

# **Libraries and their Reuse: Entropy, Kolmogorov complexity, and Zipf's Law**

Todd Veldhuizen  
Open Systems Laboratory  
Indiana University Bloomington

October 16, 2005

# Reuse

Increase productivity and decrease defect rates by reusing reliable components.

What is the role of libraries in fostering reuse?

How much reuse is achievable? (A common “target figure” is 85% or so.)

And what will it look like when we get there?

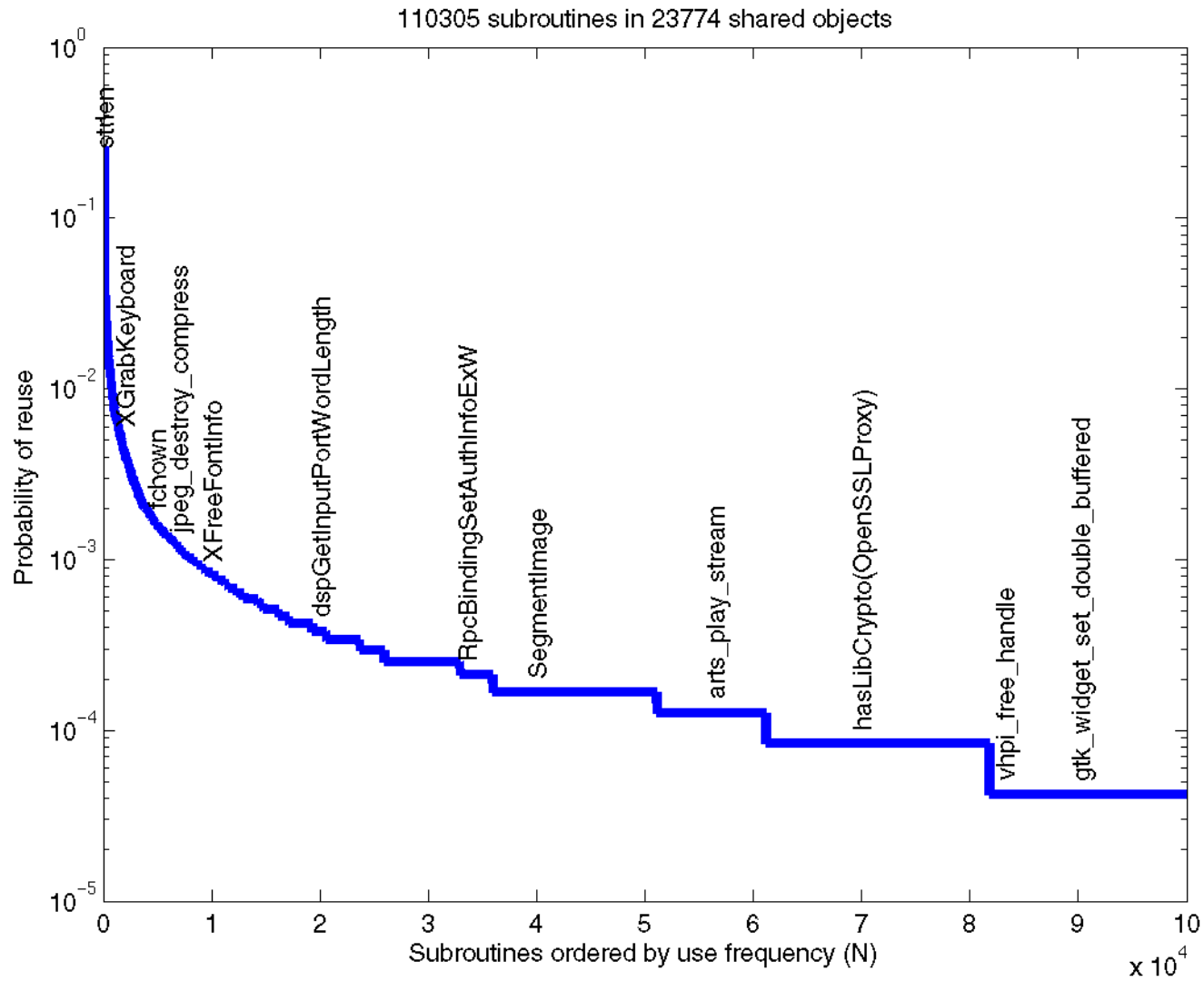
# Hypothesis 1

**Strong reuse.** Large-scale reuse will allow mass-production of software, with applications being assembled by composing large, pre-existing components. The activity of programming will consist primarily of choosing appropriate components from libraries, adapting and connecting them.

Closely associated with Component-Based Software Engineering (CBSE).

## Hypothesis 2

**Weak reuse.** Large-scale reuse will offer useful reductions in the effort of implementing software, but these savings will be a fraction of the code required for large projects. Nontrivial projects will always require the creation of substantial quantities of new code that cannot be found in existing component libraries.



## **Libraries as software compression**

Can we think of libraries as a tool for compressing programs?

Libraries capture commonly occurring motifs and patterns — like a “codebook” for compression.

If we design our library APIs well, you avoid “boilerplate” — repetitious code.

If we can reuse 80% of our code from a library, this “compresses” our program by 80%.

# **Background: Kolmogorov Complexity**

# Kolmogorov Complexity

Kolmogorov complexity, also known as Algorithmic Information Theory, was founded in the 1960s by three individuals:



R. Solomonoff



G. Chaitin



A. N. Kolmogorov



# Kolmogorov Complexity

Choose a universal machine/Turing-complete programming language  $\phi$ .

Define the Kolmogorov complexity  $C_\phi(x)$  as the shortest program producing  $x$ , given an empty string as input.

e.g., using Java: for a string  $x$ ,  $C_{\text{Java}}(x)$  is the length of the shortest Java program producing  $x$ .

# Invariance Theorem

All the really succinct languages are within a constant factor of each other.

**Theorem 1.** *There is a universal machine  $\Phi$  such that for any machine  $\phi$ , there is a positive constant  $c$  such that*

$$C_{\Phi}(x) \leq C_{\phi}(x) + c$$

Changing machines (programming languages) within the “optimal class” changes Kolmogorov complexity only by a constant additive factor.

Corollary: current programming languages are about as good as one can get; they are all equally expressive in a K-complexity sense.

## Infinite objects

Kolmogorov complexity may also be examined for infinite objects, e.g.,

$$\pi = 3.141592653589793238462643383279502884197169399375105820974 \dots$$

Take  $C(x)$  to be the length of the shortest program that enumerates the object (e.g., its digits, its elements, its Robinson diagram).

We may also speak of a program having a Kolmogorov complexity, by regarding that program as a mathematical object (e.g., a partial computable function, Robinson diagram of a transition structure).

# You can't compress random data

Almost every object is incompressible:

**Theorem 2.** *Let  $x$  be a string chosen uniformly at random. Then almost surely:<sup>1</sup>*

$$C(x) \geq \|x\| - g(\|x\|)$$

*for any function  $g(n)$  with  $\lim_{n \rightarrow \infty} g(n) = \infty$ .*

e.g., take  $g(n) = \alpha^{-1}(n, n)$  (inverse Ackermann)

---

<sup>1</sup>almost surely means, “with probability 1”

## A fruitful paradox

1. Kolmogorov complexity tells us: with probability 1 we cannot compress programs. Therefore we can expect to reuse  $\sim 0\%$  of our programs from libraries, asymptotically speaking.
2. On the other hand, libraries are useful and our subjective experience is that they DO let us shorten programs. A typical linux distro has a half-million distinct subroutines available in libraries — who would write all those if they weren't useful?

## Paradox resolved

1. The paradox is avoided only if we assume a *nonuniform* distribution on programs. Then libraries *do* shorten programs, on average, with no contradiction to Kolmogorov incompressibility.
2. We can model a problem domain as a *random source* of programs with a distribution that weights certain classes of programs as “more probable.”

# **An abstract model of software reuse**

# Libraries as software compression

What do we need to model?

- Problem domains
- Libraries
- Reuse rates

## A problem domain

A problem domain has a population of programmers who write programs. The projects undertaken in the domain share commonality of purpose: common abstractions, recurring themes or motifs, approaches, etc.

We can model this by a probability distribution on specifications; equivalently a.e., by Kolmogorov invariance, a probability distribution on *programs* written without use of a library.

To avoid sledgehammer math we think about a sequence of distributions  $p_1(w), p_2(w), \dots$  where  $p_s$  gives a probability distribution over programs of length  $\leq s$ .

# Entropy

For a distribution  $p_s(n)$  with  $n$  fixed, take its entropy to be:

$$H(p_s) = \sum_{w:|w|\leq s} -p_s(w) \log_2 p_s(w)$$

Define the 'entropy parameter' of a problem domain as:

$$H = \limsup_{s \rightarrow \infty} \left( \frac{1}{|A^{\leq s}|} H(p_s) \right)$$

where  $|A^{\leq s}|$  is the number of programs of length at most  $s$ .

## Entropy: interpretation

$1 - H$  gives the maximum achievable reuse for a problem domain:

- $H = 0$ : problem domain is not very interesting; can achieve 100% reuse!  
Have to write almost no code...
- $0 < H < 1$ : can reuse  $(1 - H)$  of a program's code from library;
- $H = 1$ : no reuse is possible (classical Kolmogorov complexity with distribution uniform or nearly uniform).

## A library

Finite libraries do not offer any reduction in the size of programs, asymptotically speaking, because of Kolmogorov Invariance:

Given a language  $\phi$  and a library  $L$ , we can package the two together and call  $\phi + L$  a language, then apply invariance.

Instead we need to think of libraries as infinite or 'potentially infinite'.

## An infinite “platonic” library

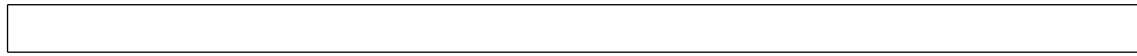
We can think of a library for a problem domain as, equivalently:

- A finite library that is ever-expanding as we explore the problem domain and discover useful abstractions and algorithms;
- A truncated version of some infinite “platonic” library that exists already for the problem domain; as time goes on we uncover more and more of its contents.

The contents of the library may not (will not) be computably enumerable (i.e., have a finite description we can use to produce its contents), so Kolmogorov invariance does not apply: we *can* achieve compression.

# The model

Uncompressed program (without library)

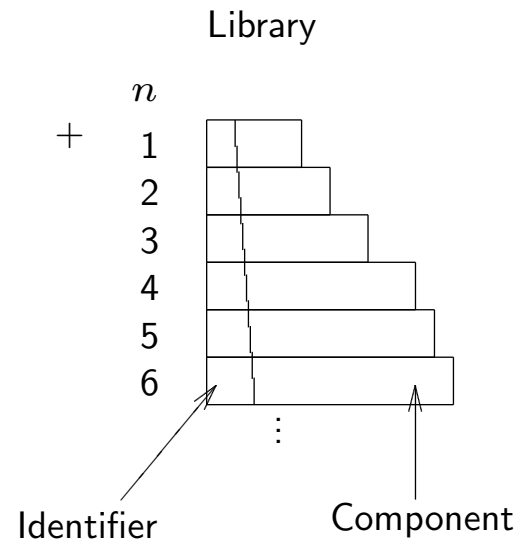


$s$  bits

Compressed program (with library)



$\geq Hs$  bits



# Predictions of the model

## A bound on reuse rates

Let  $\lambda(n)$  be the rate at which the  $n^{\text{th}}$  component is used in ‘compressed’ programs.

**Theorem 3.** *The reuse rates  $\lambda(n)$  are asymptotically bounded above by:*

$$\lambda(n) \prec \frac{1}{n \cdot o(n^\epsilon)} \quad (5)$$

This result is corroborated by the  $n^{-1}$ -like curves we see in practice.

## Achievability of $H$

Optimal compression with libraries is achievable:

**Theorem 4.** *There exists a library with which uncompressed programs of size  $s$  can be compressed to expected size  $\sim Hs$ .*

Construction uses Shannon-Fano codebooks, one for each program size, and amalgamates them together into one library.

(You wouldn't want to do this in practice.)

## Library incompleteness

aka The Full Employment Theorem for Library Writers

**Theorem 5. [Library Incompleteness]** *If a problem domain has  $0 < H < 1$  and 'honest' distributions, no finite library can achieve an asymptotic compression ratio of  $H$ .*

No finite library is complete; there are always more useful abstractions to be discovered. Discovering these abstractions requires creativity.

## Size of library components

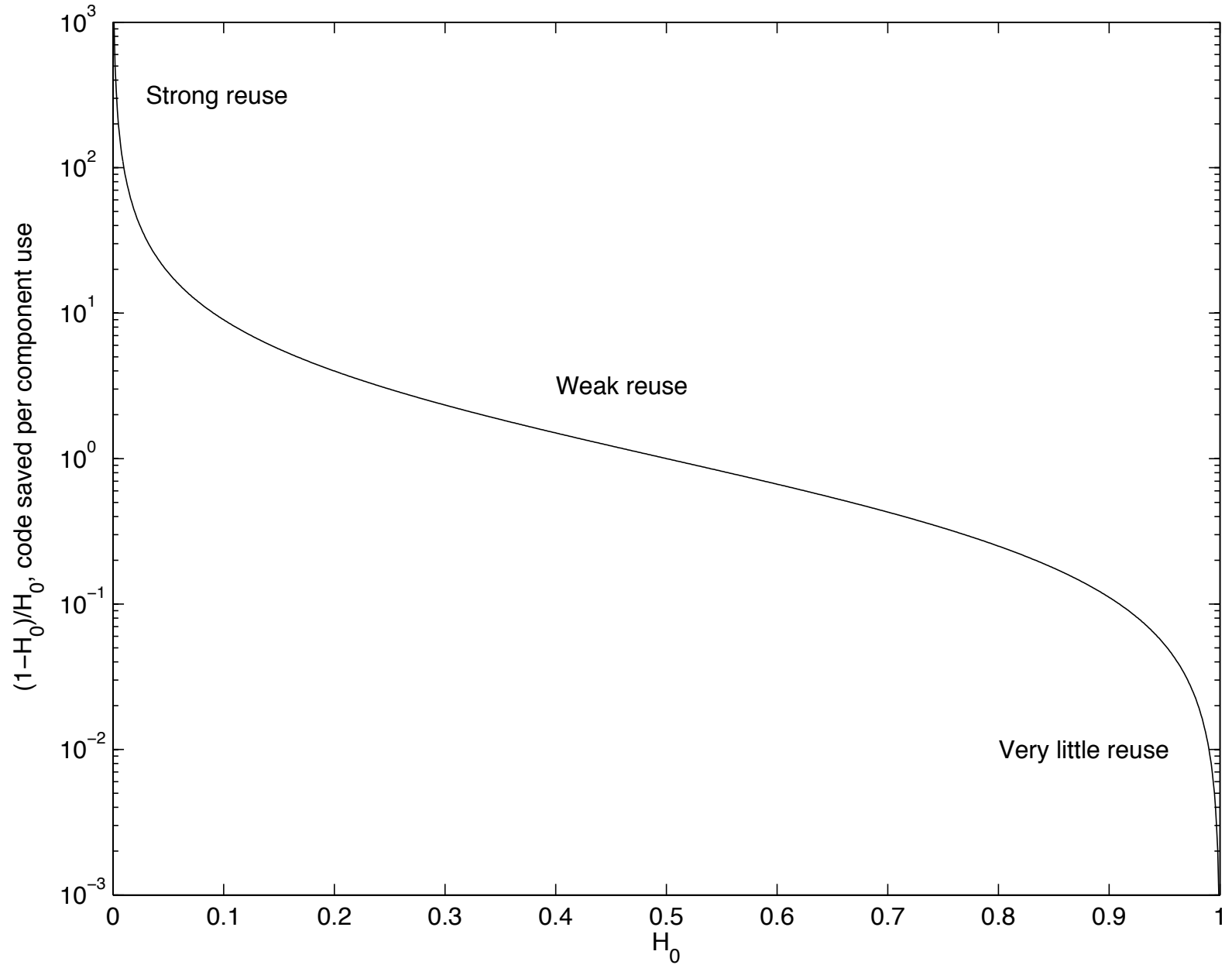
Let  $S(n)$  be the amount by which use of the  $n^{\text{th}}$  library component can reduce the size of your program.

**Theorem 6.** *If a library achieves a compression factor of  $H > 0$  in an honest problem domain, then  $S(n) \sim \frac{1-H}{H} \cdot o(n^\epsilon)$  for any  $\epsilon > 0$ .*

The important thing is the multiplier:

$$\frac{1 - H}{H} \tag{6}$$

This multiplier suggests  $H \rightarrow 0$  is where “strong reuse” is plausible (programming by wiring together big components).



# **Empirical results**

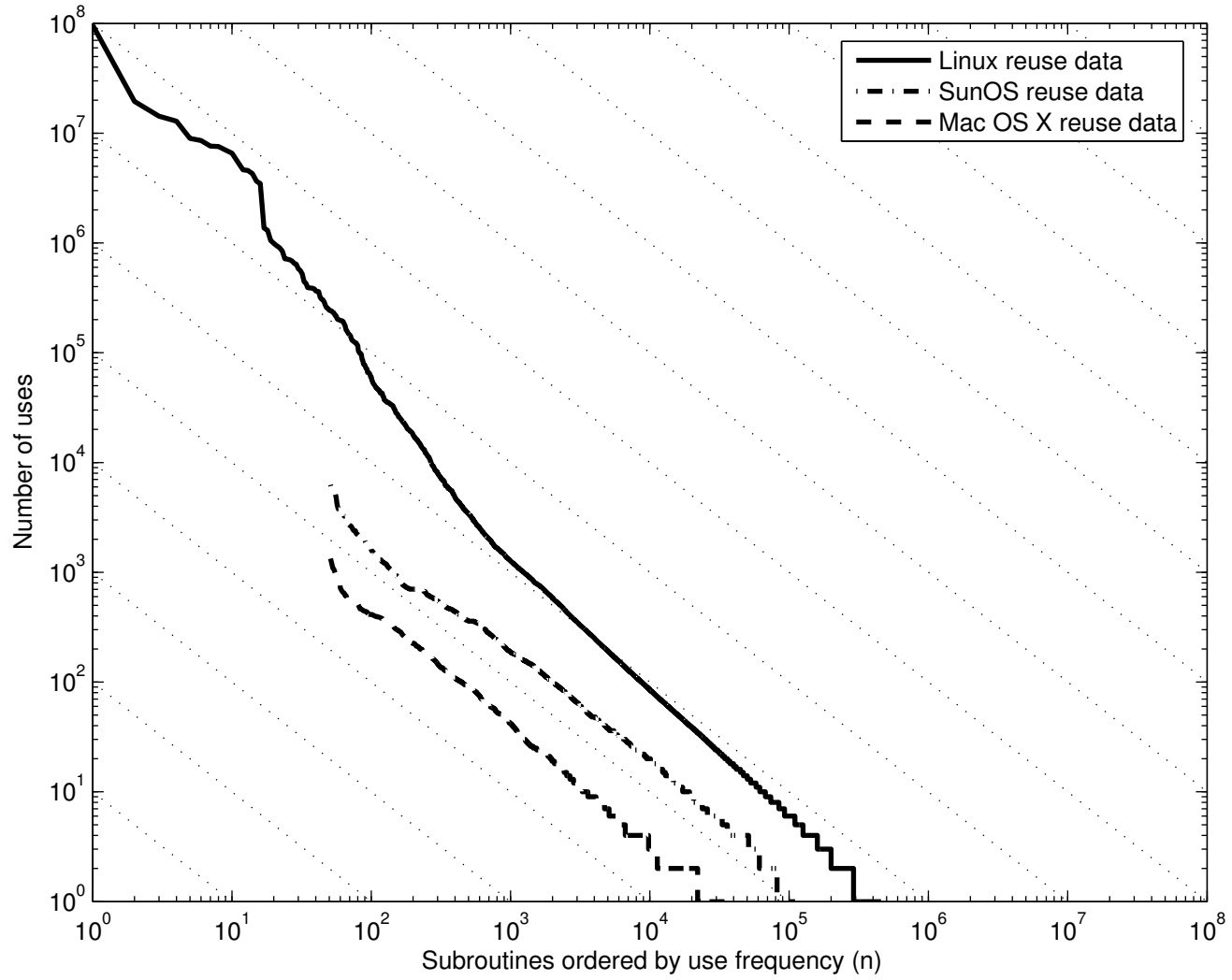
## Data from reuse on Unix systems

Methodology: collect reuse data by scanning filesystems of a Unix system, locating unique shared objects, and analyzing the reuse of library subroutines.

For SunOS, Mac OS X: nm, objdump

For Linux: disassemble, decode PLT and GOT tables for shared library calls, also collect reuse frequencies of machine instructions.

Operating System	# Objects	# Components
Linux (SuSE)	12136	455716
SunOS	23774	110306
Mac OS X	2334	37677



# **Food for thought**

## **Are library components the prime numbers of software?**

An integer factors into a product of primes.

Software can be factored into an assembly of components.

There are infinitely many primes. There are infinitely many components.

Up to a log factor, primes and software components agree on:

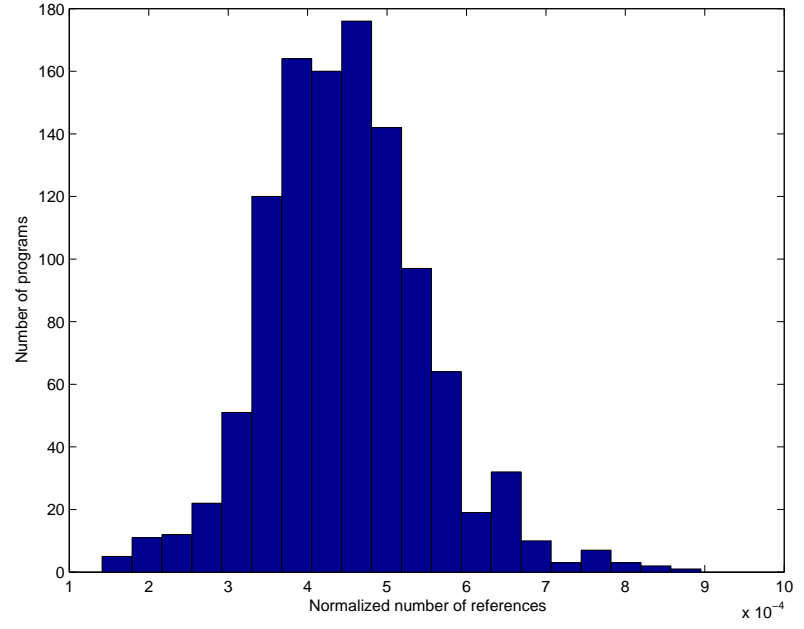
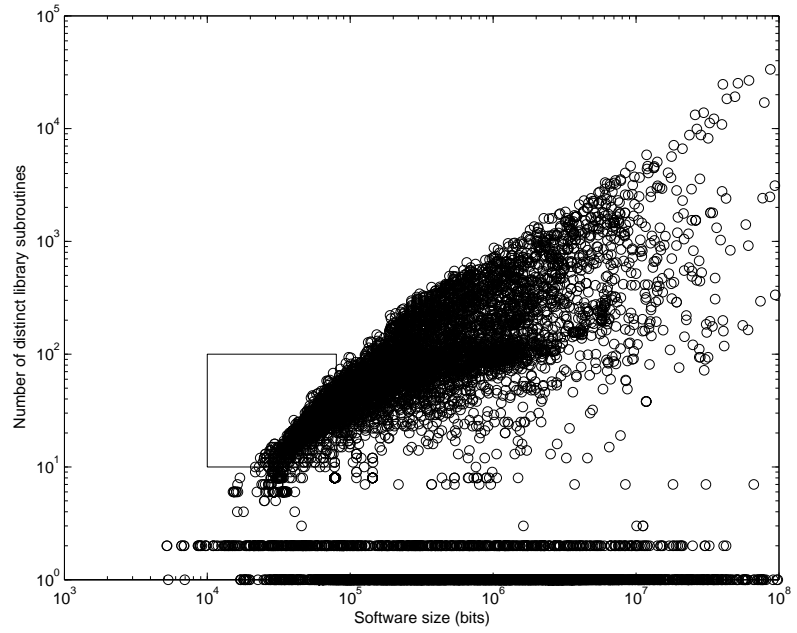
- the rate at which they “factor” something;
- their size/density (prime number theorem)

Are primes and software components two examples of a more general phenomenon?

## Erdős-Kac theorem

The number of prime divisors of an integer of  $n$  bits is normally distributed.

The number of library components used in a program of  $s$  bits looks like this—



## Practical Implications

- Blanket reuse prescriptions are misguided. Reuse targets must be pegged to  $H$  of the problem domain, and how mature the libraries are in the field.
- “Strong reuse” /CBSE can only succeed in domains with  $H \rightarrow 0$
- DSLs cannot reduce the size of programs, asymptotically; but domain-specific libraries *can*.
- Teaching progressions for libraries: optimal strategy is a “truncated codebook.” (Like vocabulary acquisition.)

- Library incompleteness: must plan for extensibility when designing libraries.
- Model fitting: can we measure  $H$  for a domain? Judge how close to “optimal reuse” a codebase is? Judge programmer maturity?

# Conclusions

- Info theory and K-complexity provides useful theory to model and predict reuse.
- This one parameter of a problem domain —  $H$  — tells us how much reuse is possible, and helps characterize different “reuse regimes”:
  - ★  $H \rightarrow 0$ : strong reuse, CBSE dream of “wiring big components”
  - ★  $0 < H < 1$ : moderate reuse; at most  $(1 - H)$  of code from libraries.
  - ★  $H \rightarrow 1$ : little to no reuse possible, should be prepared to create most of an application from scratch.