Independent of functionality*
  - *Independent of maturity*
- Using statistical mechanics and Shannon's information theorem we can prove that **any** system built under the constraints of fixed size and fixed information content will lead to power-law component size behaviour according to

$$p_i \sim (a_i)^{-\beta}$$

- This model predicts the well-observed phenomenon of defect clustering (and a few other things I haven't time to discuss).

References

My writing site:-

http://www.leshatton.org/

Specifically,

http://www.leshatton.org/variations_2010.html

Thanks for your attention.