IFIP working conference on Uncertainty quantification in scientific computing, Boulder, 1-4 Aug, 2011.

“Scientific Computation and the Scientific Method: a tentative road map for convergence"

Les Hatton

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

Version 1.1: 28/Jul/2011

Overview

- Popperian deniability
- Some early thoughts
- A tentative model for defect
- Conclusions

Popperian deniability

- Truth cannot be verified by scientific testing, it can only be falsified.
- Falsification requires quantification of experimental error.
- This has been at the heart of scientific progress.
- This process is NOT generally followed in scientific (or indeed any other kind of) computation.

The problem with defects

- We seek quantification. This means we would like to know how big the errors in our numerical experiments are.
- Unfortunately, most of what we know concerns how many defects are present and **not** how big a problem they cause.
- More than a whiff of chaos
  - {int a; b = (a=0) + a; … b can be almost anything.
  - 14 out of 14 compilers got volatile wrong in a 2008 study
  - Undetected array bound violations still with us in 2011 !
- Any engineering technology which relies on somebody getting it ‘right’ is fundamentally flawed.

Overview

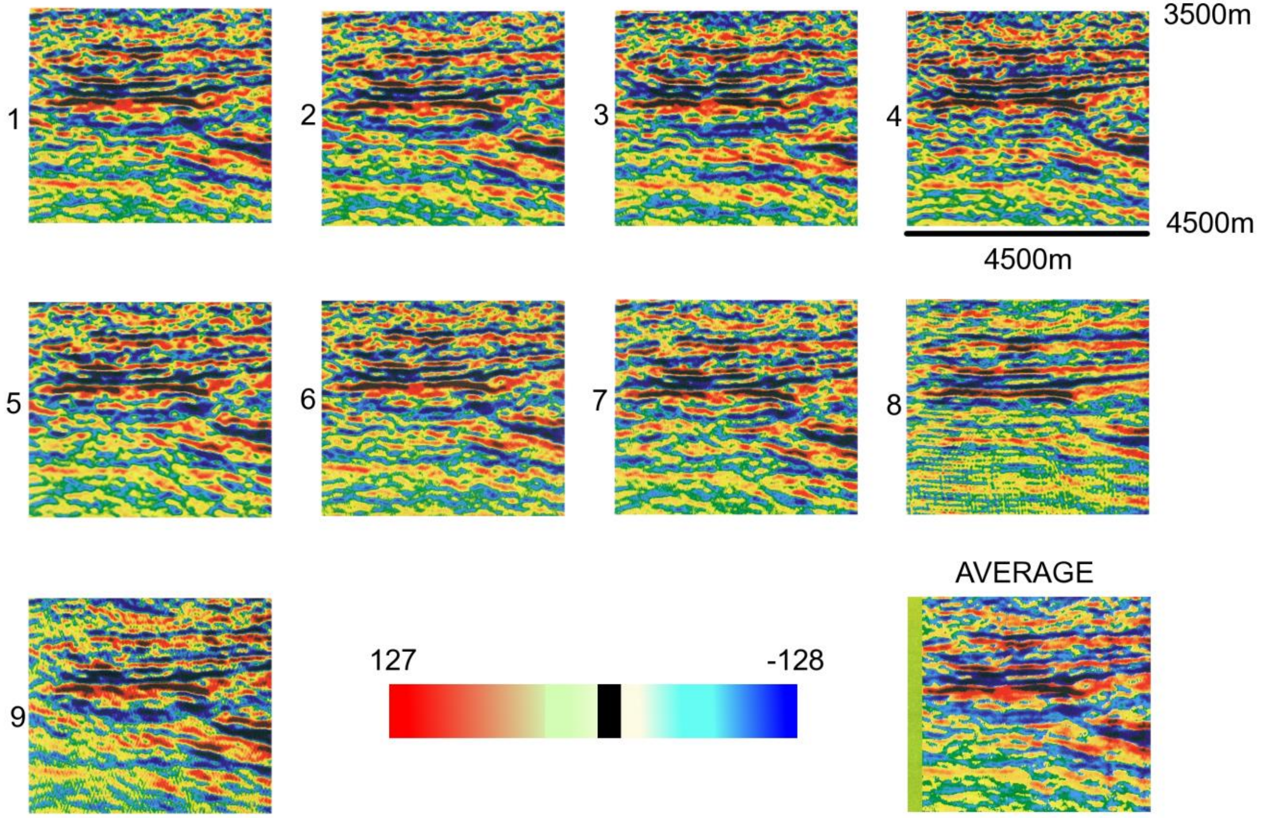
- Popperian deniability
- <u>Some early thoughts</u>
- A tentative model for defect
- Conclusions

Some early thoughts

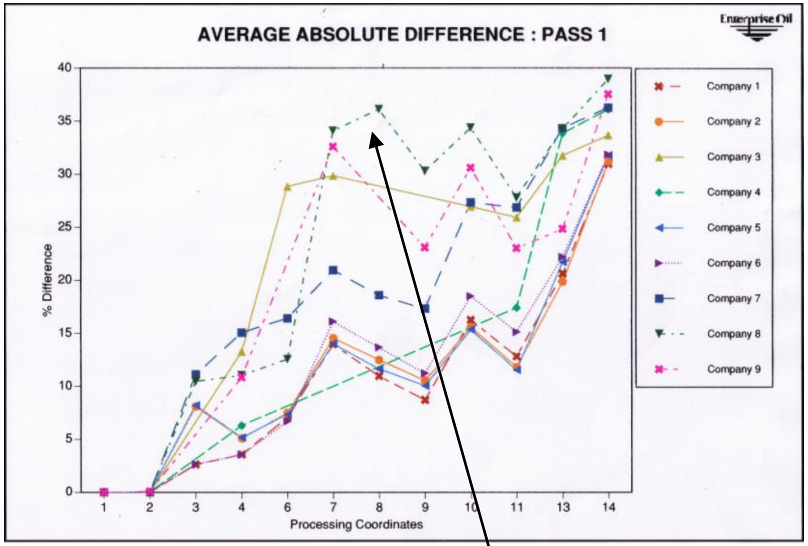
By 2010 I was reasonably convinced that:

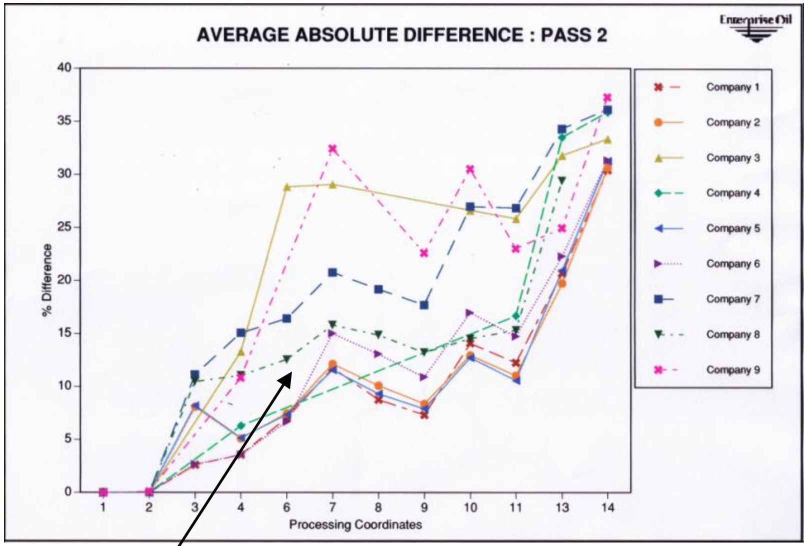
- N-version experiments are exceedingly valuable at highlighting differences, (for whatever reason), and effective at reducing those differences. (1994)
- Scientific software is littered with statically detectable faults which fail with a certain frequency (1997)
- The language does not seem to make much difference. (1999-)
- Defects appear to be fundamentally statistical rather than predictive, (2005-8)
- Software systems exhibit implementation INdependent behaviour (2007-10).

Quantification of differences by N-version (1994)



Convergence using N-version – but to what ?





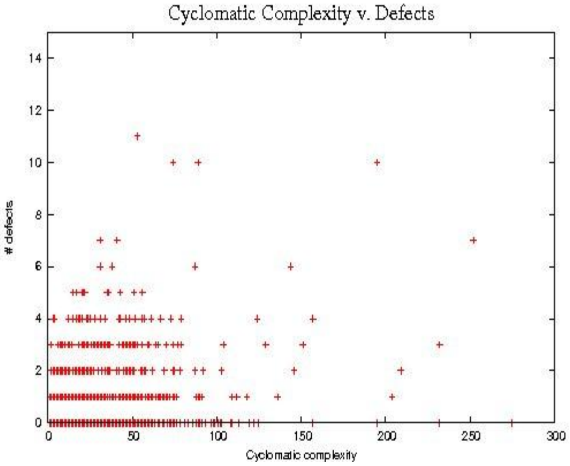
Before

After

Are defects related to static complexity ?

- There is little evidence that complexity measures such as the cyclomatic complexity v(G) are of any use at all in predicting defects

Defects

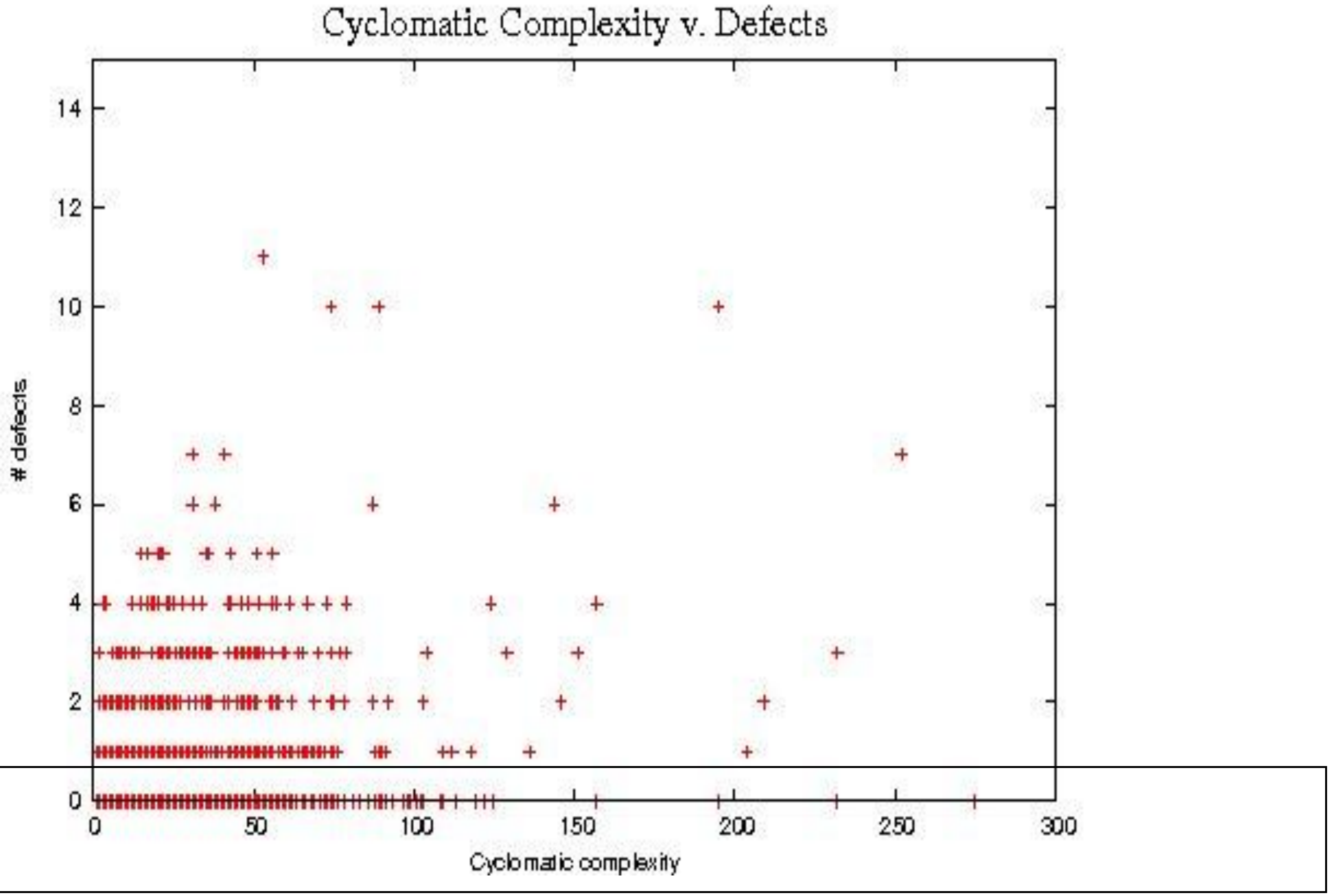


Cyclomatic number v(G)

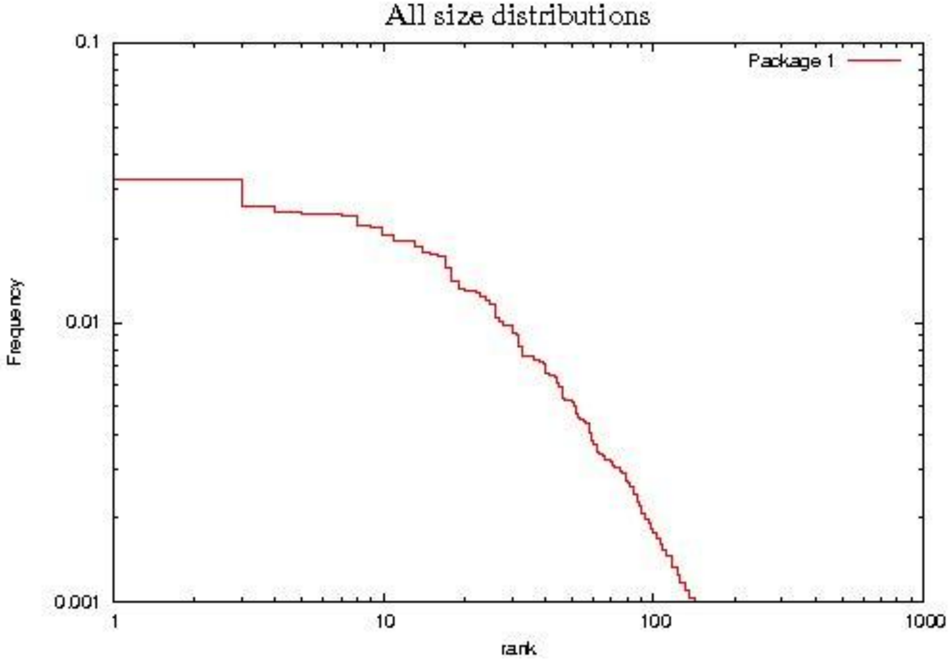
NAG Fortran library over 25 years
(Hopkins and Hatton (2008))

Is there anything unusual about ‘zero’ defect ?

PCA and endless rummaging suggest not. This may undermine *root-cause analysis*.



Software size distributions appear power-law in LOC



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)

Overview

- Popperian deniability
- Some early thoughts
- <u>A tentative model for defect</u>
- Conclusions

A tentative model

We are looking for:-

- Language independent behaviour
- Application independent behaviour
- Predicts power-law behaviour in component sizes
- Predicts simple and apparently *power-law* behaviour in defect, (observed frequently)
- Makes other testable predictions.

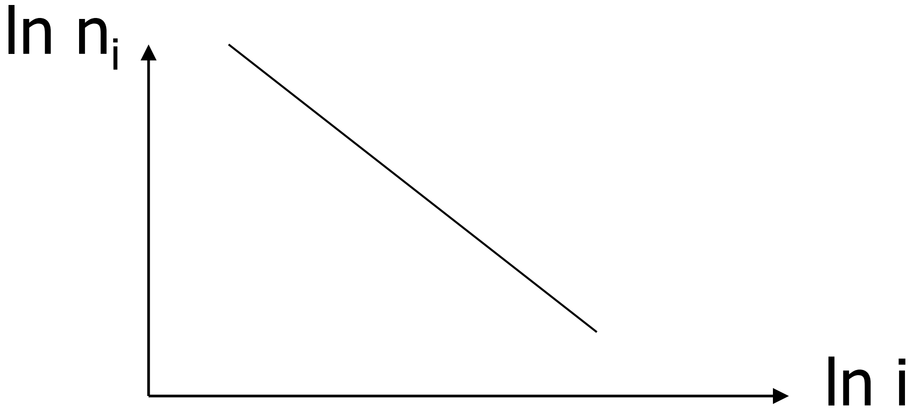
What is power-law behaviour ?

Frequency of occurrence n_i given by $n_i = \frac{nc}{i^p}$

This is usually shown as

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

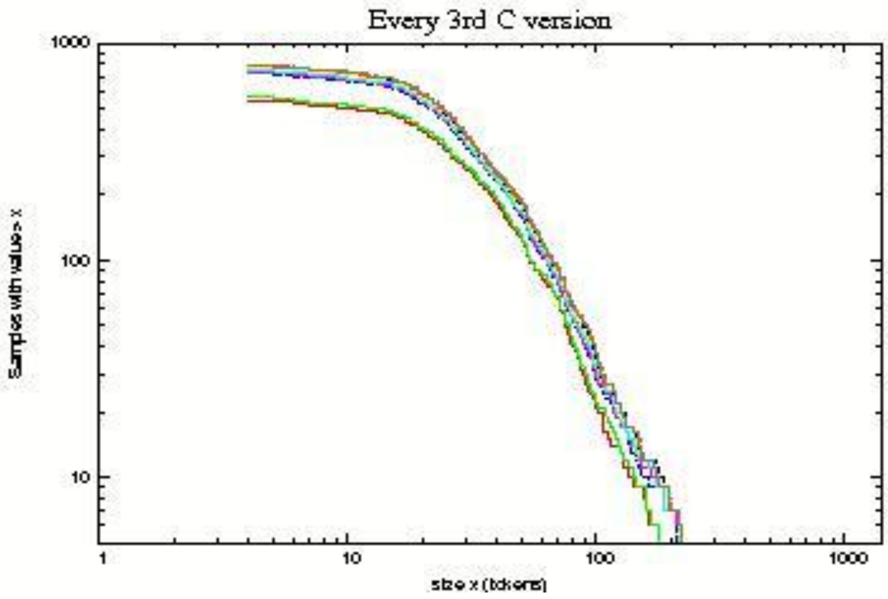
which looks like

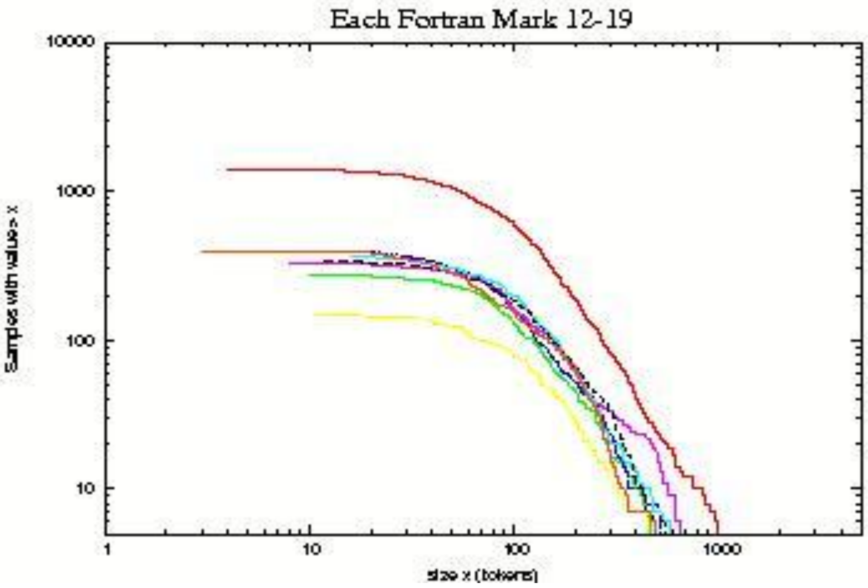


Is power-law behaviour persistent ?

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

Is power-law behaviour persistent ?





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 appears independent of programming language and application area, can we explain why ?

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

- 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)).

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:

Fixed (18)

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

+

Variable (8)

bubble a N i j t 1 2

```
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;
   }
  }
 }
}
```

Total (94)

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

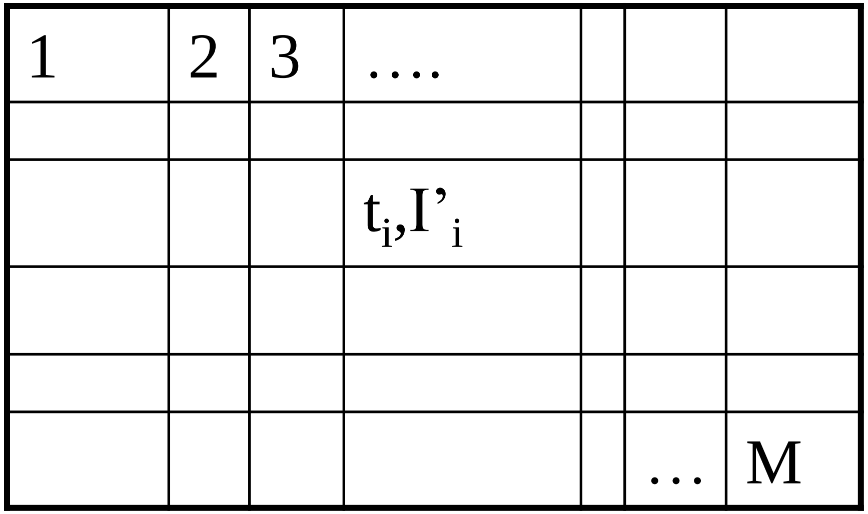
For an alphabet a_i the Hartley-Shannon information content density I'_i per token of component i is defined by

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

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. *Note: no nesting.*



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

$$I = \sum_{i=1}^{M} t_i I'_i$$

General mathematical treatment

The most likely distribution of the I'_i (= I_i/t_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

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

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

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

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

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

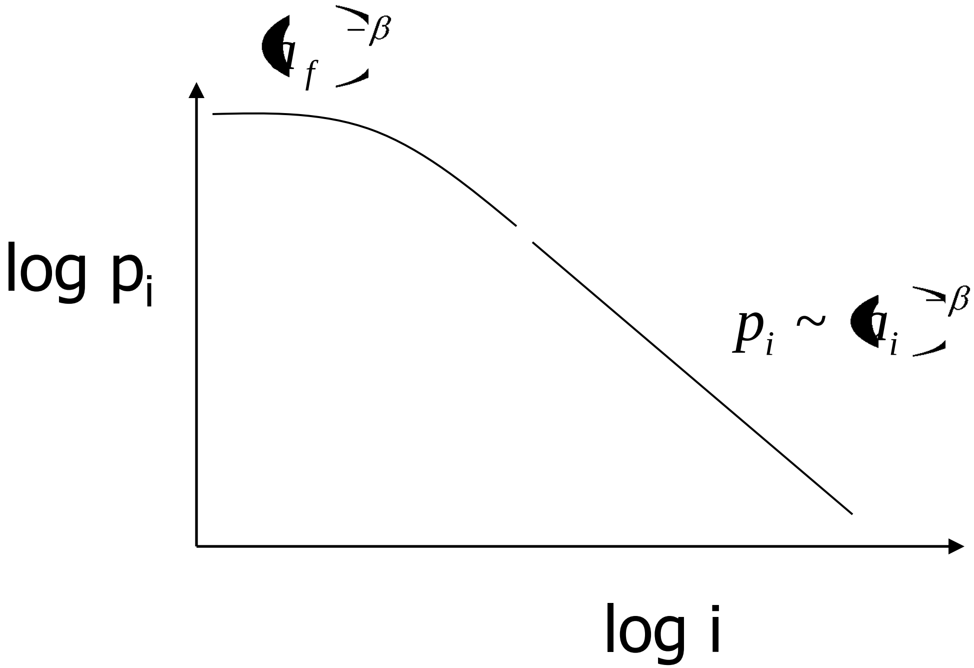
$$p_i = \frac{1}{Q(\beta)} \left(a_f + a_v(i) \right)^{-\beta} \approx \left(a_f \right)^{-\beta} \left(1 + \frac{a_v(i)}{a_f} \right)^{-\beta} \approx \underset{\text{Constant}}{\left(a_f \right)^{-\beta}}$$

- For large components, the general rule takes over

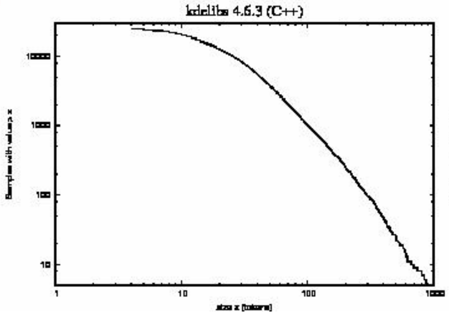
$$p_i \sim \left(a_i \right)^{-\beta}$$

Application to software systems

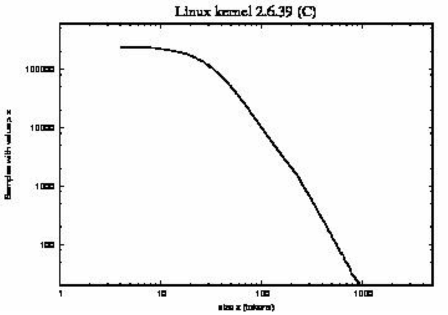
So we are looking for the following signature



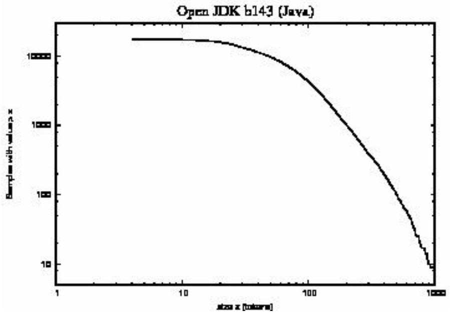
Some results



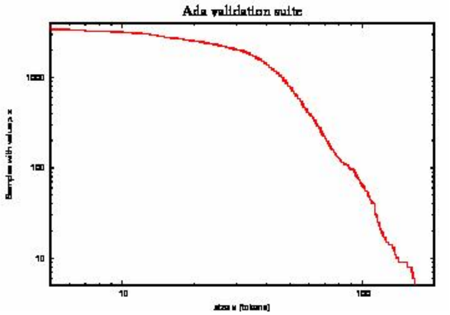
C++



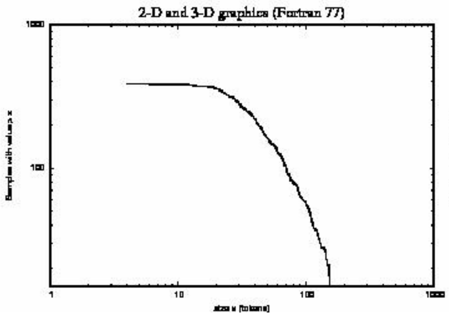
C



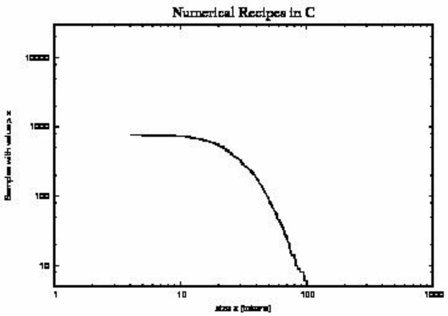
Java



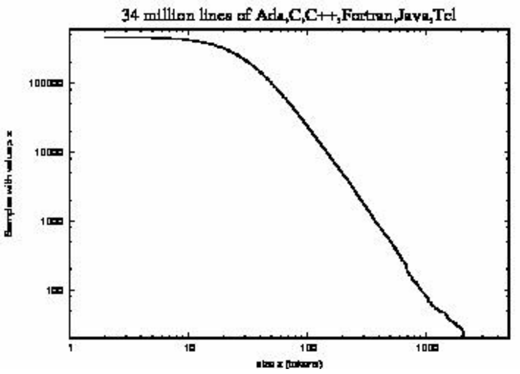
Ada



Fortran



C Numerical



34 million lines of Ada, C, C++, Fortran, Java, Tcl in 75 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

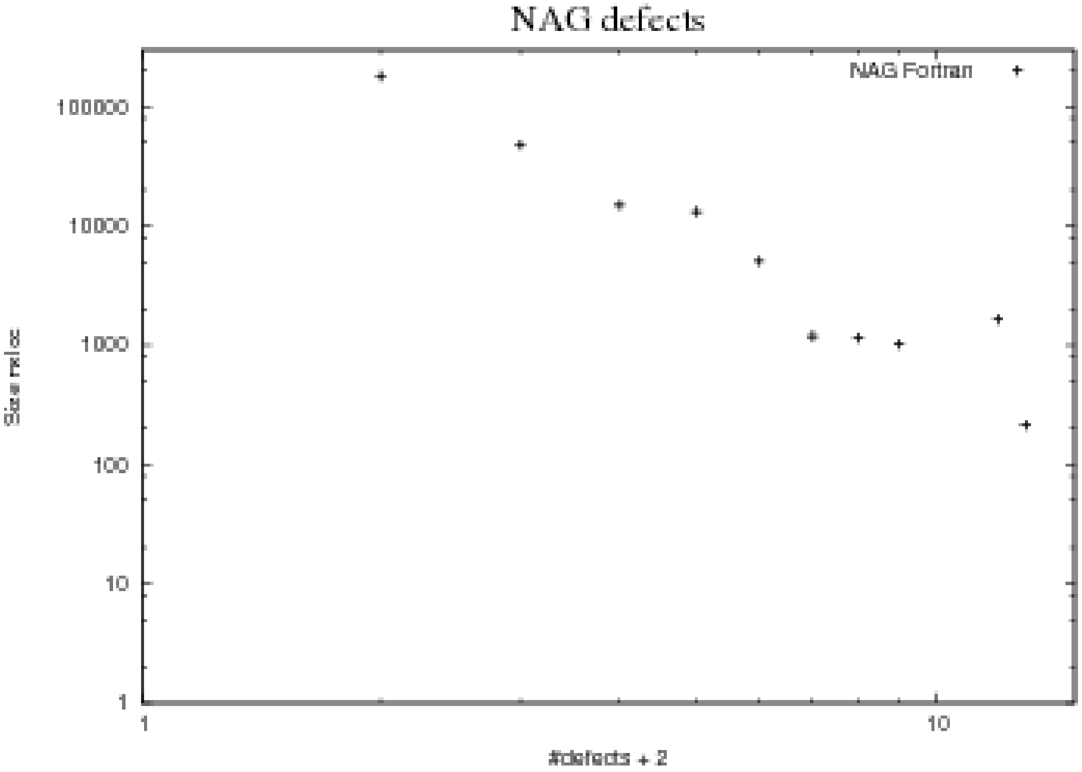
$$p_i = \frac{1}{Q(\beta)} \left(a_i\right)^{-\beta} \approx \left(t_i\right)^{-\beta} \approx \left(d_i\right)^{-\beta}$$

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

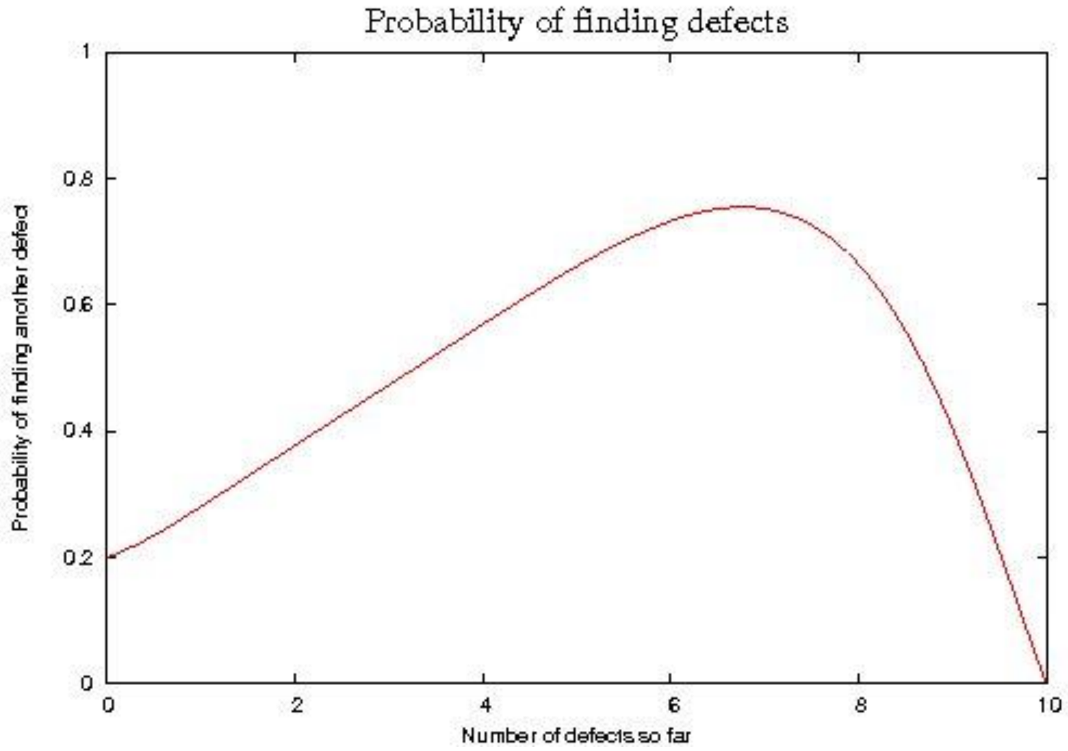
- So defects will also be distributed according to a power-law – *i.e they will cluster*.

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



Clustering can be exploited: Conditional probability of finding defects*



* See, Hopkins and Hatton (2008), http://www.leshatton.org/NAG01_01-08.html

Overview

- Popperian deniability
- Some early thoughts
- A tentative model for defect
- <u>Conclusions</u>

Conclusions

- Bounding defects is inherently difficult but N versions (or open source) both seem to offer ways of improving software agreement but by an unknown amount.
- Static structural relationships with defect appear to be a blind alley, (cyclomatic complexity …,).
- Defects cluster and this can be exploited.
- Software systems appear to exhibit macroscopic behaviour independent of implementation or language

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

References

My writing site:-

http://www.leshatton.org/

Specifically,

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

Thanks for your attention.