

## Software Reliability Engineering

**Florin Popențiu VLĂDICESCU**

Professor Florin Popențiu Vlădicescu graduated in Electronics and Telecommunications from University POLITEHNICA of Bucharest in 1974 and holds a PhD in Reliability in 1981. He has been Chairholder of the UNESCO Chair in Information and Communication Engineering at City University London since 1998 and also he has been appointed Director of the "UNESCO Chair" Department at University of Oradea. Professor Florin Popențiu Vlădicescu has published over 100 papers in international journals and conference proceedings and is co-author of three books.

He has worked for many years on problems associated with software reliability and has been Co-Director of two NATO Research Projects, involving collaboration with partner institutions throughout Europe. Also he is on the advisory board of several international journals, including "Reliability and Risk Analysis: Theory & Applications" and "Microelectronics Reliability", and is a reviewer for "ACM Computing Reviews". He is an expert for the Seventh Framework Programme-FP7. His Research ID is [E-5787-2010](#). Also in 2009 he has been nominated UNESCO Expert in the field of Higher Education, Research and Knowledge. Professor Popențiu Vlădicescu is currently Visiting Professor at the Paris Institute of Technology - "ParisTech" - which brings together 24 of France's best engineering and business schools. He also lectures at the Technical University of Denmark. He was elected Fellow of the Academy of Romanian Scientists in 2008 and Director of the Doctoral School "Engineering sciences" – November 2011.

**Course book**  
First Edition

Debrecen (HU)  
2012.

edited by Florin Popențiu VLĂDICESCU



# Software Reliability Engineering

**First Edition**

Course book of Series of  
Advanced Mechatronics Systems

Debrecen (HU)  
2012.

# Software Reliability

**Florin POPENTIU**

**University Politehnica of Bucharest**

**ENSTA ParisTech**

*popentiu@imm.dtu.dk*

*Fl.Popentiu@city.ac.uk*



# Software Reliability - Course Objectives

- Measuring software reliability
- Generic approach and specific models
- Evaluating predictions
- Capabilities and limitations
- Reliability assessment in the life-cycle
- Data collection
- Unanswered questions

# Software Reliability Course - Agenda

1. **Motivation**
2. Introduction to Software Engineering
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
7. Lessons Learned
8. Background Literature



# 1. Motivation

- Introduction
- Errors, faults and failures
- Faults and failures: examples
- Actual software disasters
- The relationship between faults and failures
- Case Study – Therac 25
- When are software faults introduced?
- What is reliability?
- What is dependability?
- Dependable systems
- Hancock's Half Hour!

# What is “Software”?

- An abstract digital definition of system behaviour.
- Made concrete by compilation and loading.
- Defines much of system internal state.
- Includes interfaces (H/W-s/W, HCI, etc.).
- Includes documentation.
- May be firmware, microcode, operating system, application.
- System may consist of H/W, S/W, or both.

# Why use software?

- Light
- Easily modified
- Supporting hardware very reliable
- Demand for “smart” systems

# How Hard is Software ?

Software is so hard that –

- £1 billion is spent annually in the UK on software



- Over half this is wasted!

**“Waste” includes:**

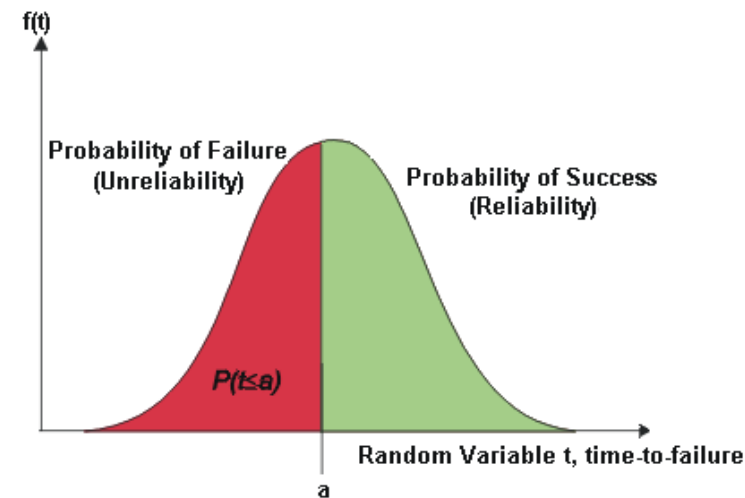
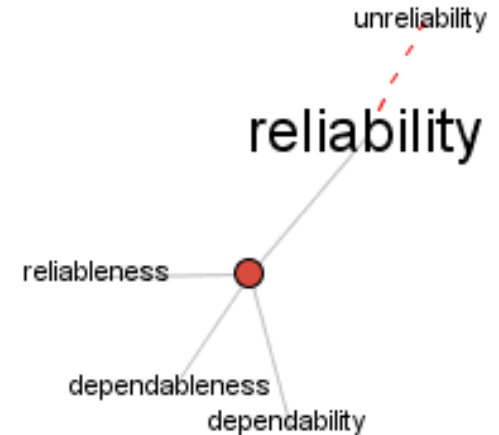
- Cost of overruns
- Cost of correcting errors
- Penalty due to not using quality assurance procedures
- “unnecessary” expenditure on maintenance.



# What is reliability?

## Two senses:

- **General:** The ability of a system to perform a required function under given conditions for a specific time interval.
- **Precise:** The probability that a system will operate without failure under given conditions for a specific time interval.




# The “Software Crisis”

**OVER due, OVER budget, UNDER quality**

- Society is increasingly dependent on complex digital systems
- They are delivered late
- They cost more than was planned
- They are not good enough
  - don't meet their functional requirements
  - unreliable, unsafe, insecure, unusable, not maintainable

**In particular, they are “undependable”**

- 
- **Software reliability** is rarely of concern to most people until something goes wrong.
  - Physical system components deteriorate over time, while software does not.
  - However, unlike a human operator, software does not adapt well to situations which were not anticipated by its designers, and such failures can prove enormously costly.

**SOFTWARE  
RELIABILITY 2012.**  
**THE LATEST  
DEVELOPMENTS  
IN SOFTWARE  
TECHNOLOGY.**

# Does Software Reliability Make Sense?

- *Developers, users and military organizations* are often concerned about the *reliability of systems that include software*.
- Over the years, reliability engineers have developed detailed and elaborated *methods of estimating the reliability of hardware systems* based on an estimate of the reliability of their components.
- Software can be viewed as one of those components, and an *estimate of the reliability of software* is *considered essential to estimating the reliability of the overall system*.



# Hardware/Software differences

- **Hardware is manufactured**
  - Designed once
  - Many imperfect copies
- **Software is “all design”**
  - Design transformed into code
  - Many perfect copies
- **Every software product is a “prototype”**

# Hardware/Software maintenance

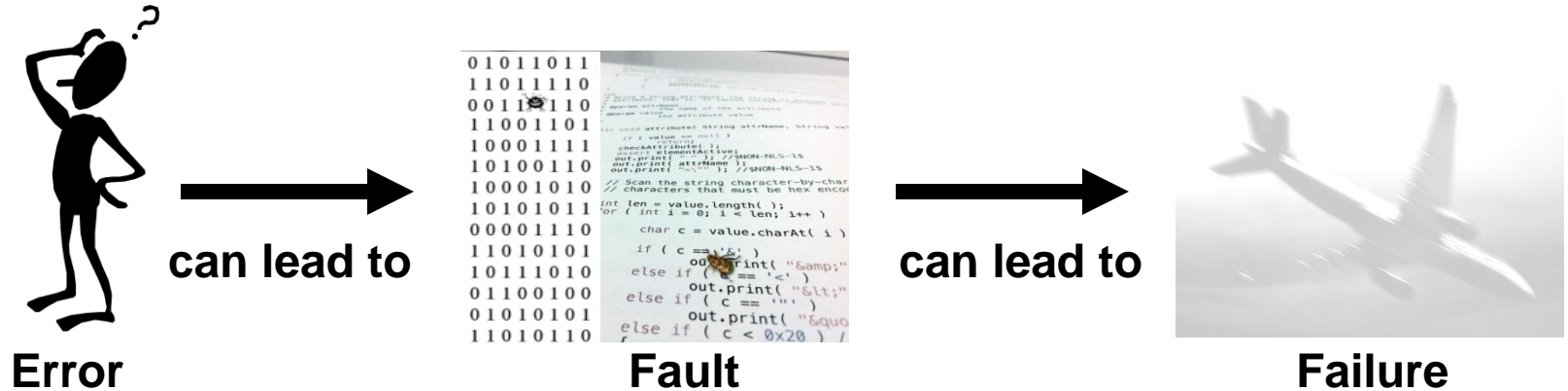
## Hardware maintenance

- System “down” until repaired
- Corrective maintenance restores system to “good as old”
- Design change difficult
- Requires movement of men and material

## Software maintenance

- “Transient” failures
- Corrective maintenance improves system design
- Design change easy
- Requires movement of information

# Errors, faults, and failures



- **Error: Designer's mistake**  
e.g. failure to distinguish signed and absolute value numbers in an algorithm
- **Fault: Encoding of an error into software**  
e.g. 'X:=Y' is coded instead of 'X:=ABS(Y)'
- **Failure: Deviation of the software from its specified delivery or service (incorrect output or timing of output)**  
e.g. nuclear reactor exhibits behaviour likely to be an earthquake hazard.

# Failures of Complex Systems (1)

## **“Physical” failure:**

- Hardware component breaks.
- Cause is physical (e.g., wear-out, overload, corrosion).
- “fault” appears in system at that point in time.
- Fault may cause failures unless “masked” by “redundancy”.
- System repaired by replacement of broken component.
- System is thereby restored to its previous good state.

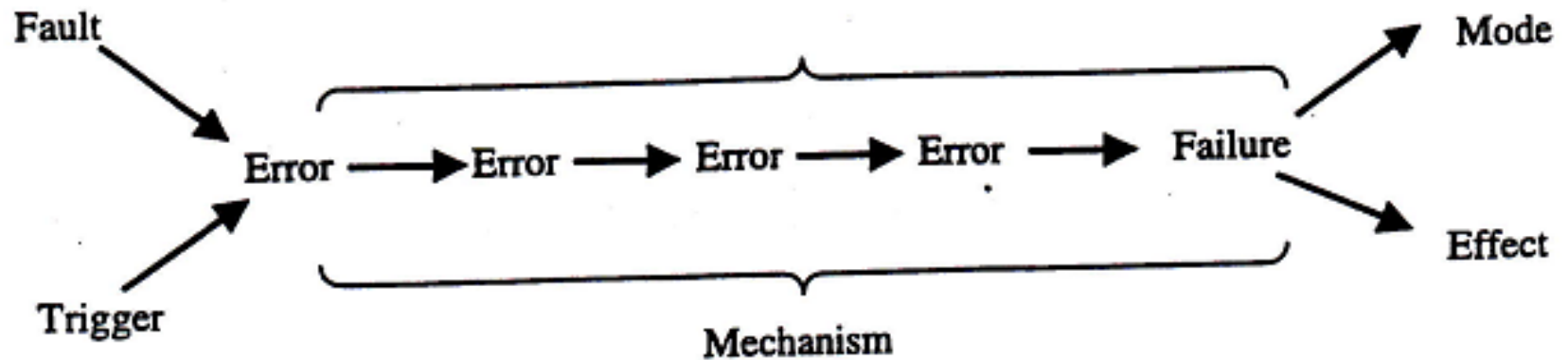


# Failures of Complex Systems (2)

## Design failure:

- defect in design
- cause is intellectual (bad requirement, careless design)
- fault is present in system, but “latent”
- may cause failure with some inputs or internal states
- repair by changing design
- system is different from its previous “bad” state

# System Failure



Cause → {  
Fault = something "wrong" with the system  
Trigger = circumstances which activate fault  
}

Error = incorrect internal state

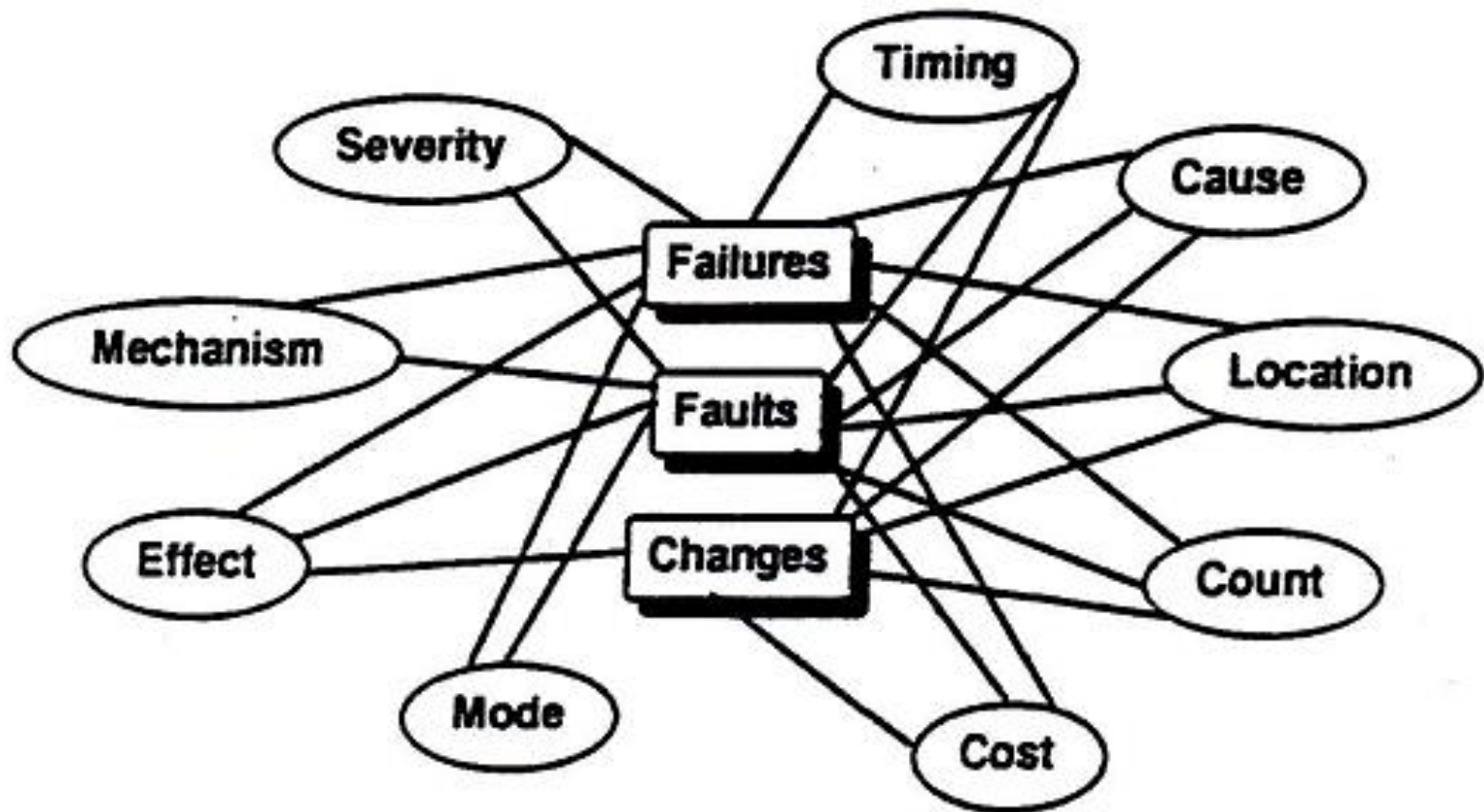
Mechanism = cause & effect propagation of error

Failure = departure from required behaviour

Mode = observable symptoms

Effect = consequence for environment

# Attributes of failures, faults and changes



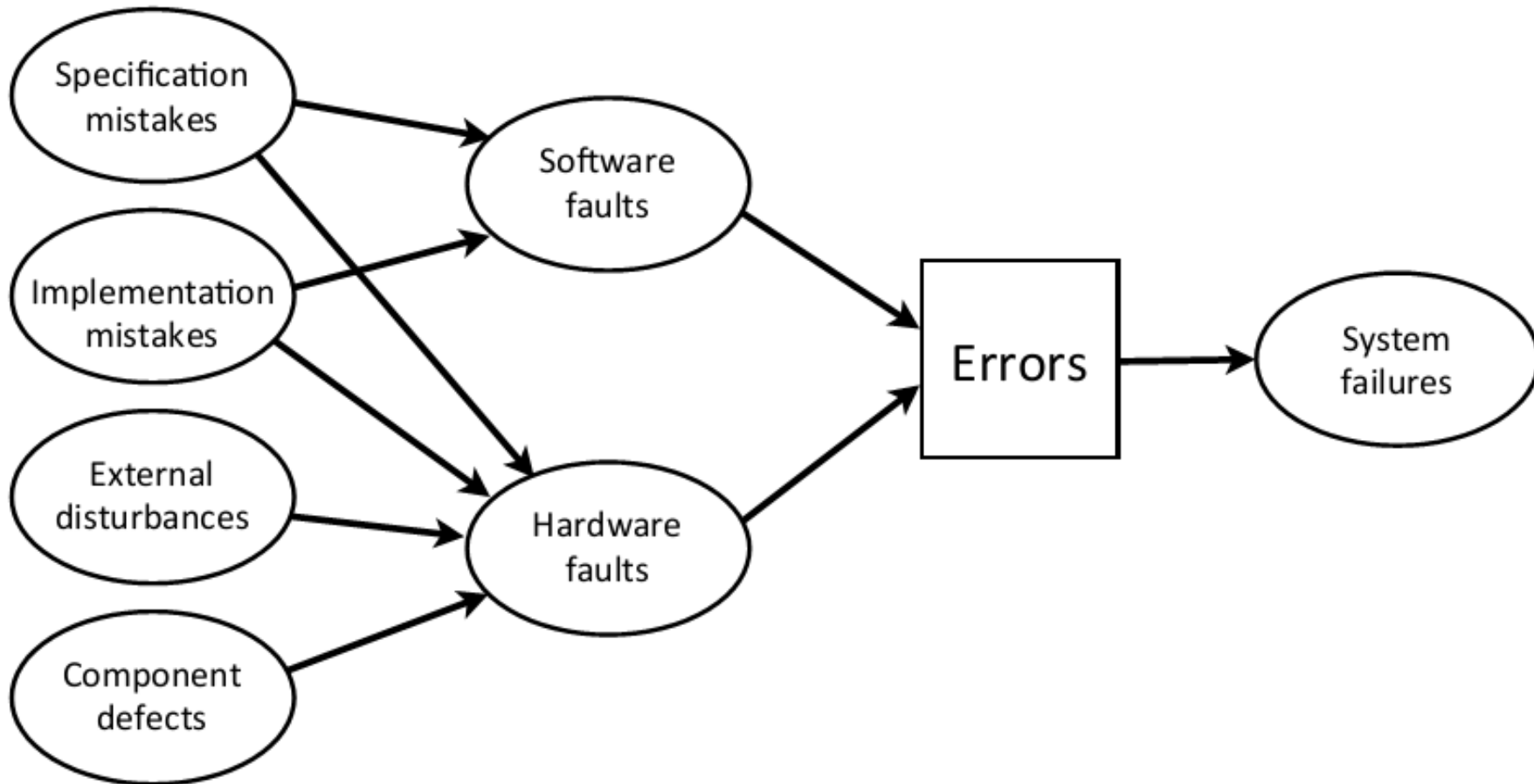
# Failures and faults

- A failure corresponds to unexpected run-time behaviour observed by a user of the software.
- A fault is a static software characteristic which causes a failure to occur.
- Faults need not necessarily cause failures. They only do so if the faulty part of the software is used.
- If a user does not notice a failure, is it a failure? Remember most users don't know the software specification.



# Causes of faults

- Problems at any stages of the design process can result in faults within the system.



# Causes of faults, cont.

## Specification mistakes

- Incorrect algorithms, architectures, hardware or software design specifications
  - Example: the designer of a digital circuit incorrectly specified the timing characteristics of some of the circuit's components

## Implementation mistakes

- Implementation: process of turning the hardware and software designs into physical hardware and actual code
- Poor design, poor component selection, poor construction, software coding mistakes
  - Examples: software coding error, a printed circuit board is constructed such that adjacent lines of a circuit are shorted together

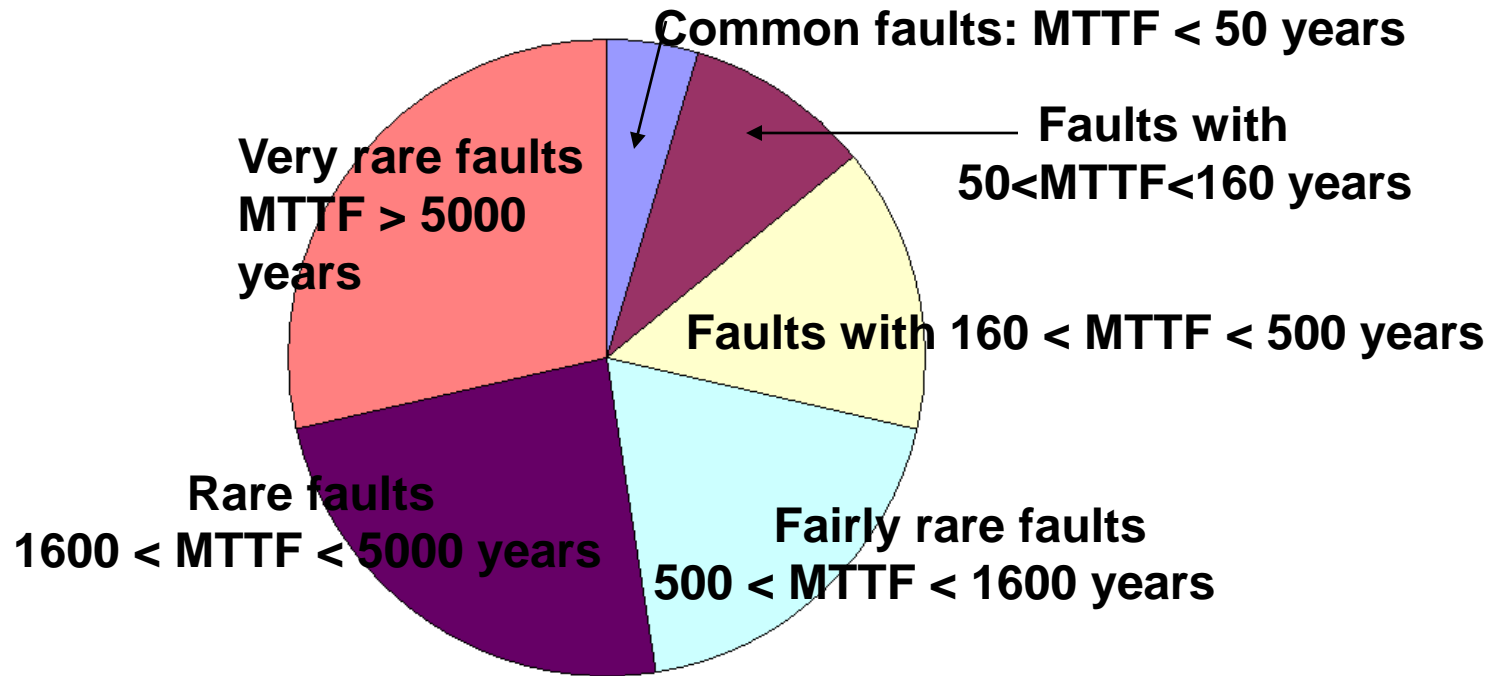
# Failure classification

| <i>Failure class</i> | <i>Description</i>                                   |
|----------------------|--|
| Transient            | Occurs only with certain inputs                      |
| Permanent            | Occurs with all inputs                               |
| Recoverable          | System can recover without operator intervention     |
| Unrecoverable        | Operator intervention needed to recover from failure |
| Non-corrupting       | Failure does not corrupt system state or data        |
| Corrupting           | Failure corrupts system state or data                |

# Failure consequences

- Reliability measurements do NOT take the consequences of failure into account.
- Transient faults may have no real consequences but other faults may cause data loss or corruption and loss of system service.
- May be necessary to identify different failure classes and use different measurements for each of these.

# The relationship between faults and failures



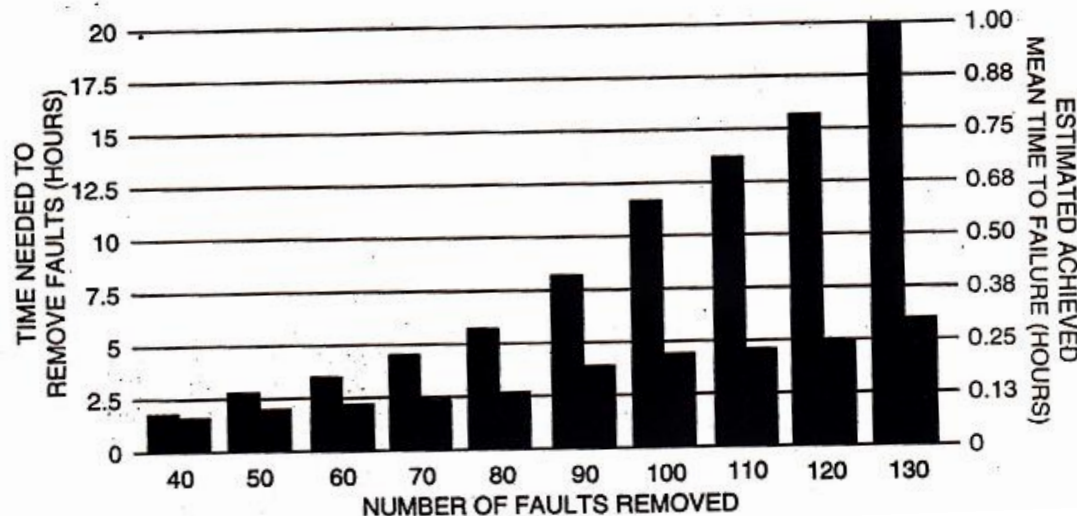
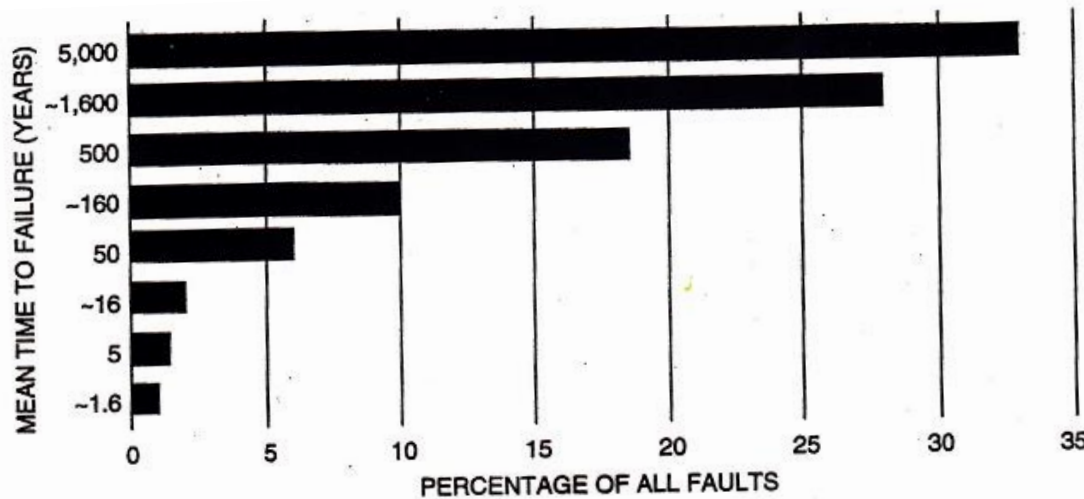
**(MTTF = Mean Time To Failure)**

**Sample from 9 major software products, each with many thousands of years logged use world-wide.**

Ref: Adams E., Optimizing preventive service of software products”, IBM J Research & Development.



# Diminishing returns



- Software faults persist even in well-debugged programs. Edward N. Adams of IBM found that bugs that remained in a system were primarily “5000-year” bugs – that is, each of them would produce a failure only once in 5000 years (top). Such faults make debugging an exercise in diminishing returns: in the test of a military command-and-control system (bottom), the time needed to remove the bugs begins to outpace by far the resulting improvement in the estimated reliability, measured in terms of estimated achieved MTTF.

For visual clarity, the graphs have been plotted on different time scales.

Source: Littlewood B., Strigini L., The Risks of Software, Scientific American, Nov., 1992, 62-75.



# Faults and failures: examples

- **Therac 25**

**Failure:** man killed by huge overdose of therapeutic radiation.

**Fault:** improper echoing of keyboard commands on VDU.

- **NASA probe to Venus**

**Failure:** probe went off course and was lost.

**Fault:** in the navigational FORTRAN code the statement .

DO 3I=1.3 was written instead of DO 3I=1,3.

- **A320**

**Failure:** An A320 crashed in India killing 92 people.

**Fault:** ?????

(see the story of the Ariane Flight 501, 1996:

<http://www.youtube.com/watch?v=IONcgYzVFlg>)

# Ariane 5 Rocket Explosion - 1996

## Loss

rocket: \$500 million

project: \$7 billion

## Description

- Intentionally destroyed 40 s after liftoff

## Cause

- Variable overflow because of reuse of software module (from Ariane 4) related to horizontal velocity measurement.
- This happened simultaneously in both the active and the back-up computer.



# Software disasters

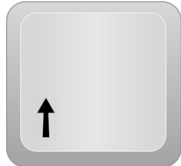
- **Therac 25**

- Software controlled radiation therapy.
- Software interlock governed high/low beam strength.
- Interaction of operator error and software fault -> high strength beam. without shield in place
- Killed 2, injured others.

- **Citibank (1989)**

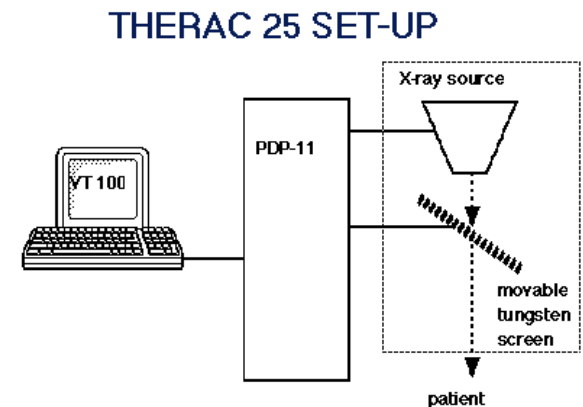
- Electronic funds transfer through CHAPS.
- Interaction of operator error and design fault -> wrong date supplied.
- Repeated previous day's transfers.
- Money recovered within 2 days!

# Case study: Therac 25

- Radiotherapy machine failure
  - 2 deaths, several injuries
- 2 modes of operation
  - X-ray: high-intensity beam strikes tungsten target
  - electron: low-intensity beam with target retracted
  - treatment programmed using monitor and keyboard
- The accidents
  - high-intensity beam, with target retracted
  - “Malfunction 54”
- The trigger
  - use of  to correct a typing error

# Therac 25 Failure

- Location: East Texas Cancer Center
- Timing: 21 March 1986 (#treatment hours unknown)
- Mode: 'Malfunction 54' on operator screen
- Effect: Beam strength too great by factor of 100
- Mechanism: Use of up arrow key corrupted internal software variable
- Cause: Unintentional design fault
- Severity: Critical (loss of life)
- Cost: Financial loss in  
litigation/investigation



**You cannot predict what you cannot measure!**



# Patriot Missile System - 1991

## Loss

28 soldiers dead

100 injured

## Incident

- The system failed to intercept an incoming missile

## Cause

- Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer (e.g., 32, 33, 34...)
- The Patriot battery had been up around 100 hours, resulting in an precision error of 0.34 seconds





# IC4 Marslev incident - 2011

## Loss

- Tens, maybe hundreds of millions DKK

## Description

- On 14 November 2011 DSB stopped operations with all IC4 trains after an IC4 train overran a red stop signal at Marslev

## Cause

- Slippery rails and “inefficient” braking system
- See the DTU report



# Amazon Cloud Outage - 2011

## Loss

- difficult to quantify, but many Web 2.0 websites were affected for hours, days

## Description

- Amazon EC2 services were unavailable for a hours
- Some permanent data loss

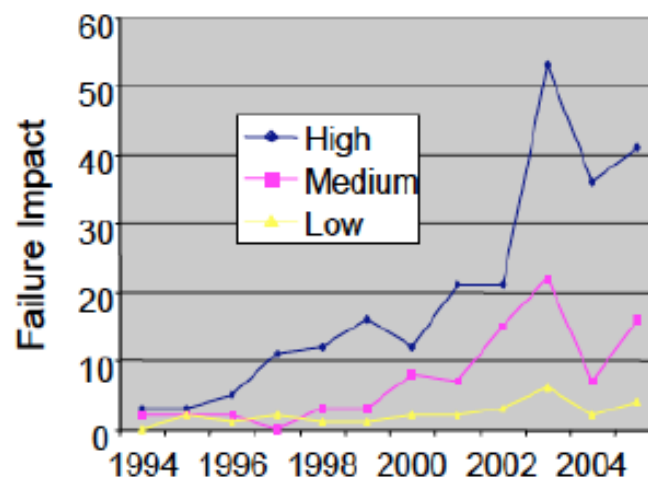
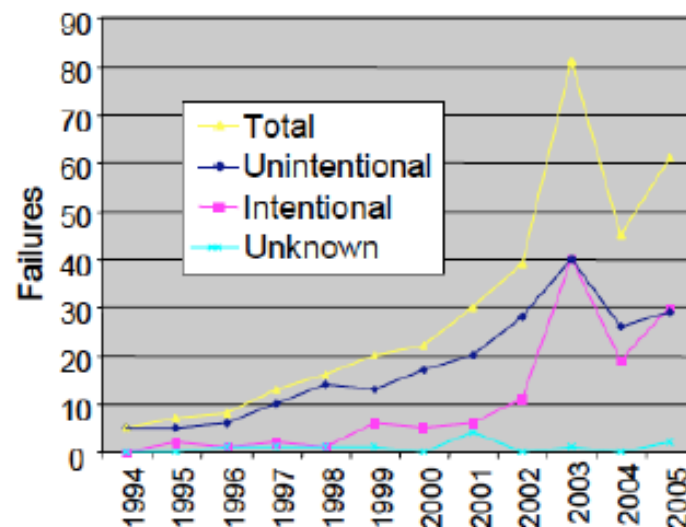
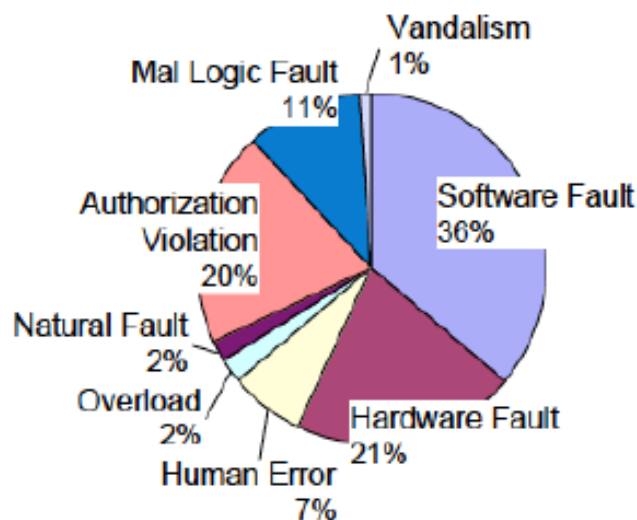
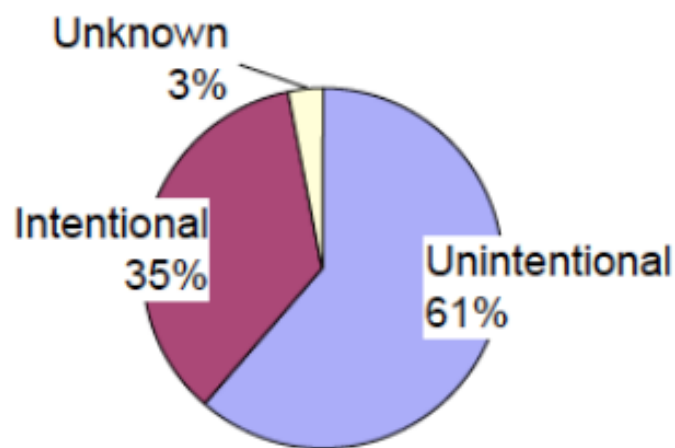


## Cause

- “The trigger for this event was a network configuration change.”

# Analysis of 347 computer-related infrastructure failure cases

[Rahman, Beznosov, Marti, "Identification of sources of failures and their propagation in critical infrastructures from 12 years of public failure reports", Int. Journal on Critical Infrastructures, vol.5, n°3, 2009]



# Why is Software So Bad?

**It is essentially difficult:-**

- Novel
- Complex
- Discontinuous
- “Invisible”
- Hard to predict
- Hard to measure

**It is also NOT ENGINEERED**



# Novelty by design

(1)

## Novel designs give nasty surprises!

- “Traditional” artefacts “evolve” gradually:-
  - bridges: standard designs “off the shelf”
  - cars: most design features go back 100 years
  - even so, disasters occur: Tacoma, Tay
- Software has been around for only 60 years:-
  - few “standard” design
  - frantic rate of change

# Novelty by design

(2)

## Novel designs give nasty surprises!

- New problems generate new solutions
- Every software product is a “prototype”:-
  - unique, even though many identical copies exist
  - each program is only developed ONCE (unlike widgets: developed once, produced in millions)



# Complexity

## **The human mind cannot cope with complexity**

- Software is the most complex thing ever made
- Difficult to visualise
- Unstructured
- 10.000.000 lines of source is common
- Absorbs most intellectual effort in system development

# Discontinuous Behaviour

## Software is discrete

- Billions of internal states
- Most of them can give rise to failure
- Impossible to test exhaustively
  - too many paths
  - too many internal states
  - too many input cases
  - non-deterministic behaviour
- When things go wrong, they go VERY wrong

# The “Craft” Approach

**Software has tended to be a cottage industry:-**

- **Produced by creative effort**

- more like writing a novel than engineering
- written by geniuses for geniuses
- “How dare you criticise my creation?”

- **Invisible**

- “Why should you read my code?”
- “Of course its all right! I wrote it! I tested it!”
- “You’re crushing the butterfly’s wing of my creativity!”

- **This is WRONG attitude for an engineer!**

- Weinberg: “Ego – less programming”
- Professional maturity
- Even you can make mistakes!

# What is “dependability”?

*Defined as:*

“The extent to which the user can justifiably depend on the service delivered by a system.”

*J-C. Laprie: Dependability: basic concepts and terminology*

Important concepts: “required service”, “user”, “system”

“Umbrella” term: not measurable attributes

Different authors use different sets of attributes  
(Laprie, BS5760)

# Definitions of dependability attributes

## “RAMURSES”

| Attribute ...           | ... defined as ability of a system to ...  |
|-------------------------|--|
| <b>Reliability:</b>     | <i>... deliver required service</i>  |
| <b>Availability:</b>    | <i>... be in an “up” state</i>   |
| <b>Maintainability:</b> | <i>Corrective... be repaired to remove faults</i><br><i>Adaptive ... be modified for new environment</i><br><i>Perfective ... be enhanced to improve service</i> |
| <b>Usability:</b>       | <i>... provide ease of access for user</i>   |
| <b>Recoverability:</b>  | <i>... resume service after failure</i>  |
| <b>Safety:</b>          | <i>... be used without accident</i>  |
| <b>Efficiency:</b>      | <i>... complete task within given resources</i>  |
| <b>Security:</b>        | <i>... resist unauthorised interference</i>  |

# Dependable systems

We need depend on systems. They must be:

|                      |   |
|----------------------|---|
| <b>Reliable:</b>     | <i>Deliver the required service under given time.</i>   |
| <b>Safe:</b>         | <i>Must not kill people.</i>  |
| <b>Secure:</b>       | <i>Must not allow unauthorised access.</i>  |
| <b>Usable:</b>       | <i>Must be “friendly”: easy to learn and use.</i>   |
| <b>Maintainable:</b> | <i>Must be quick to recover after failure, and easy to repair so that they do not fail again.</i> |
| <b>Available:</b>    | <i>Must be ready for use a high proportion of time.</i>   |
| <b>Extendable:</b>   | <i>Must be easy to enhance to perform new functions.</i>  |



# How to make software-based systems more dependable

|  |  |
|--|--|
| <b><i>Fault avoidance:</i></b>           | Good management<br>Disciplined process<br>Careful requirements capture & design                                    |
| <b><i>Fault removal:</i></b>             | Design and code inspection<br>Static analysis<br>Testing   |
| <b><i>Fault tolerance:</i></b>           | Defensive design: “belt & braces”<br>Diverse redundant modules<br>Manual back-up                                   |
| <b><i>Dependability measurement:</i></b> | Realistic trial<br>Collect data (failures, faults, operating time)<br>Analyse data to evaluate “ <b>RAMURSES</b> ” |

# Dependability - Resilience

**Dependability:** ability to deliver service that can justifiably be trusted

## Threats

(accidental, malicious)

... ➡ Failures ➡ Faults ➡ Errors ➡ Failures ➡ Faults ➡ ...

## Attributes

Availability Reliability Safety Confidentiality Integrity Maintainability

## Means

Fault Prevention Fault Tolerance Fault Removal Fault Forecasting

**Resilience:** The persistence of dependability when facing changes

# Safety Integrity

## Definition

- **Safety integrity** is the likelihood of a safety-related system satisfactorily performing the required safety functions under all stated conditions within a stated period of time.

## Captured using **Safety Integrity Levels (SILs)**

- SILs are a measure of the required protection against failure
- SILs are assigned to the safety requirements in accordance with target risk reduction
- Used to determine what methods and techniques should be applied (or not applied) in order to achieve the required integrity level

# IEC 61508 Standard

IEC 61508 is intended to be a basic functional safety standard applicable to all kinds of industry.

- The standard covers the complete safety life cycle, and may need interpretation to develop sector specific standards. It has its origins in the process control industry sector.

Can be tailored to different domains (automotive, chemical)

Comprehensive

Includes SILs, including failure rates

Covers recommended techniques

## Notes:

- IEC = International Electrotechnical Commission
- E/E/PES = electrical/electronic/programmable electronic safety related systems

# Safety-Integrity Table of IEC 61508

| Safety Integrity Level | Low demand mode of operation<br>(Average probability of failure to perform its design function on demand) |                      |
|------------------------|---|----------------------|
| 4                      | $\geq 10^{-5}$ to $< 10^{-4}$   | (> 99.99 % reliable) |
| 3                      | $\geq 10^{-4}$ to $< 10^{-3}$   | (> 99.9 % reliable)  |
| 2                      | $\geq 10^{-3}$ to $< 10^{-2}$   | (> 99% reliable)     |
| 1                      | $\geq 10^{-2}$ to $< 10^{-1}$   | (> 90% reliable)     |

| Safety Integrity Level | High demand mode or continuous mode of operation<br>(Probability of dangerous failure per hour) |  |
|------------------------|---|--|
| 4                      | $\geq 10^{-9}$ to $< 10^{-8}$   |  |
| 3                      | $\geq 10^{-8}$ to $< 10^{-7}$   |  |
| 2                      | $\geq 10^{-7}$ to $< 10^{-6}$   |  |
| 1                      | $\geq 10^{-6}$ to $< 10^{-5}$   |  |

The higher the SIL, the harder to meet the standard

High demand for e.g. car brakes, critical boundary SIL 3

Low demand for e.g. airbag, critical boundary is SIL 3, one failure in 1000 activations



# Hancock's Half Hour

*Tony Hancock:* “But I’ve just been throttled half to death by a flamin’ python! Why won’t the insurance company pay out?”



*Sid James:* “Well, you see, they only insured you against accident, but the snake meant it!”





# Software Reliability Course - Agenda

1. Motivation
- 2. Introduction to Software Engineering**
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
7. Lessons Learned
8. Background Literature

## 2. Introduction to Software Engineering

- The phases of a software project
- Types of software
- Achieving software reliability: diverse sources of information
- Products, processes, and resources
- Achieving software reliability: diverse approaches

# What is “Software Engineering”

- Application of mathematical, scientific, organisational principles
- Concept, design, implementation, maintenance
- Achieve adequate quality with given time and resources
- Large projects

# The Phases of a Software Project

- Enthusiasm
- Disillusionment
- Panic
- Collapse
- Search for the guilty
- Punishment of the innocent
- Rewards and honours for those not involved

# Types of software

- Games
- Working machine programs
- Operating systems
- Commercial applications
- Process control
- Embedded military & nuclear
- Flight-critical ('Fly-by-wire', e.g. A330-340)

What is 'quality' in each case?

'Quality is conformance to requirements'. Philip B. Crosby

'Quality is free, but only to those who are willing to pay heavily for it' [T. DeMarco and T. Lister](#), *Peopleware : Productive Projects and Teams*, 2nd Ed. by Tom Demarco, Timothy R. Lister , ISBN: 0932633439

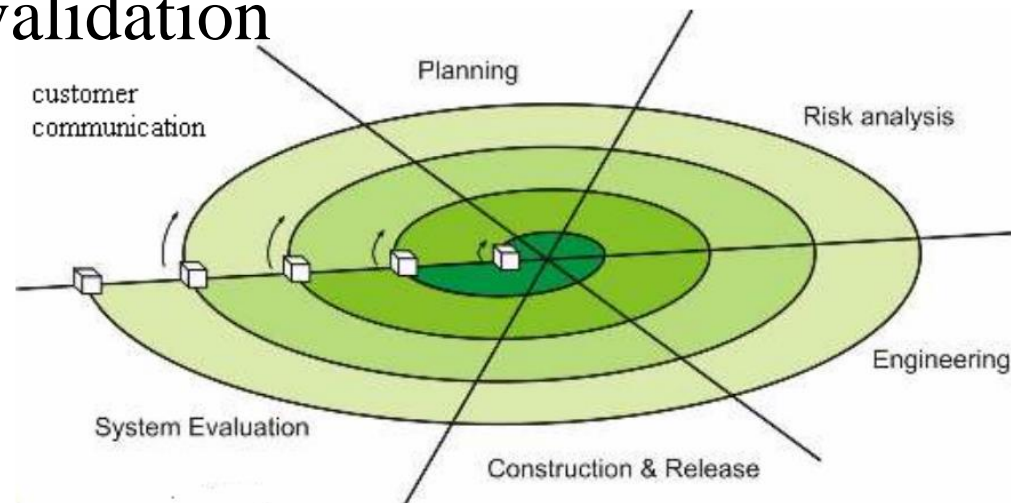
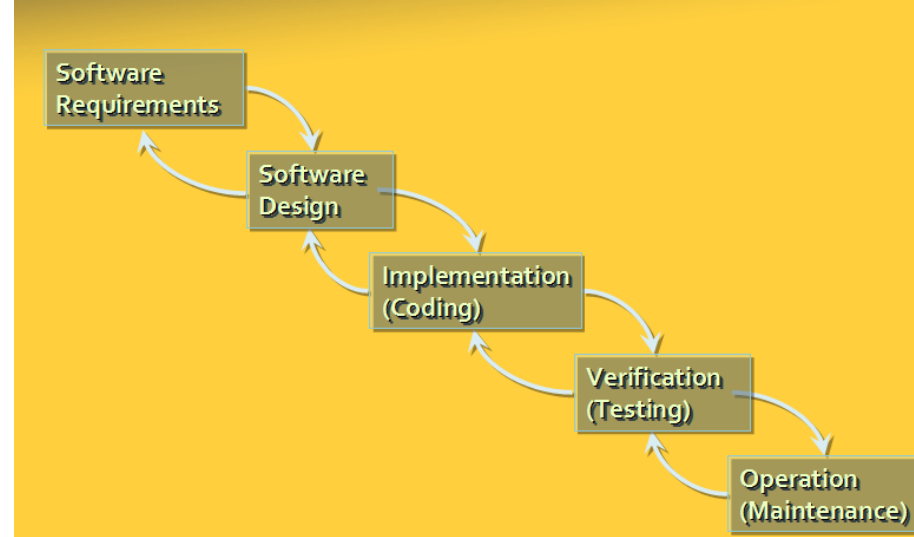
So what are the requirements?

'Horses for courses!'

# Software Development Phases

- Concept
- Software Requirements
- Top Level Design
- Detailed Design
- Code/implementation
- Unit/development Test
- System/verification/validation Test
- Operational Test
- Operation

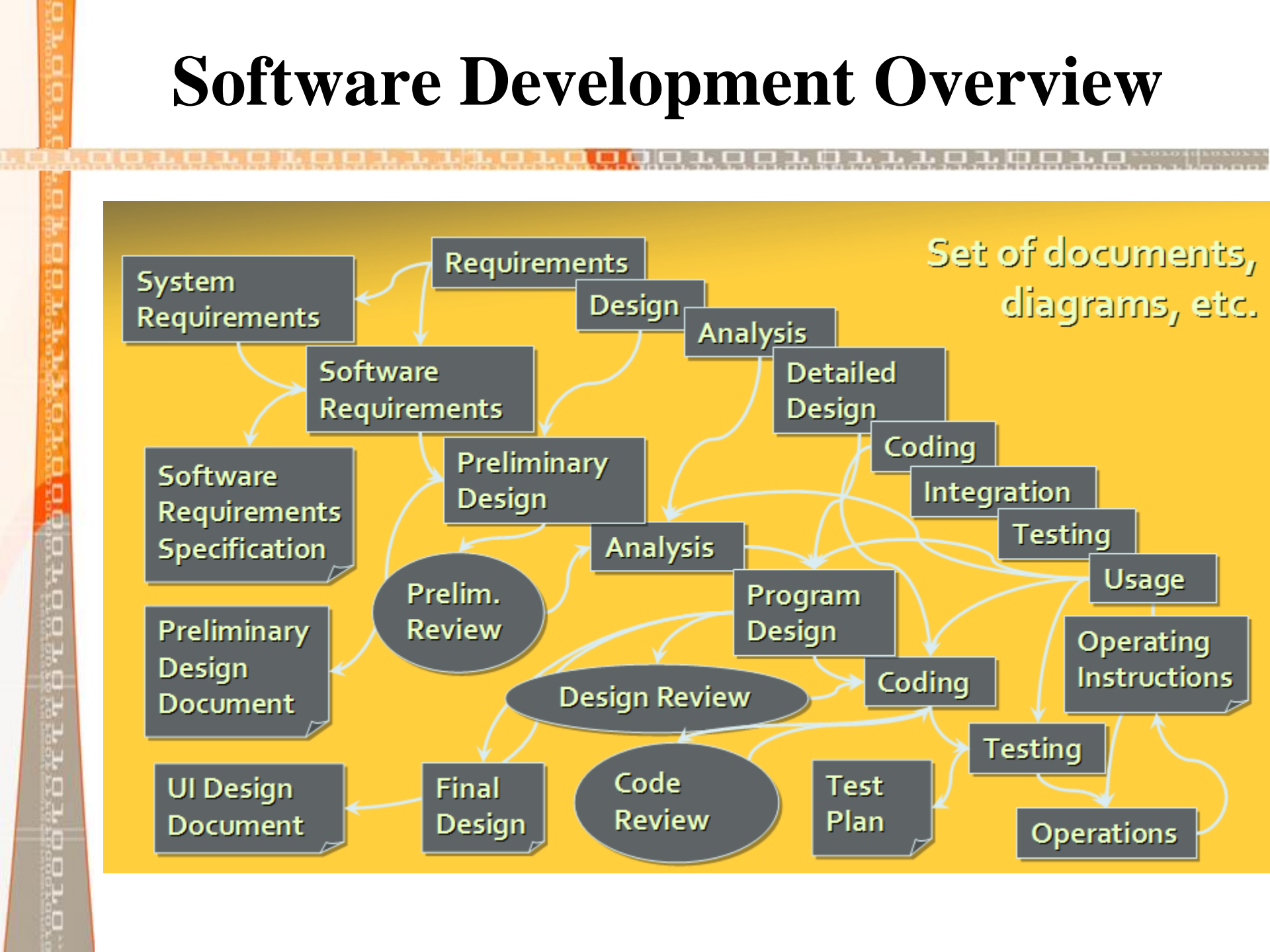
The waterfall development process:





# Software Development Overview

Set of documents, diagrams, etc.



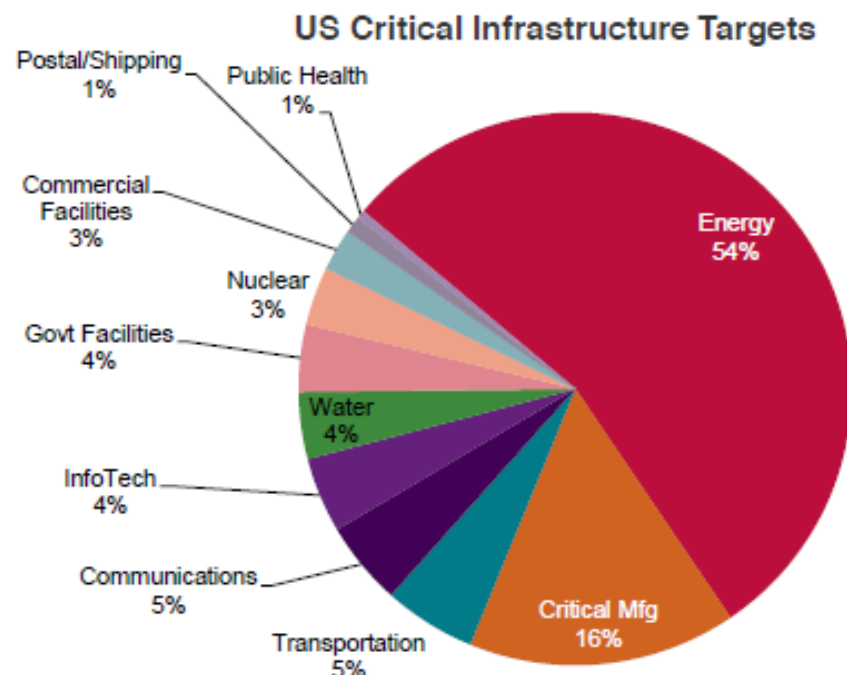
# When are software faults introduced?

|                                |   |
|--------------------------------|---|
| <b>Requirements:</b>           | Gripen: “control laws”, ICL (Fujitsu Service)<br>“usability problems” |
| <b>System design:</b>          | Therac: no fault-tolerance (H/W interlock)                            |
| <b>Software specification:</b> | Mariner: missing “bar” in mathematical formula                        |
| <b>Coding:</b>                 | Mercury orbiter: “dot for comma”                                      |
| <b>Compilation:</b>            | Pascal compilers: -ve exponential, floating point I/P                 |
| <b>Maintenance:</b>            | LAS(2): London Ambulance Service “memory leak”                        |

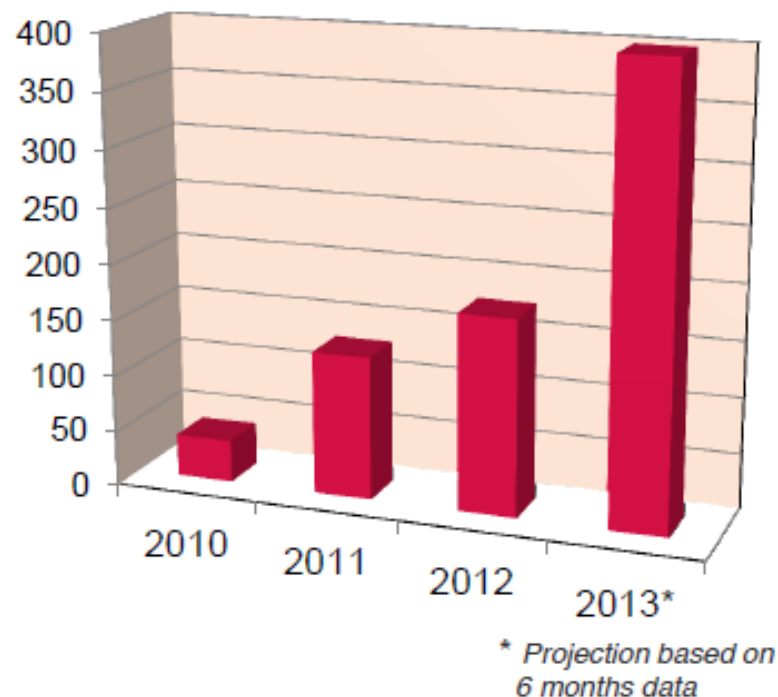
**Requirements faults give most serious failures but ...**

**... can we ensure against failures that are “deliberate”?**

# Cyber attacks: the threat is real



**Reported Attacks on US Critical Infrastructure**



**Source: US Dept of Homeland Security ICS-CERT**

John S. Kendall, Potential Impact of Cyber Attacks on Critical Infrastructure, Unisys

# System Requirements Analysis

- **Establish need and feasibility**
  - Overall functional requirements
  - Dependability requirements
  - Cost and schedule constraints
- **Subsystem/component breakdown**
  - Identify system elements
  - Define the process carried out by each
  - Define interfaces
  - Apportion dependability among elements
- **Identify software elements**
  - Define software FRs and NFRs
  - Initial statement of software requirements
  - Develop software requirements specification

# Reliability specification

- Reliability requirements are only rarely expressed in a quantitative, verifiable way.
- To verify reliability metrics, an operational profile must be specified as part of the test plan.
- Reliability is dynamic – reliability specifications related to the source code are meaningless.
  - No more than  $N$  faults/1000 lines
  - This is only useful for a post-delivery process analysis

# Specification validation

- It is impossible to empirically validate very high reliability specifications.
- No database corruption means PODOF of less than 1 in 200 million.
- If a transaction takes 1 second, then simulating one day's transaction takes 3.5 days.
- It would take longer than the system's lifetime to test for reliability.



# Steps to reliability specification

- For each sub-system, analyse the consequences of possible system failures.
- From the system failure analysis, partition failures into appropriate classes.
- For each failure class identified, set out the reliability using an appropriate metric. Different metrics may be used for different reliability requirements.

# Examples of a reliable specification

| <b><i>Failure class</i></b>  | <b><i>Example</i></b>  | <b><i>Reliability metric</i></b>                                     |
|------------------------------|--|--|
| Permanent,<br>Non-corrupting | The system fails to operate with any card which is input.<br>Software must be restarted to correct failures. | ROCOF<br>1 occurrence/1000 days                                      |
| Transient,<br>Non-corrupting | The magnetic stripe data cannot be read on an undamaged card which is input                                  | PODOF<br>1 in 1000 transactions                                      |
| Transient,<br>corrupting     | A pattern of transactions across the network causes database corruption                                      | Unquantifiable!<br>Should never happen in the lifetime of the system |

# Reliability and formal methods (1)

- The use of formal methods of development may lead to more reliable systems as it can be proved that the system conforms to its specification.
- The development of a formal specification forces a detailed analysis of the system which discovers anomalies and omissions in the specification.
- However, formal methods may not actually improve reliability.

# Reliability and formal methods (2)

- The specification may not reflect the real requirements of system users.
- A formal specification may hide problems because users don't understand it.
- Program proofs usually contain errors.
- The proof may make assumptions about the system's environment and use which are incorrect.

# IV & V

## • Independent Verification and Validation

- Two contractors: developer and monitor
- Commercially independent
- Monitor has no vested interest in delivery
- Access points contractually defined

## • Advantages

- Checker is well-motivated
- Diversity of approach

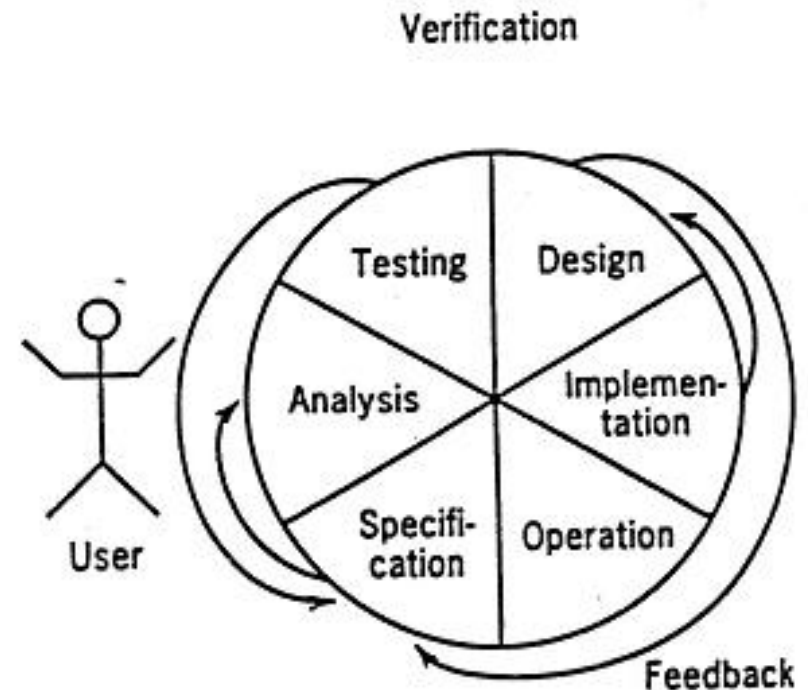
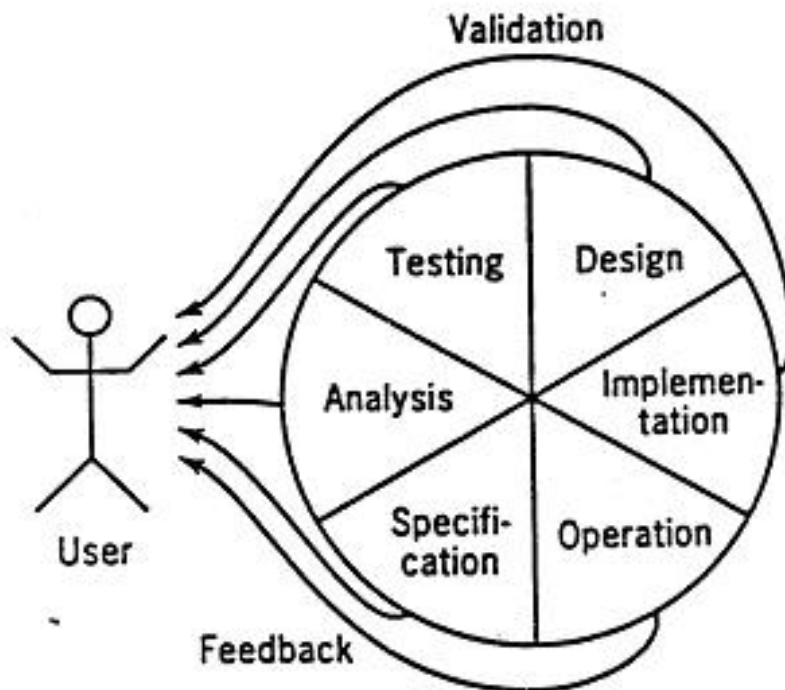
## • Disadvantages

- Expensive (up to 60% on contract)

# VALIDATION AND VERIFICATION

**Verification:** Are we building the product *right*?

**Validation:** Are we building the *right* product?





# Verification & Validation

The System may have been **verified**;

*Have we built the system “correctly”?*

But inadequately **validated**?

*Have we built the “correct” system?*

This is often satirised by customer’s remark:

*It’s just what I asked for, but not what I want!*

# Testing Strategies

- **Dynamic testing Vs Static testing (static analysis)**
  - in dynamic testing, the test data is executed on real machine
- **Black Box Testing Vs White Box Testing**
  - in black box testing, test cases are derived from the specification or requirements without reference to the code itself or its structure
  - in white box testing, test data are derived from the internal program structure
- **Testing Random**
  - using test cases in which all the test data are random

# Testing has many purposes...

- Reliability testing – measuring reliability
- Acceptance testing – fit for delivery?
- Unit testing – modules working on isolation?
- Integration testing – modules working as a system?
- Etc...

**... but only one goal**

## **To discover faults**

- A successful test is one which establishes the presence of one or more faults in the software being tested.

# Remember ...

- ... testing aims to find faults.
- ... testing is finished when the acceptance criteria have been met – not when the time runs out.
- ... the importance of test specification and planning.

# FAGAN INSPECTION

## **M.E. Fagan, IBM**

- Hardware inspection methods applied to software
- In use since early 70's
- Shown to be effective
- Larger award to originator

### **Highly formalised**

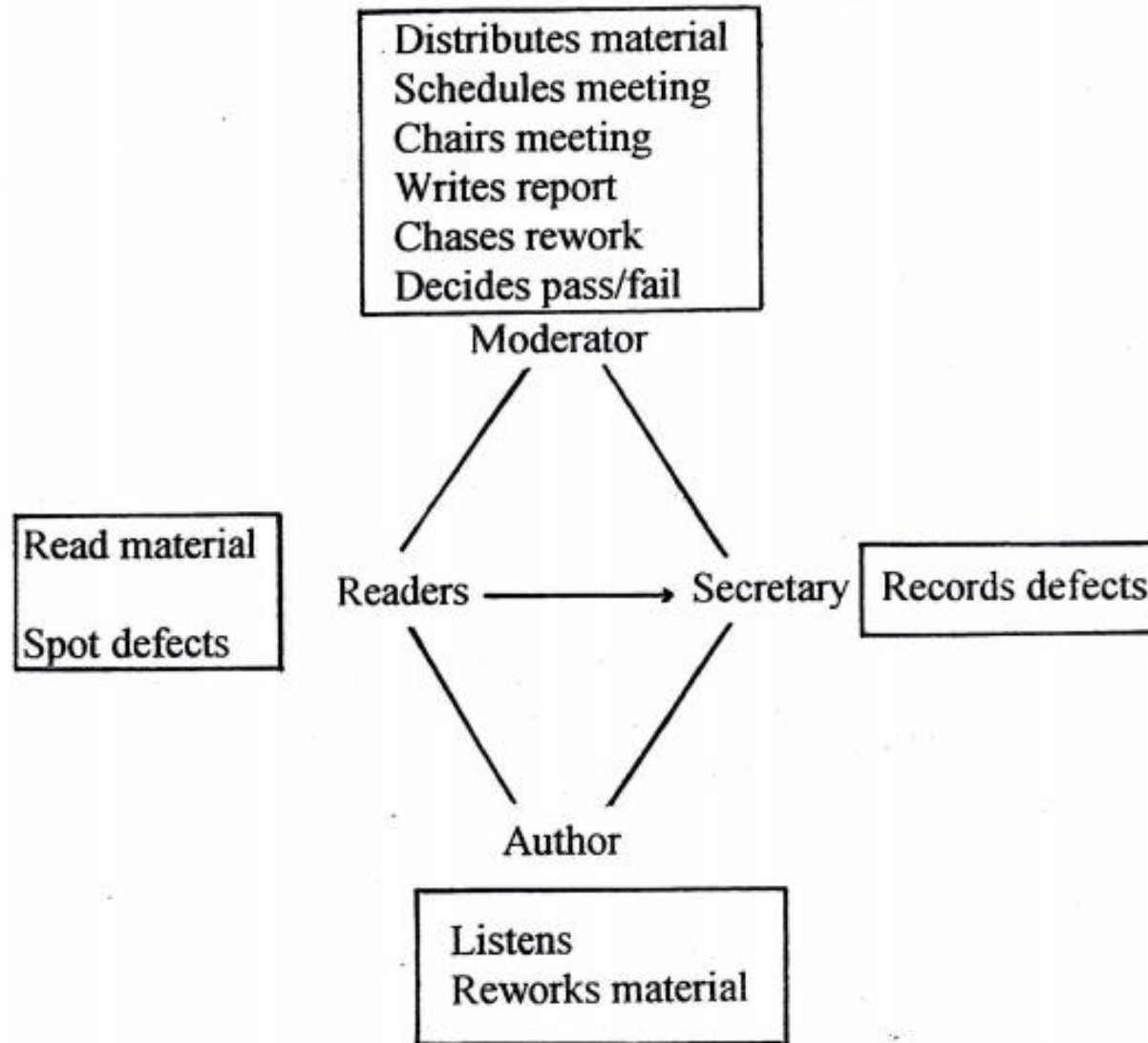
- Formal committee
- Two readers, independent of author
- Record defects, don't argue about repairs
- Declined preparation time
- Defined rate of reading
- Defined pass criteria

### **Generates statistics**

- Defects found in each module
- Defects found per K lines
- Defects found at each inspection
- Estimate efficiency of detection
- Estimate remaining defects/KLOC



# FAGAN INSPECTION TEAM



# Testing after development

- Acceptance testing
  - Completed system Vs requirements of real user
- Alpha test
  - User and developer test system using real data
- Beta test
  - Release of product to a section of the market for real use
- Installation testing
  - Tests to check on the installation process
- During use
  - Using spare capacity to do additional automatic testing

# SOFTWARE COST AND SCHEDULE

## Project problems:

|  |   |
|--|---|
| Cost estimation difficult                      | <ul style="list-style-type: none"><li>- main cost is of skilled effort</li><li>- “90%” syndrome</li><li>- “gutless estimation”</li><li>- effort <math>\propto</math> size</li><li>- How to estimate size?</li></ul>   |
| Schedule estimation difficult                  | <ul style="list-style-type: none"><li>- “90%” syndrome</li><li>- “schedules <math>\sqrt[3]{\text{effort}}</math> ”?</li><li>- How to estimate effort?</li><li>- “Putting more people on a late project makes it later!”</li><li>- Q: How does a project get 1 year behind schedule?<br/>A: One day at a time!</li></ul> |
| Planning s/W development is <b>NOT TRIVIAL</b> |   |
| “90%” syndrome                                 | The first 90% of the project consumes the first 90% of the budget, the remaining 10% of the project consumes the remaining 90% of the budget”   |

## Scheduling Problems



Managers find scheduling increasingly difficult.  
Large software projects are often:

- Late
- Over budget.



## *Bad Quality Software*



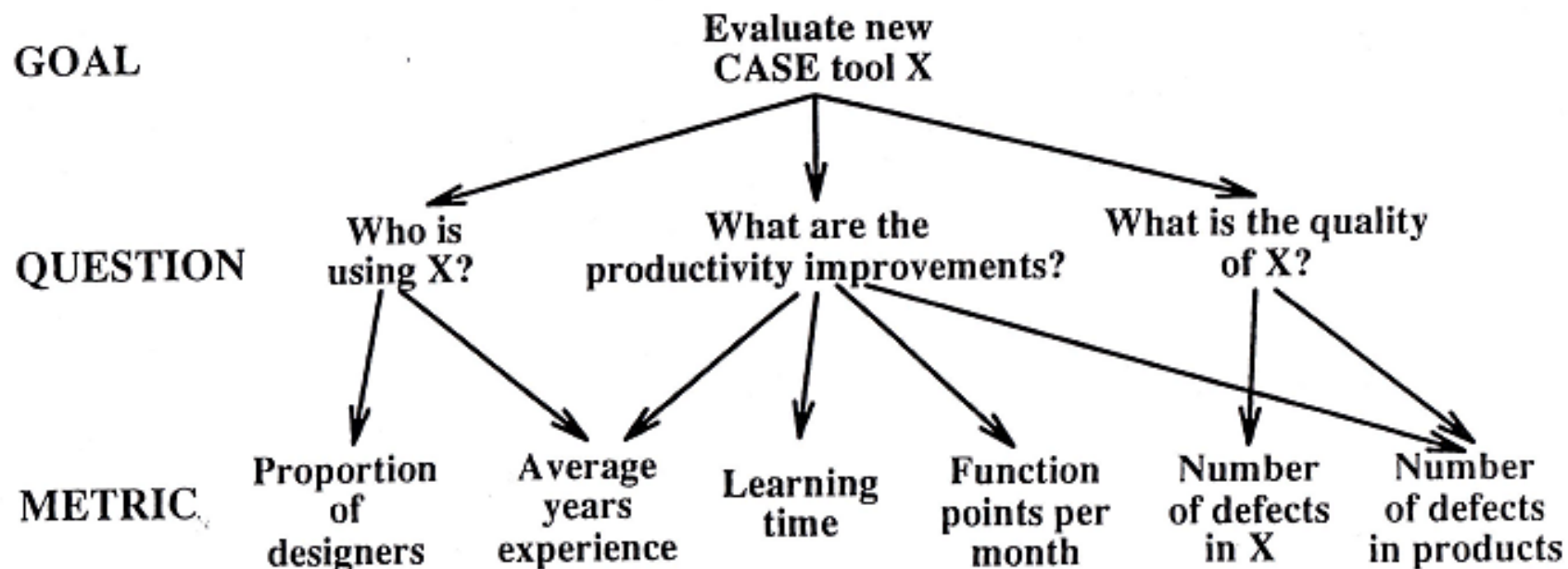
Software is all too frequently:

- unreliable
- unsuited to the user's needs.





# Goal/Question/Metric (GQM)



Basili VR, Rombach HD, 'The TAME project: Towards improvement-oriented software environments', IEEE Trans Softw Eng 14(6), 1988, 758-773



# Cost, Schedule, Quality

**Managing any project is a juggling act:**

|                        |  |
|------------------------|--|
| <b><i>Cost</i></b>     | <ul style="list-style-type: none"><li>-Resources are always finite</li><li>-Plant, raw materials, effort</li><li>-Job must be done within budget</li></ul>                       |
| <b><i>Schedule</i></b> | <ul style="list-style-type: none"><li>-Time is limited</li><li>-Delivery dates, market opportunities</li><li>-Job must be done by deadline</li></ul>                             |
| <b><i>Quality</i></b>  | <ul style="list-style-type: none"><li>-Nothing is ever perfect</li><li>-Reliability, functionality, shininess</li><li>-Product must be good enough (“fit for purpose”)</li></ul> |

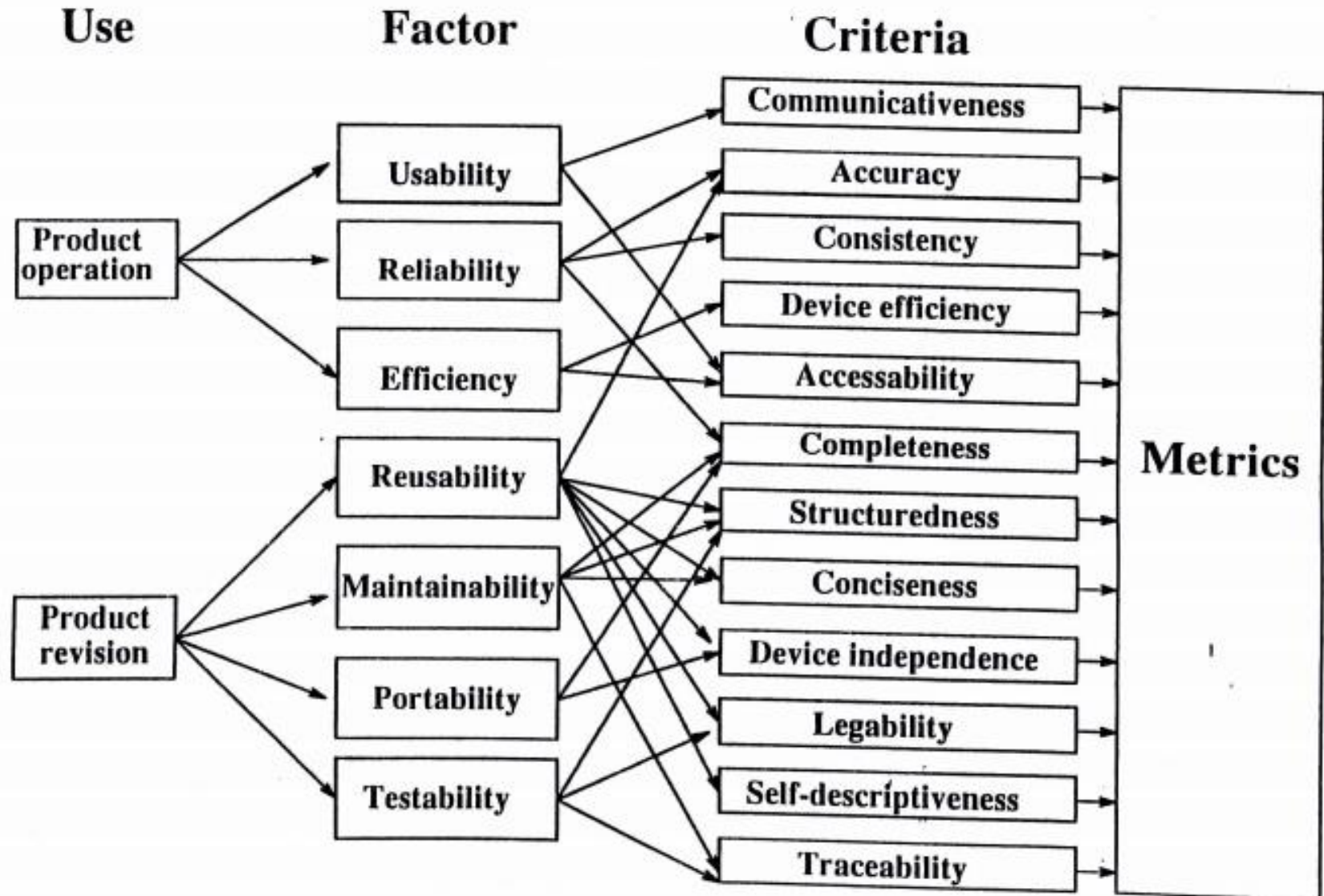
**Deliver adequate quality, on time, within budget**

# SEI – Capability Maturity Model

- CMM was developed by Software Engineering Institute and it is a strategy to improve software quality by improving the process by which software is developed. The five levels of CMM and their characteristics are given below:

| <b>Maturity Level</b>                | <b>Characterization</b>   |
|--------------------------------------|---|
| <i>Maturity Level 1 (Initial)</i>    | <i>Adhoc process:</i> Cost, schedule and quality are unpredictable                      |
| <i>Maturity Level 2 (Repeatable)</i> | <i>Basic Project Management:</i> Planning and tracking can be repeated                  |
| <i>Maturity Level 3 (Defined)</i>    | <i>Process Definition:</i> The process is stable and repeatable                         |
| <i>Maturity Level 4 (Managed)</i>    | <i>Process measurement:</i> The process is measured and operates within measured limits |
| <i>Maturity Level 5 (Optimizing)</i> | <i>Process Control:</i> The focus is on continuous process improvement                  |

# Software Quality Models



# ISO 9126: Software Product Evaluation (1)

## Quality characteristics and guidelines for their use

The chosen characteristics are:

**Functionality**

**Reliability**

**Usability**

**Efficiency**

**Maintainability**

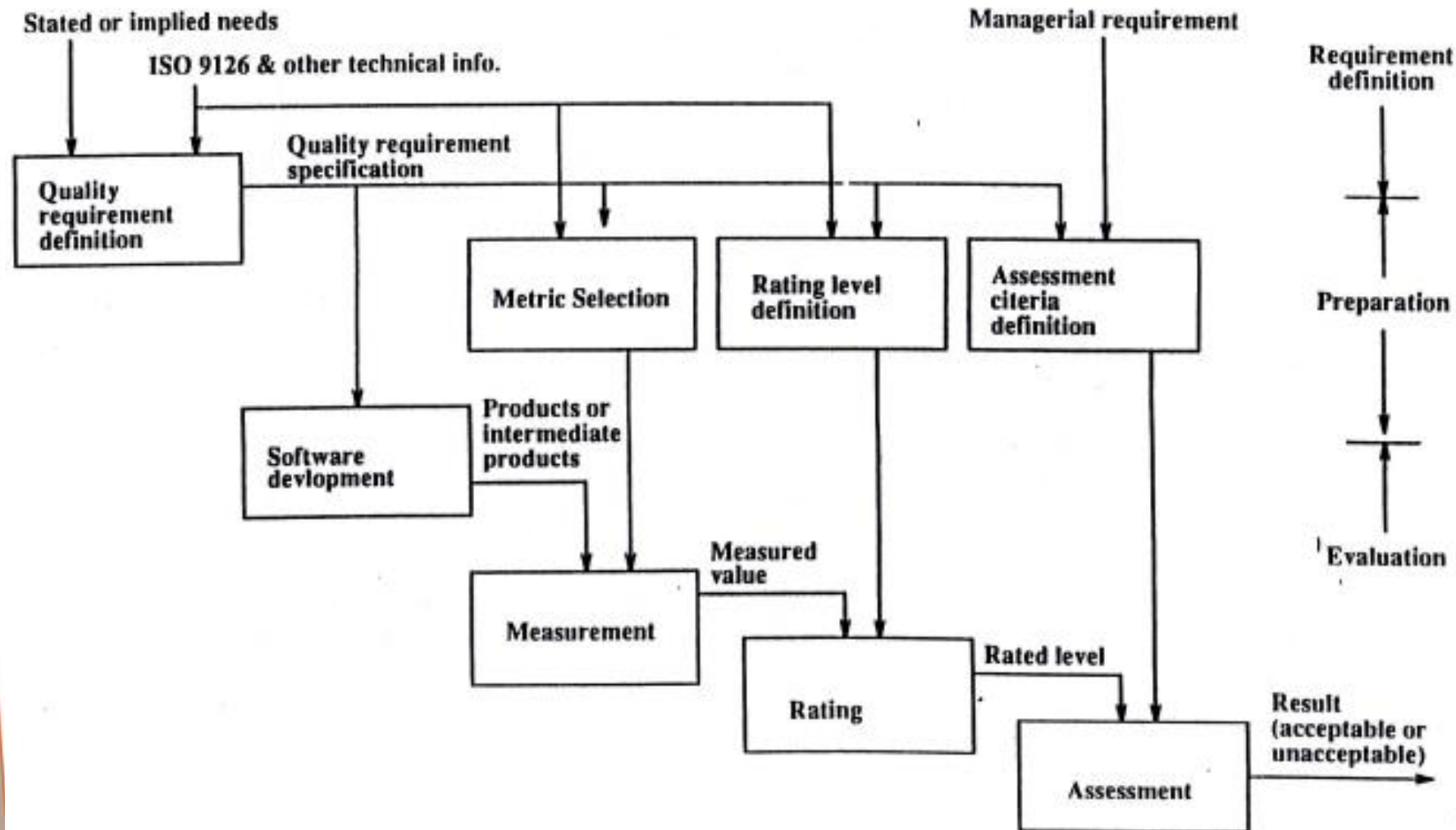
**Portability**

Each is defined as ‘a set of attributes that bear on ...’.

e.g. Reliability is ‘a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.’

# ISO 9126: Software Product Evaluation (2)

## The evaluation process model





# The Cleanroom Approach (1)

- *The Cleanroom* process was originally developed by Harlan Mills from IBM Fellow Department.
- The name *Cleanroom* was chosen to evoke the cleanrooms used in the electronics industry to prevent the introduction of defects during the fabrication of semiconductors.



# The Cleanroom Approach (2)

- The *Cleanroom* software engineering process is a software development process intended to produce software with *a certifiable level of reliability*.
- The focus of the *Cleanroom* process is on *defect prevention*, rather than *defect removal*.

# The Cleanroom Approach (3)

The first two principles of the *Cleanroom* process are:

- Software development based on *formal methods*
- Incremental implementation under *statistical quality control*: The quality of each increment is measured against pre-established standards to verify that the development process is proceeding acceptably.

# The Cleanroom Approach (4)

The third principle of the *Cleanroom* process is:

- *Statistically sound testing*: Based on the formal specification, a representative subset of software input/output trajectories is selected and tested. This sample is then statistically analyzed to produce an estimate of the reliability of the software, and a level of confidence in that estimate.

# Software Engineering Assessment

**“You can’t control what you can’t measure.”**

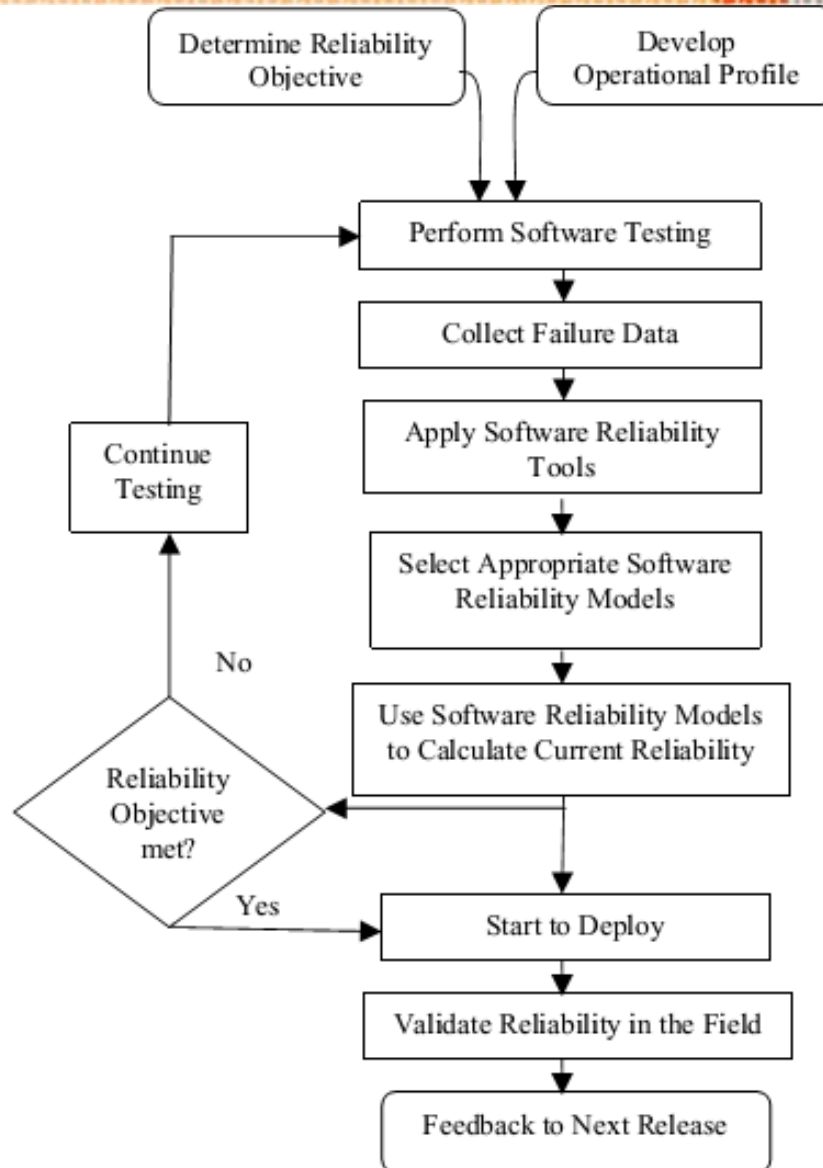
Tom De Marco: “Controlling Software Projects”

**“If You Can't Measure It, You Can't Manage It”**

Peter Drucker !

|  |  |  |  |
|--|--|--|--|
| <input checked="" type="checkbox"/> Start of test defect density | <input checked="" type="checkbox"/> End of test defect density |  |  |
| <input checked="" type="checkbox"/> Start of test defects        | <input checked="" type="checkbox"/> End of test defects        |  |  |
|  | <input checked="" type="checkbox"/> End of test failure rate   | <input checked="" type="checkbox"/> Failure rate at next major release | <input checked="" type="checkbox"/> Average failure rate |
|  | <input checked="" type="checkbox"/> End of test MTTF           | <input checked="" type="checkbox"/> MTTF at next major release         | <input checked="" type="checkbox"/> Average MTTF         |
|  | <input checked="" type="checkbox"/> End of test MTBI           | <input checked="" type="checkbox"/> MTBI at next major release         | <input checked="" type="checkbox"/> Average MTBI         |
|  | <input checked="" type="checkbox"/> End of test reliability    | <input checked="" type="checkbox"/> Reliability at next major release  | <input checked="" type="checkbox"/> Average reliability  |
|  | <input checked="" type="checkbox"/> End of test availability   | <input checked="" type="checkbox"/> Availability at next release       | <input checked="" type="checkbox"/> Average availability |

# Software Reliability Engineering

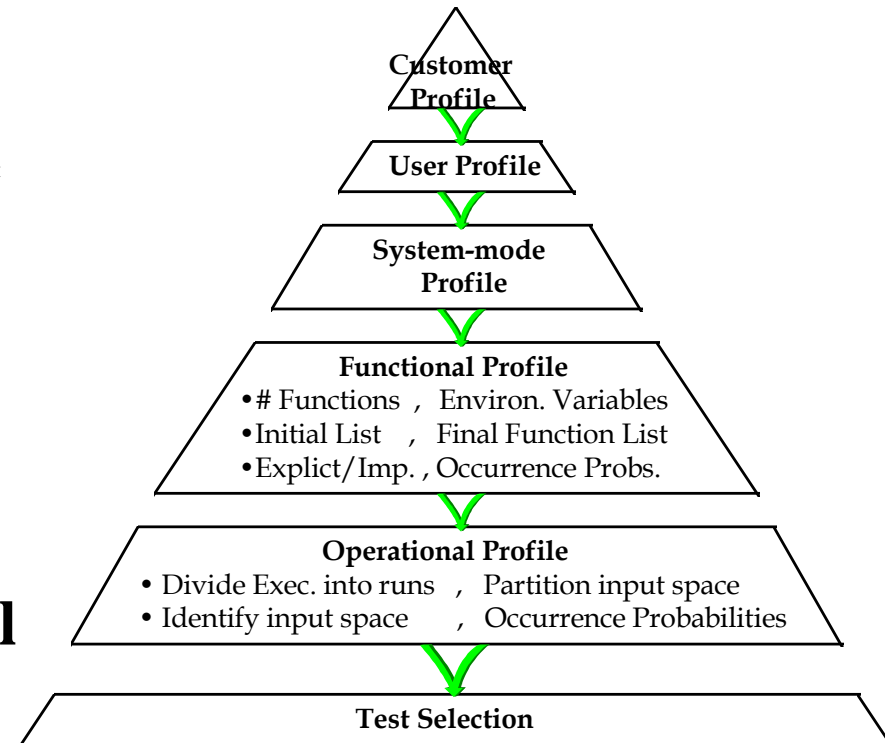


- A *reliability objective* is the specification of the *reliability goal* of a product from the customer viewpoint.
- The *operational profile* is a set of disjoint alternatives of system operational scenarios and their associated probabilities of occurrence.
- *Reliability modeling* is an essential element of the *reliability estimation process*.

# Developing an operational profile

Developing an operational profile for a system involves one or more of the following five steps:

1. Find the **customer profile**
2. Establish the **user profile**
3. Define the **system-mode profile**
4. Determine the **functional profile**
5. Determine the **operational profile** itself





# Statistical testing

- Testing software for reliability rather than fault detection.
- Test data selection should follow the predicted usage profile for the software.
- Measuring the number of errors allows the reliability of the software to be predicted.
- An acceptable level of reliability should be specified and the software tested and amended until that level of reliability is reached.

# Statistical testing procedure

- Determine operational profile of the software.
- Generate a set of data corresponding to this profile.
- Apply tests, measuring amount of execution time between each failure.
- After a statistically valid number of tests have been executed, reliability can be measured.

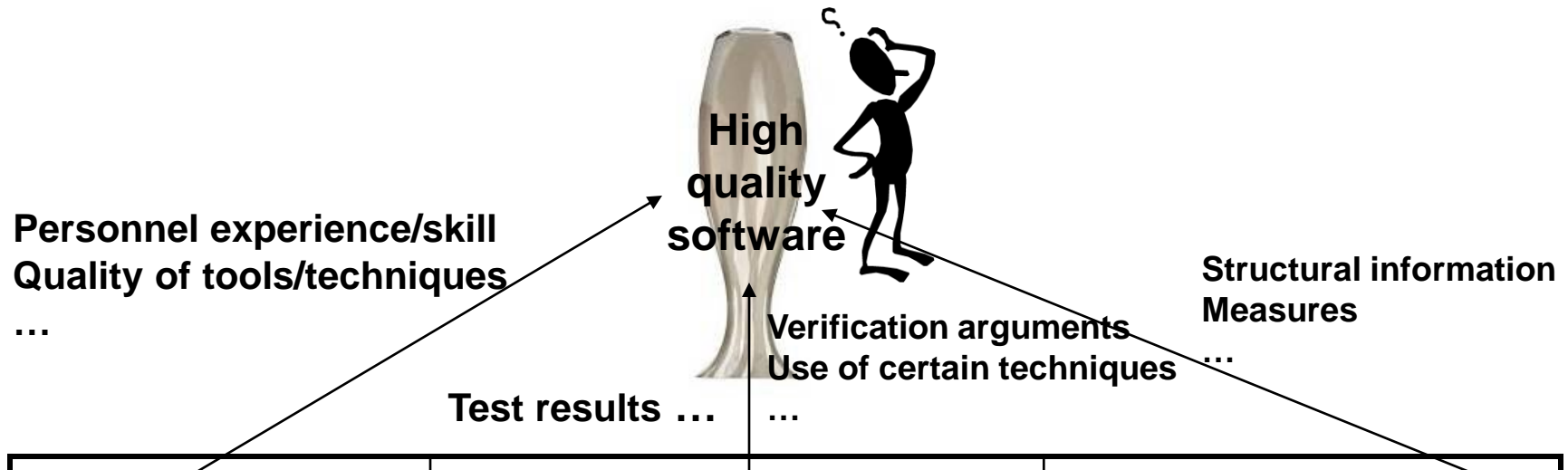
# Statistical testing difficulties

- Uncertainty in the operational profile
  - This is a particular problem for new systems with no operational history. Less of a problem for replacement systems.
- High costs of generating the operational profile
  - Costs are very dependent on what usage information is collected by the organisation which requires the profile .
- Statistical uncertainty when high reliability is specified
  - Difficult to estimate level of confidence in operational profile
  - Usage pattern of software may change with time.

# Operational profile generation

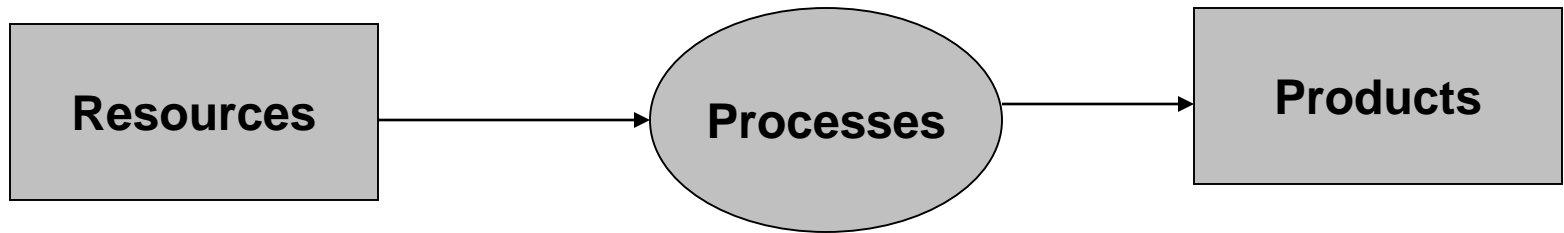
- Should be generated automatically whenever possible.
- Automatic profile generation is difficult for interactive systems.
- May be straightforward for ‘normal’ input but it is difficult to predict ‘unlikely’ inputs and to create test data for them.

# Achieving Software Reliability: diverse sources of information



| <i><b>Resources</b></i> | <i><b>Processes</b></i> | <i><b>Products</b></i> |
|-------------------------|-------------------------|------------------------|
| PEOPLE                  | FORMAL DEVELOPMENT      | DOCUMENTS              |
| TOOLS                   | DESIGN                  | TEST PLANS             |
| TECHNIQUES              | TEST                    | PROOFS                 |
| STANDARDS               | REVIEW MEETINGS         | MEETING MINUTES        |
| QUALITY PLANS           | ...                     | ...                    |
| ...                     |                         |                        |

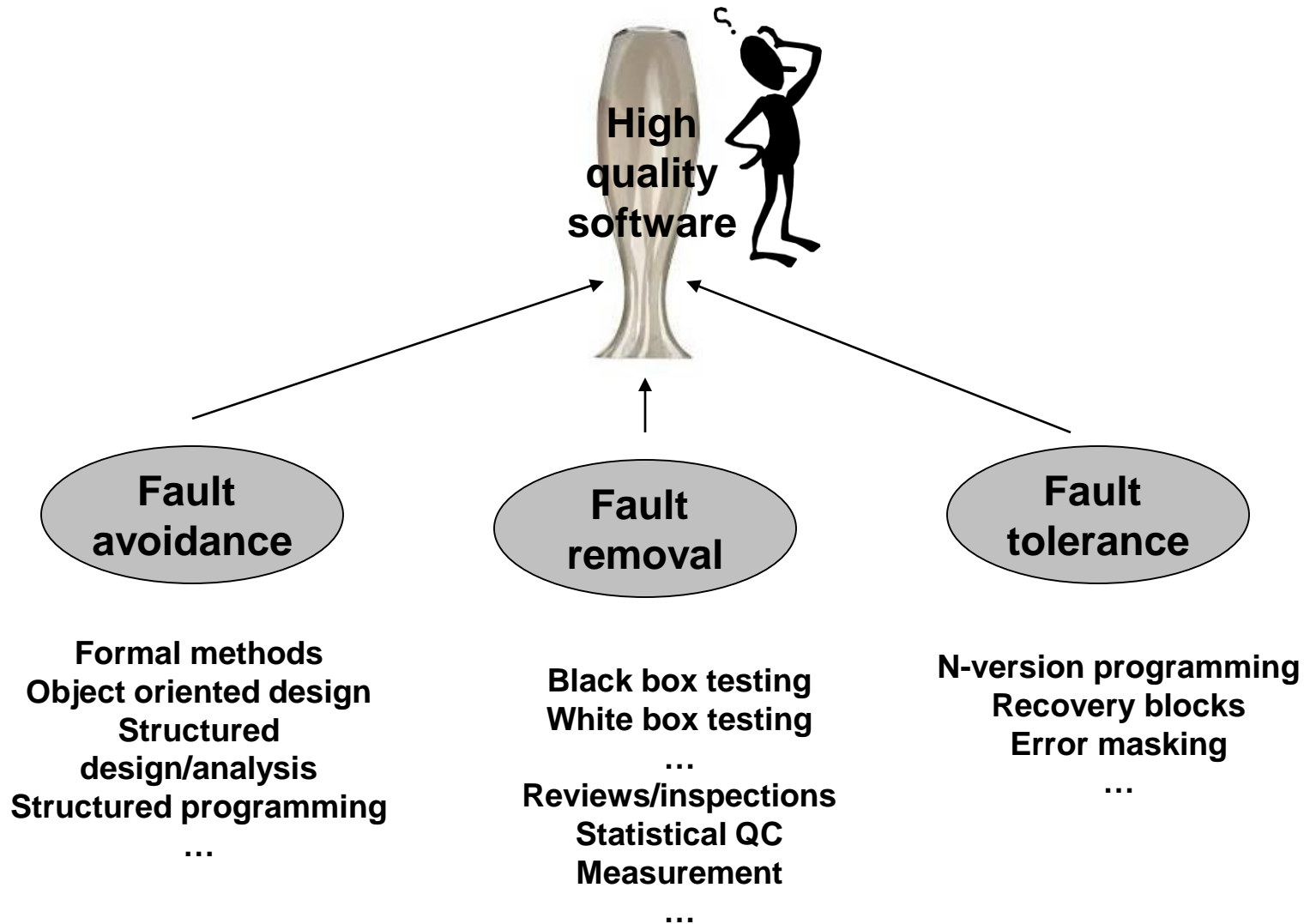
# Products, Processes, and Resources



- **Resource:** an item which is input to a process
  - people, hardware, software, etc.
- **Process:** a software related activity or event
  - testing, designing, coding,
- **Product:** an object which results from a process
  - test plane, specification and design documents, source and object code, minutes of meetings, etc.



# Achieving software reliability: diverse approaches



# WHY USE FAULT TOLERANCE

- Avoidance and Removal never perfect
- Residual faults always possible
- Must prevent residual faults causing failure
- Only way to achieve ultra-reliability(?)

# SOFTWARE FAULT TOLERANCE PROBLEMS

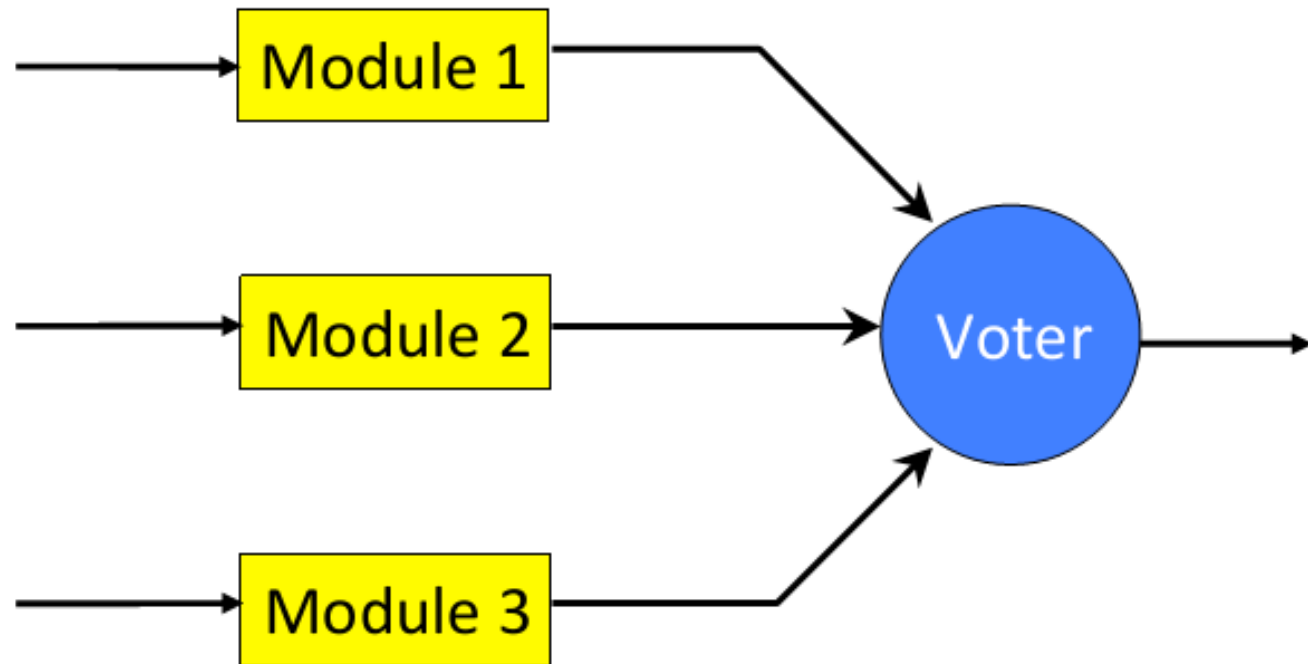
Versions / blocks  
do not fail  
independently

- Difficulties for one team are difficulties for other
- Same mistakes
- Specification faults
- Specification faults affect all versions / blocks

**BUT**

- Significant improvement in reliability usually achieved
- Shown by experiment

# Triple Modular Redundancy (TMR)



**Masking:** the use of sufficient redundancy may allow recovery without explicit error detection.

# Checkpointing - Definition

A **checkpoint** is a snapshot of entire state of the process at the moment it was taken

- all information needed to restart the process from that point

Checkpoint saved on **stable storage** of sufficient reliability

- Most commonly used - **Disks**: can hold data even if power is interrupted (but no physical damage to disk); can hold enormous quantities of data very cheaply
  - Checkpoints can be very large - tens or hundreds of megabytes
- **RAM** with a battery backup is also used as stable storage
- No medium is perfectly reliable - reliability must be sufficiently high for the application at hand

# Example: Airbus 330/340 Fight Control System





# Advantages of “fly-by-wire”

- Pilot workload reduction
  - The fly-by-wire system provides a more usable interface and takes over some computations that previously would have to be carried out by the pilots.
- Airframe safety
  - By mediating the control commands, the system can ensure that the pilot cannot put the aircraft into a state that stresses the airframe or stalls the aircraft.
- Weight reduction
  - By reducing the mechanical linkages, a significant amount of weight (and hence fuel) is saved.

# Fault tolerance

- Fly-by-wire systems must be fault tolerant as there is no “fail-safe” state when the aircraft is in operation.
  - In the Airbus, this is achieved by replicating sensors, computers and actuators and providing “graceful degradation” in the event of a system failure. In a degraded state, essential facilities remain available allowing the pilot to fly and land the plane.

# Failsafe operation

## Definition

- A system is **failsafe** if it adopts “safe” output states in the event of failure and inability to recover.

## Notes

- Example of failsafe operation
  - Railway signaling system: failsafe corresponds to all the lights on red
- Many systems are not failsafe
  - Fly-by-wire system in an aircraft: the only safe state is on the ground

# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
- 3. Measuring Software Reliability**
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
7. Lessons Learned
8. Background Literature

# 3. Measuring Software Reliability

- Measures of software reliability
- Software measurement
- Software engineering assessment
- Definition of software reliability
- Software reliability: measurement
- Predictive measures
- Reliability models

# Key points (1)

- *Reliability* is usually the most important *dynamic software characteristic*.
- Professionals should aim to produce reliable software.
- Reliability depends on the pattern of usage of the software. *Faulty software can be reliable*.
- *Reliability requirements* should be defined *quantitatively* whenever possible.



# Key points (2)

- There are many different reliability metrics. The metric chosen should reflect the type of system and the application domain.
- Statistical testing is used for reliability assessments. Depends on using a test data set which reflect the use of the software.
- *Reliability growth models* may be used to predict when a *required level of reliability* will be achieved.

# The predictor distribution of time to failure (TTF).

Software reliability growth models yield predictor distribution of time to failure (TTF). From this we can derive:

- Probability of mission success
- Median TTF (sometimes Mean)
- Hazard rate (ROCOF)
- Expected no. of faults detected
- Expected further execution time required to achieve the above.

# Software Measurement

- Perfection cannot be guaranteed
  - Proof is fallible
  - “Good practice” may not be good enough
  - *“No faults found” does not mean “No faults left”*
- Must measure quality
  - measure dependability dynamically
  - under realistic conditions
  - collect data

# Definition of software reliability

- The **reliability** of a software item is the probability that the system of which it is part will operate, without failure due to the activation of a fault in the software, under given conditions for a given time interval.
  - *Probability*: Subjective degree of belief
  - *Failure*: Departure of system behaviour from what is required
  - *Fault*: Design defect due to human error
  - *Conditions*: Defines “mode of use” of software
  - *Time*: “Execution time” (measure of software use)

# Software reliability measurement

- *Software reliability measurement* is always a *prediction problem*: how is the software likely to behave from now given its past record of failures.
- We can predict future failures well if we have observed past failures frequently.
- This does not help us with high reliability requirements. How reliable is a system which has not failed for 10,000 hours of use?
- The system requirement for the A330 is MTTF of  $10^9$  flying hours. The software requirement is higher. How can we certify this system?

# Prediction versus estimation (1)

- The major difference between software reliability prediction and software reliability estimation is that *predictions are performed based on historical data while estimations are based on collected data.*
- *Predictions*, by their nature, will almost certainly be *less accurate than estimations.* However, they are useful for improving the software reliability during the development process.



# Prediction versus estimation (2)

- If the organization *waits until collected data is available (normally during testing)*, it will generally *be too late* to make substantial improvements in software reliability.
- ***The predictions should be performed iteratively during each phase of the life cycle and as collected data becomes available the predictions should be refined*** to represent the software product at hand.

# Predictive measures

- Predictive measures invariably require a prediction system.
- A *prediction system* consists of a mathematical model, together with a set of procedures for determining unknown parameters, and interpreting results. The procedures are stochastic.

**The model alone is insufficient; using the same model will yield different results if we use different prediction procedures.**

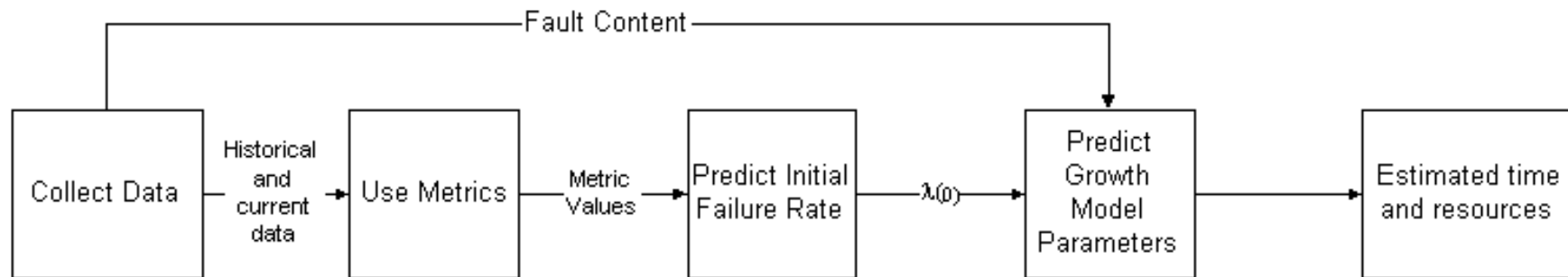
# Reliability Measurement Goal

- Reliability measurement is a *set of mathematical techniques* that can be used *to estimate and predict* the reliability behavior of software during its *development and operation*.
- The primary goal of software reliability modeling is to answer the following question:

**“Given a system, what is the probability that it will fail in a given time interval, or, what is the expected duration between successive failures?”**

# Software reliability predictions

- **Software reliability prediction** is performed at *each phase of the software development process* up to software system test.
- Software reliability predictions are made during the software development phases *that precede software system test*, and are available in time to *feed back into the software development process*. The predictions are based on *measurable characteristics of the software development process* and the products produced by that process.



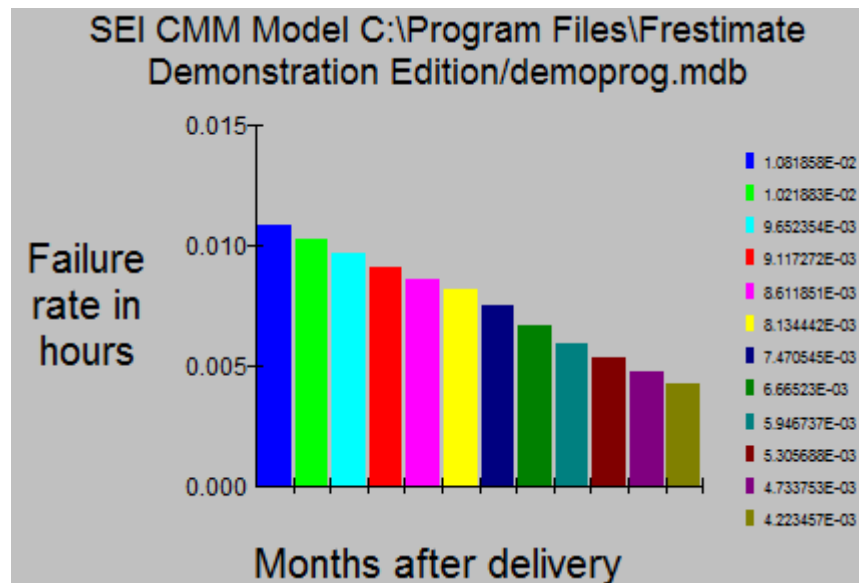
# Probabilistic modelling

- Why are statistical methods necessary?
- Why reliability?
- What is the nature *of the failure process*?
- ... *of the debugging process*?
- How can we measure, predict?
  
- Why do we want to measure it anyway? Some potential benefits:
  - some software is safety-critical (A330-340, Sizewell B)
  - all software needs to be sufficiently reliable (warranties? support costs? etc)
  - methodology for a rational choice between SE technologies (eg are formal methods the most cost-effective way of achieving R?)
  - management tool for scheduling and monitoring software development (is project on time?)

# Finding errors: does it increase or decrease your confidence in the software reliability?

*“The number of errors detected by the verification process attest to the effectiveness of the software development principles ... Significantly enhance the probability of achieving essentially error-free software.”*

(Westinghouse commenting on their work on the Sizewell B nuclear protection system.)



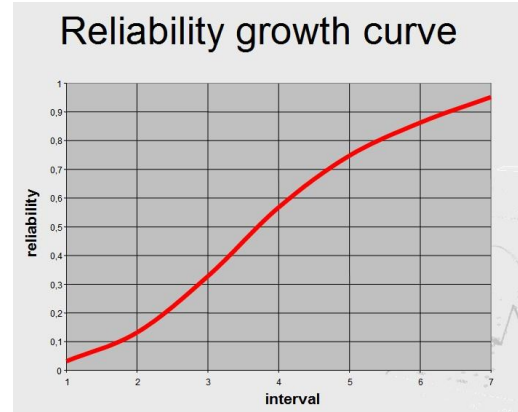
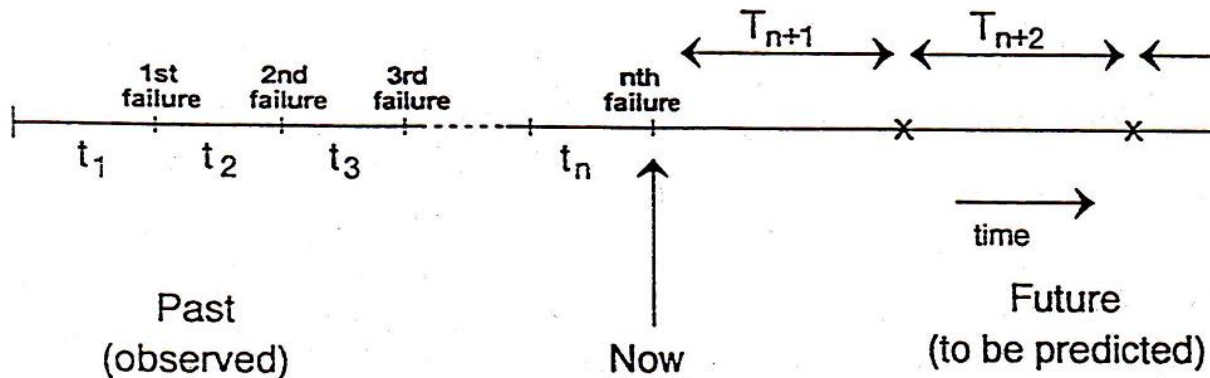
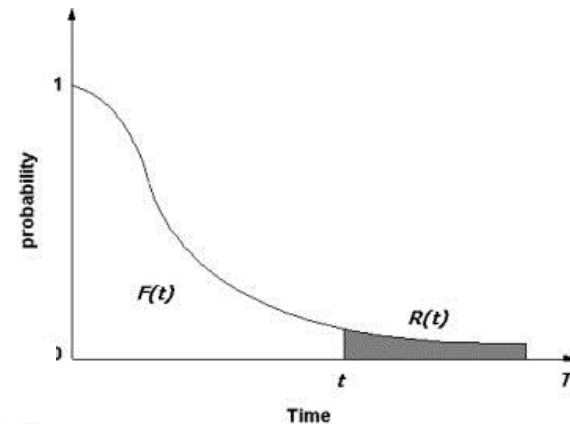


# Refined data for software reliability models

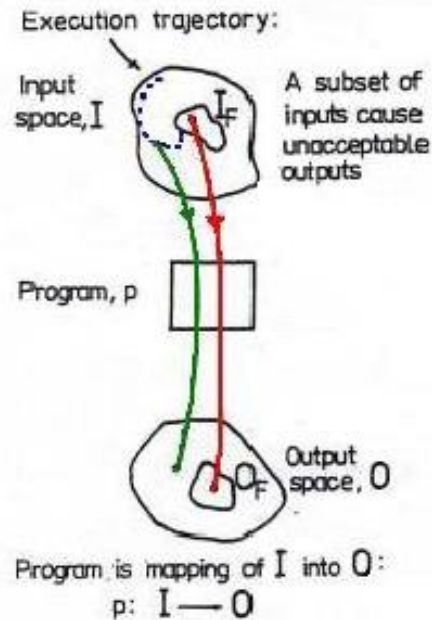
- Failure time data
  - List of interfailure times
    - \* execution time between activation of successive new faults
- Failure count data
  - Count of new faults activated, and total execution time accumulated, in successive calendar periods

# Why probability?

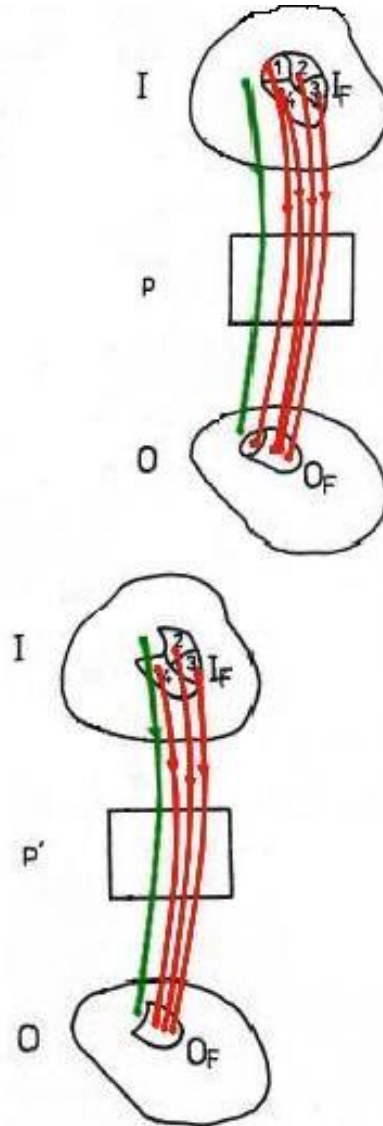
- A computer is a deterministic machine – why don't we know when it will fail next?
- There is intrinsic uncertainty
  - about the sequence of inputs it will receive
  - about where faults lie
  - about the effect of attempts to remove faults
- We need probability to describe such uncertainty



# Uncertainty Modelling



A conceptual model of the software failure process as a mapping  $p: I \rightarrow O$



Modelling type 1  
 uncertainty is easiest.  
 Seems plausible to  
 assume  $I_F$  encountered  
 purely randomly:

- Time to failure is exponential

$$F(t) = \Pr(T < t) = 1 - e^{-\lambda t}$$

$$f(t) = F'(t) = \lambda e^{-\lambda t}$$

$$R(t) = e^{-\lambda t}$$

How to model type 2  
 uncertainty: the way in  
 which the value of  $\lambda$   
 changes as debugging  
 proceeds.

# Types of uncertainty

There is *intrinsic* uncertainty about future failure behaviour because of:

1. Uncertainty about the *operational environment*: even if we knew  $I_F$  we would not know when it would be encountered next.
2. Uncertainty about the *effect of fault-removal*:
  - we never know whether a fix is successful
  - even if it is successful, we do not know how much it improves overall reliability.

Models must be judged by their ability to capture both sources of uncertainty  
Frequentist for 1, but not for 2?

# A conceptual model of the software failure process

- In summary:
  - debugging creates a sequence of programs  $p(1), p(2), \dots$
  - there is a sequence of failure subsets  $I_F(1), I_F(2), \dots$
  - these have 'sizes' represented by random variables  $\lambda(1), \lambda(2), \dots$
  - distribution of  $T_i$  is exponential with rate  $\lambda(i)$ .

# Software reliability metrics

| <i><b>Metric</b></i>  | <i><b>Definition</b></i>  | <i><b>Formula</b></i>                           |
|---|---|---|
| <b>Reliability</b><br>$R(t)$  | The probability that a given piece of software will execute without failure in a given environment for a given period of time | $R(t) = P(T > t)$ $= 1 - F(t)$ $= 1 - P(T < t)$ |
| <b>Mean time to failure (MTTF)</b>                                  | The time which is expected to elapse between the current time and the next failure  | $MTTF = \int_0^{\infty} t f(t) dt$              |
| <b>Median <math>m</math></b>  | This term implies the point of statistical distribution that a given quantity is equally likely to fall either side of        | $F(m) = 1/2$                                    |
| <b>Rate of occurrence of failures ROCOF <math>\lambda(t)</math></b> | The current rate at which failures are occurring  | $\lambda(t) = f(t)/R(t)$                        |

**Note:  $F(t)$  is called distribution function of the random variable  $T$ ; its probability density function is  $f(t) = F'(t)$**



# Reliability metrics

- **Mean time to failure**

- Measure of the time between observed failures
- MTTF of 500 means that the time between failures is 500 time units
- Relevant for system with long transactions e.g. CAD systems

- **Availability**

- Measure of how likely the system is available for use. Takes repair/restart time into account
- Availability of 0.998 means software is available for 998 out of 1000 time units
- Relevant for continuously running systems e.g. telephone switching systems

# Software reliability

- **Cannot be defined objectively**

- Reliability measurements which are quoted out of context are not meaningful

- **Requires operational profile for its definitions**

- Requires operational profile defines the expected pattern of software usage

- **Must consider fault consequences**

- Not all faults are equally serious. System is perceived as more unreliable if there are more serious faults.

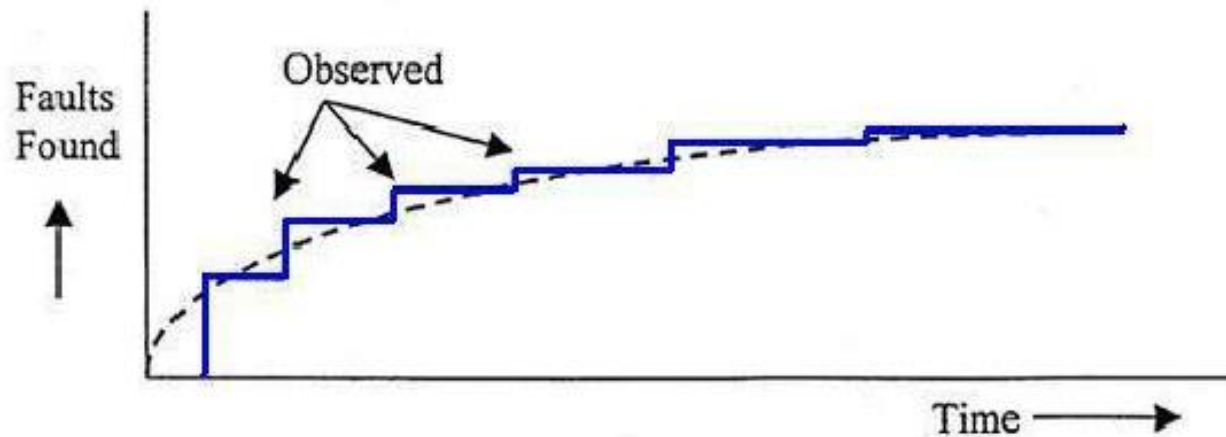
# Reliability economics

- Because of very high costs of reliability achievement, it may be more cost effective to accept unreliability and pay for failure cost.
- However, this depends on social and political factors. A reputation for unreliable products may lose future business.
- Depends on system type – for business systems in particular, modest reliability may be adequate.

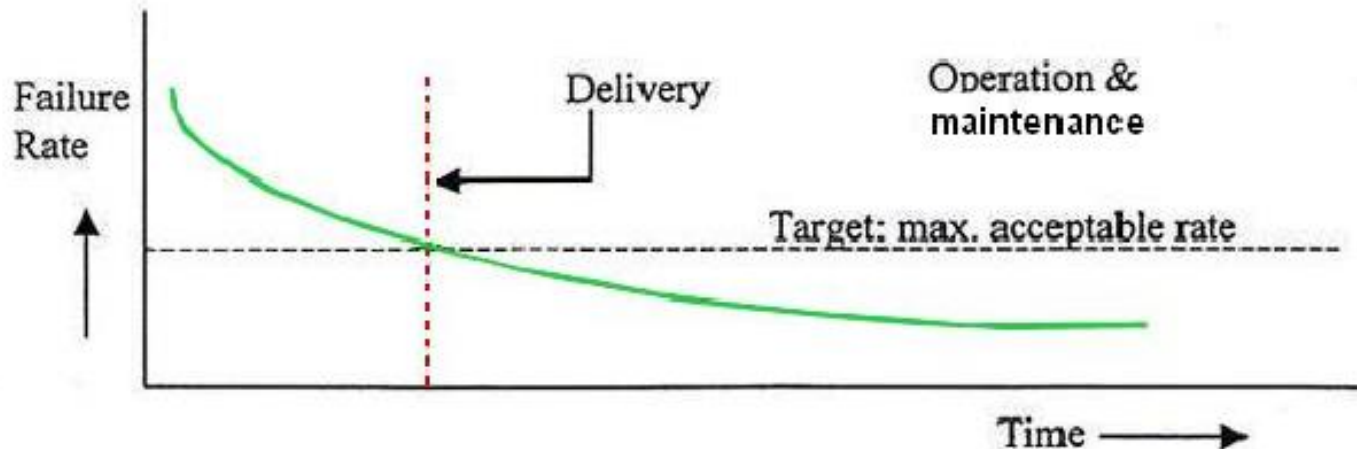
# Reliability measurement

- Measure the number of system failure for a given number of system inputs
  - Used to compute POFOD
- Measure the time (or number of transactions) between system failures
  - Used to compute ROCOF and MTTF
- Measure the time to restart after failure
  - Used to compute AVAIL

# Reliability growth (faults found, failure rate)



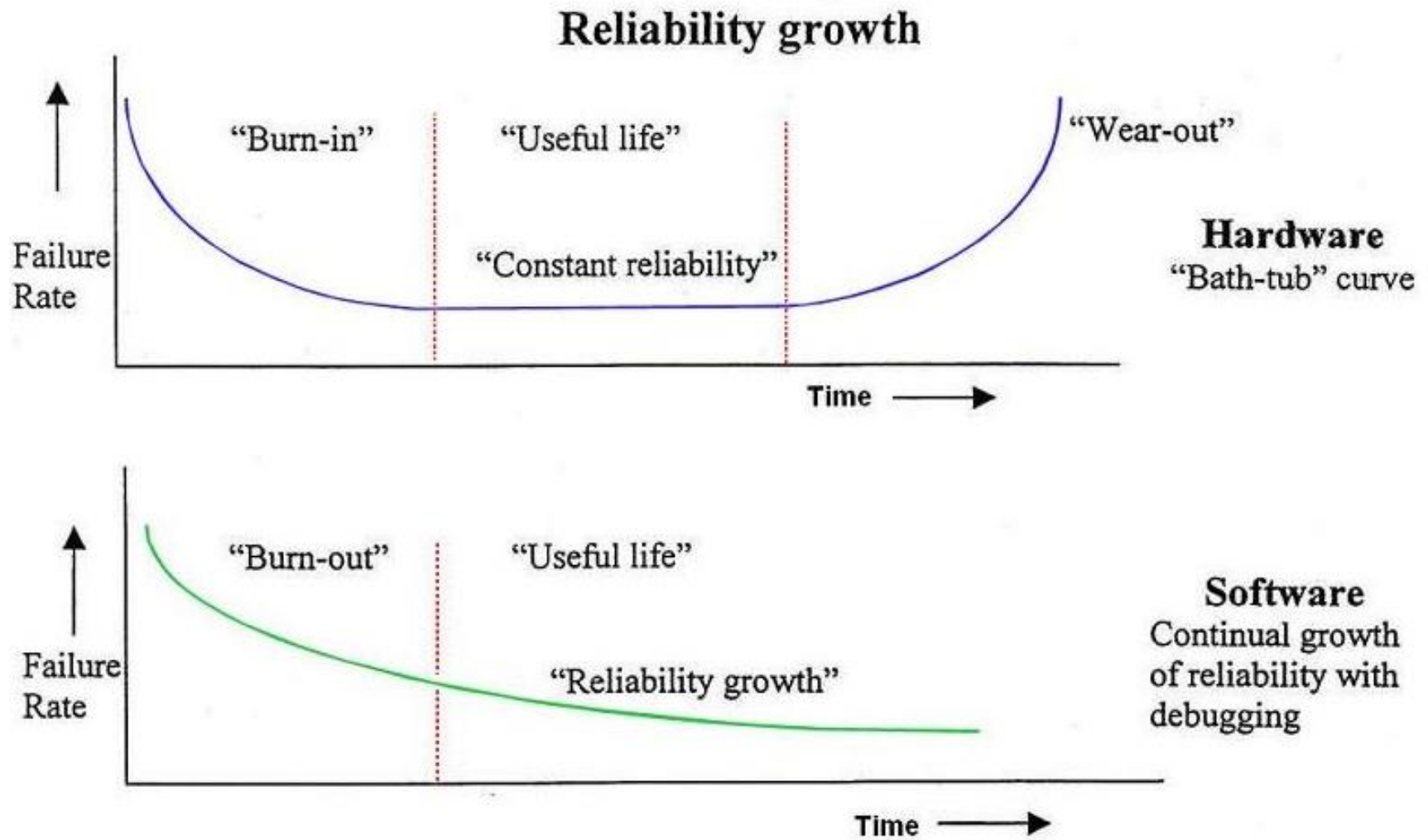
Record faults detected over operating time. Analyse the pattern and predict future behaviour.



Monitor failure rate during test and trial.

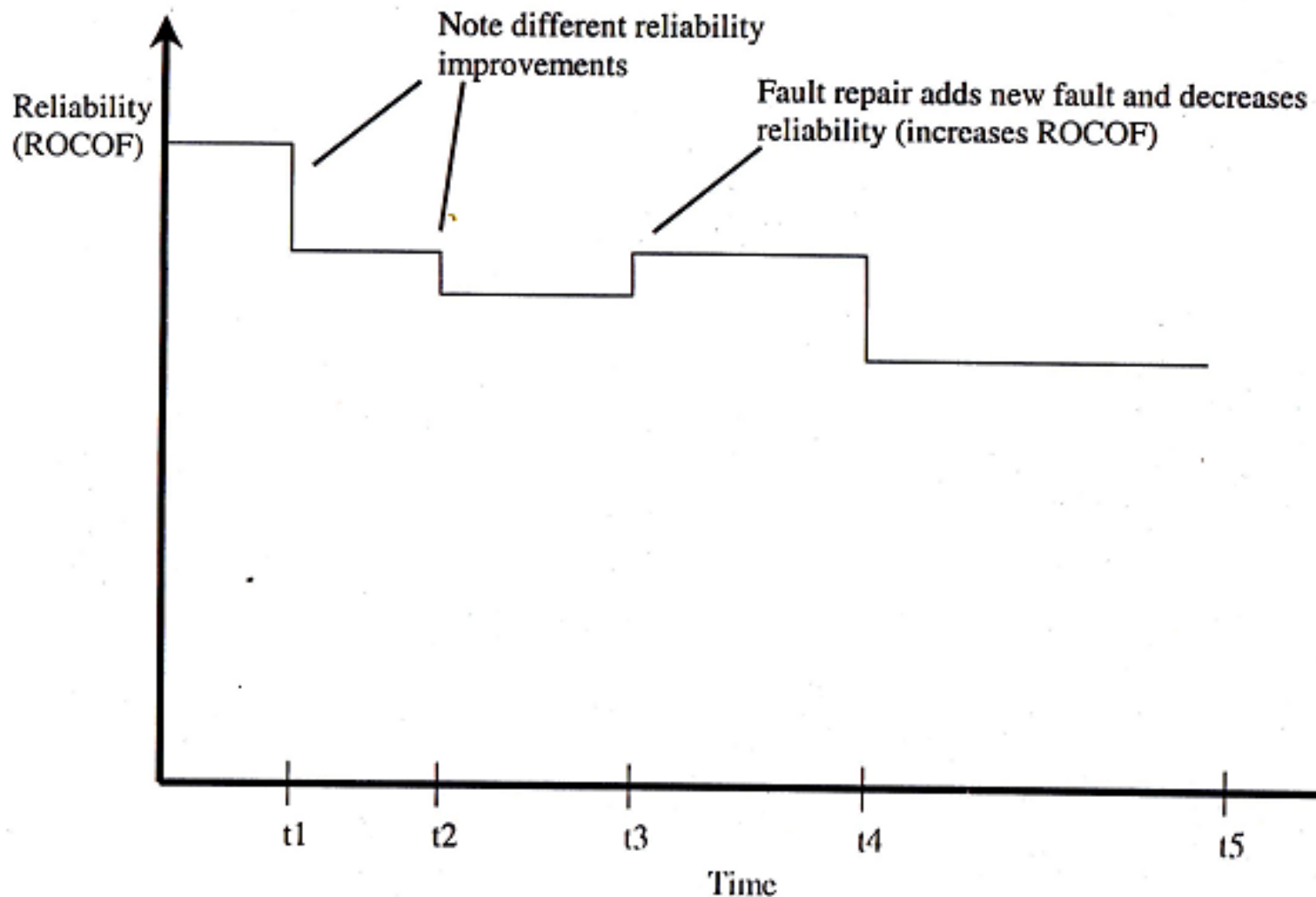
Deliver when pre-defined target value is reached

# Reliability growth (failure rate)



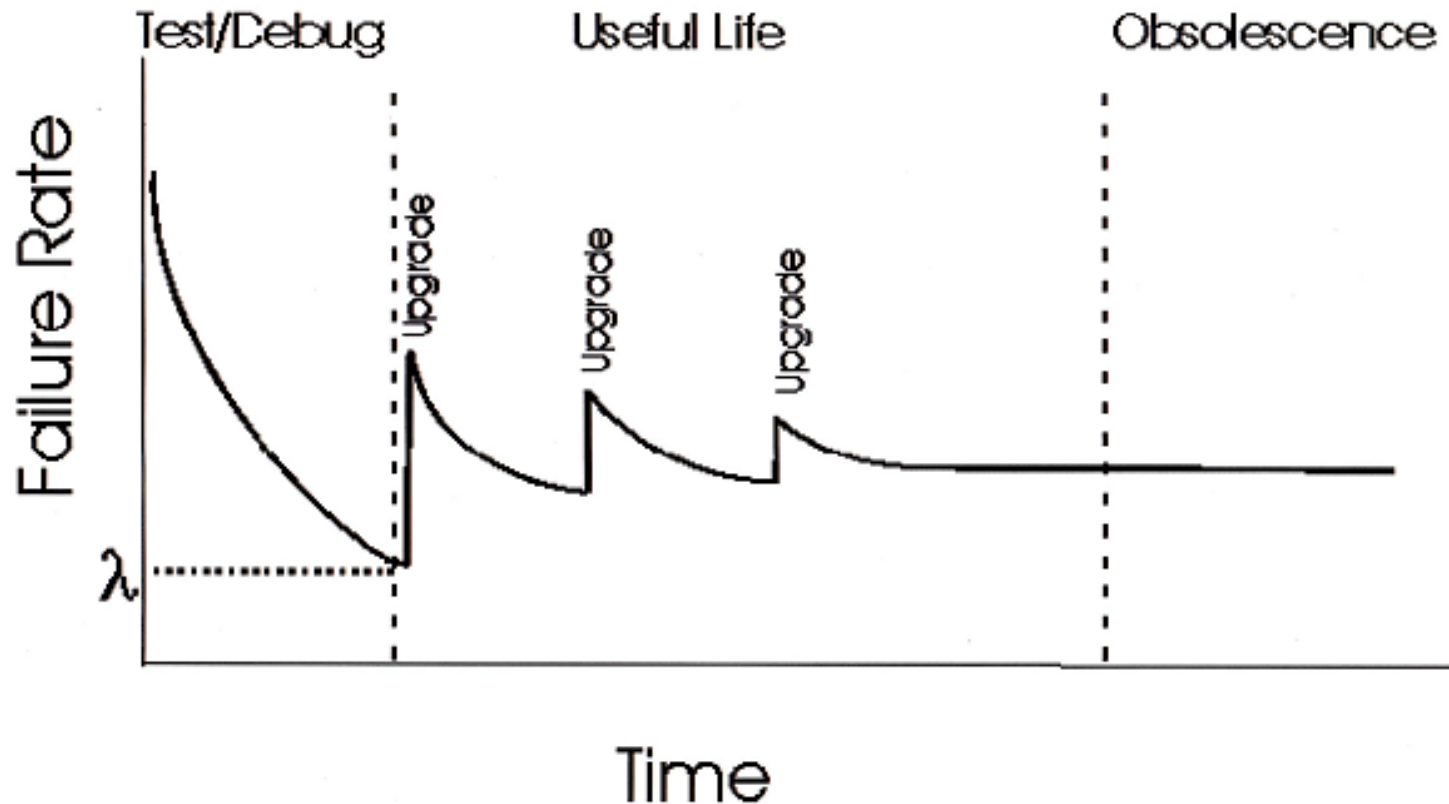


# Random-step reliability growth

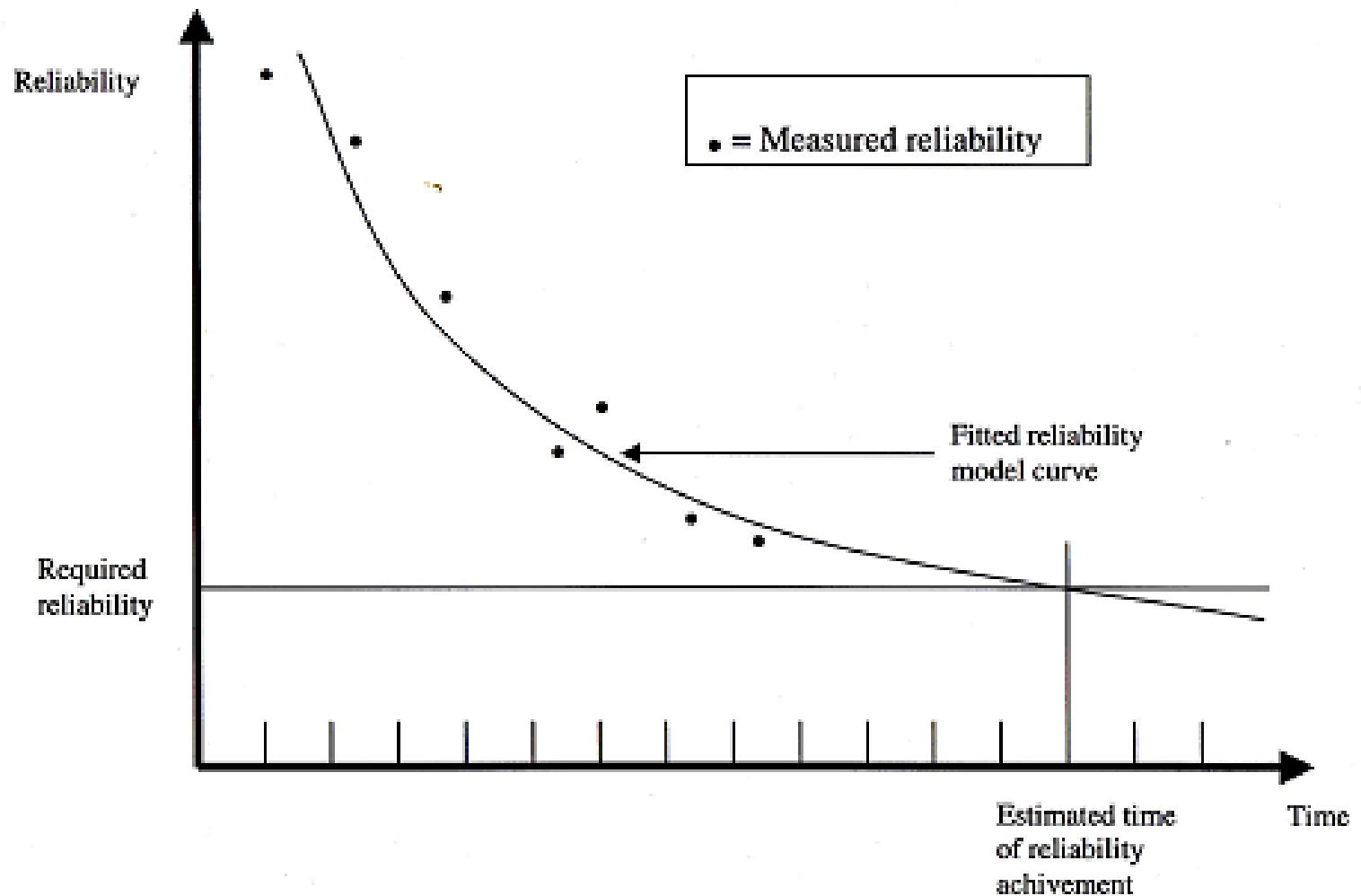


# Bathtub curve

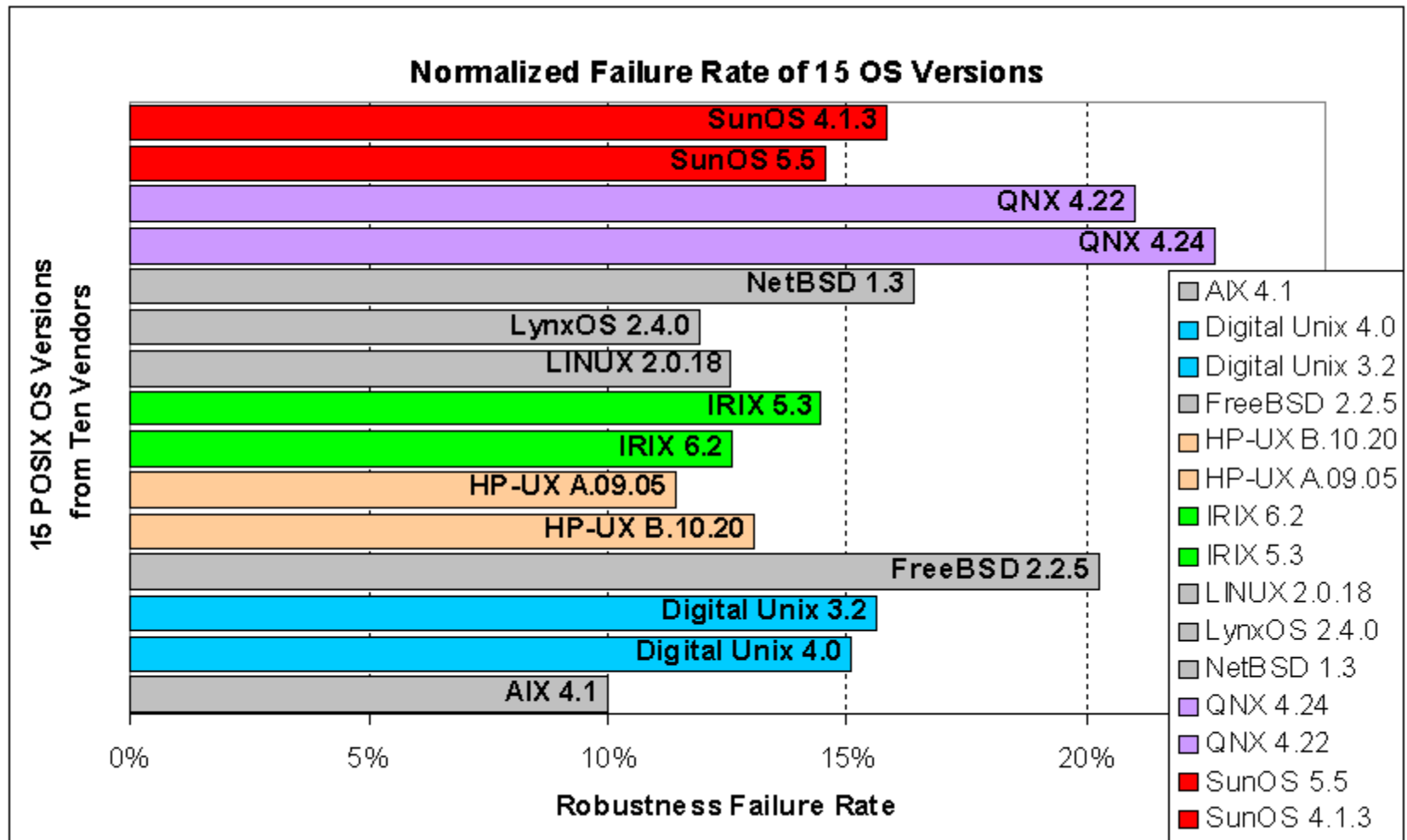
## Revised bathtub curve for software reliability



# Reliability prediction



# Software failure rate - example



# Reliability improvement

- Reliability *is improved* when software faults which occur in the most frequently used parts of the software *are removed*.
- Removing  $x\%$  of software faults will not necessarily lead to an  $x\%$  reliability *improvement*.
- In a study, removing 60% of software defects *actually led to a 3% reliability improvement*.
- Removing faults with serious consequences is the most important objective.

# Some parametric software reliability models

! Remember that the problem is one of prediction. This entails the following *triad*:

1. *The model itself*, which gives a complete probability specification of the process (eg the joint distribution of  $\{T_i\}$ , with unknown parameters, say  $\beta$ )
2. *An inference procedure* for the unknown parameters of 1 based on realisations (data)  $t_1, t_2, \dots, t_n$
3. *A prediction procedure* which combines 1 and 2 to make predictions about future behaviour

**Notice that:**

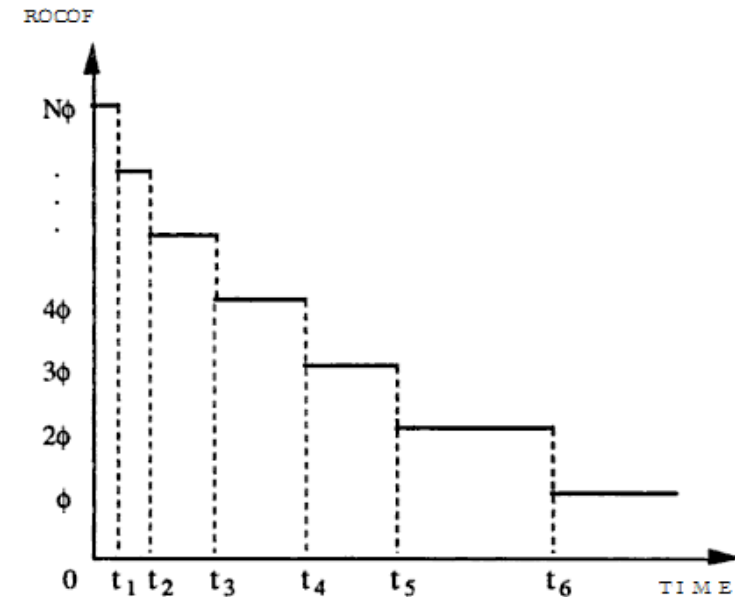
- Disaster can strike at any (or all) of these stages!
- A “good” model seems necessary for good prediction, but is not sufficient
- All models are “wrong” (but some are more wrong than others!)
- Various solutions to 2, eg ML estimation, eg Bayesian posterior distribution
- ditto for 3, eg substitution of ML ests., eg Bayesian predictive Distributions.



# Jelinski-Moranda [JM] model (1)

This assumes:

- $t_1, t_2, t_3, \dots$  are independent random variables
- Type 1 uncertainty:  $t_i$  is exponentially distributed, parameter  $\lambda_i$
- Type 2 uncertainty:  $\lambda_i = (N-i+1)\phi$ , where  $N$  is the initial number of faults (finite) and  $\phi$  is contribution to overall failure rate of each fault.
- No fault introduction while correcting detected faults: each activated fault is corrected before new executions
- Inference by ML, prediction via 'plug-in' rule.
- $R(t_i) = \exp(-\lambda_i t_i)$  is the reliability function



# Jelinski-Moranda [JM] model (2)

The JM model assumes that  $\{t_k, k=1,n\}$  are the realisations of the random variables  $T_k$  with exponential probability density functions:

$$f_k(t) = \lambda_k(t) e^{-\int_0^t \lambda_k(u) du} = (N - k + 1)\phi e^{-(N-k+1)\phi t} \quad (1)$$

where

$$\lambda_k(t) = (N - k + 1)\phi = \text{const.} \quad (2)$$

When our observation of the reliability growth(debugging) begins, the program contains  $N$  faults. Removal of a fault occurs whenever a failure occurs, and at each event the rate of occurrence of failures is reduced by an amount  $\phi$ . Thus,  $\phi$  can be taken to represent the size of a fault.

# Jelinski-Moranda [JM] model (3)

When  $\{t_k, k=1,n\}$  are the observed data, the current reliability is:

$$R_k(t) = e^{-\lambda_k(t)} = e^{-(N-k+1)\phi t} \quad (3)$$

The unknown parameters of the model,  $N$  and  $\phi$ , are estimated by maximum likelihood method .

For the series of “n” recorded faults the likelihood is:

$$L = \prod_{k=1}^n f_k(t_k) \quad (4)$$

# Jelinski-Moranda [JM] model (4)

and further (5),

$$\begin{aligned}\ln L &= \ln \prod_{k=1}^n f_k(t_k) = \sum_{k=1}^n \ln[f_k(t_k)] = \sum_{k=1}^n [\ln(N-k+1)] + n \ln \phi - \phi \sum_{k=1}^n [(N-k+1)t_k] \\ &= \sum_{k=1}^n [\ln(N-k+1)] + n \ln \phi - \phi y_n - \phi(N-n)x_n\end{aligned}$$

$$\text{where } x_n = \sum_{k=1}^n t_k \quad y_n = \sum_{k=1}^n x_k$$

and the mathematical relation:

$$nx_n - y_n = \sum_{k=1}^n (k-1)t_k \quad (6)$$

# Jelinski-Moranda [JM] model (5)

$N$  and  $\phi$ , are estimated using the following derivative equations:

$$\begin{aligned}\frac{\partial \ln L}{\partial N} &= 0 = \sum_{k=1}^n \frac{1}{N - k + 1} - \phi x_n \\ \frac{\partial \ln L}{\partial \phi} &= 0 = \frac{n}{\phi} - y_n - (N - n)x_n\end{aligned}\tag{7}$$

and using the notation  $N = \alpha - 1$ , we obtain:

$$\sum_{k=1}^n \frac{1}{\alpha - k} - \frac{n}{\alpha - (n + 1) + \frac{y_n}{x_n}} = 0\tag{8}$$

with the restrictions:

$$N \in \mathbf{N}, \phi > 0, N \geq n$$



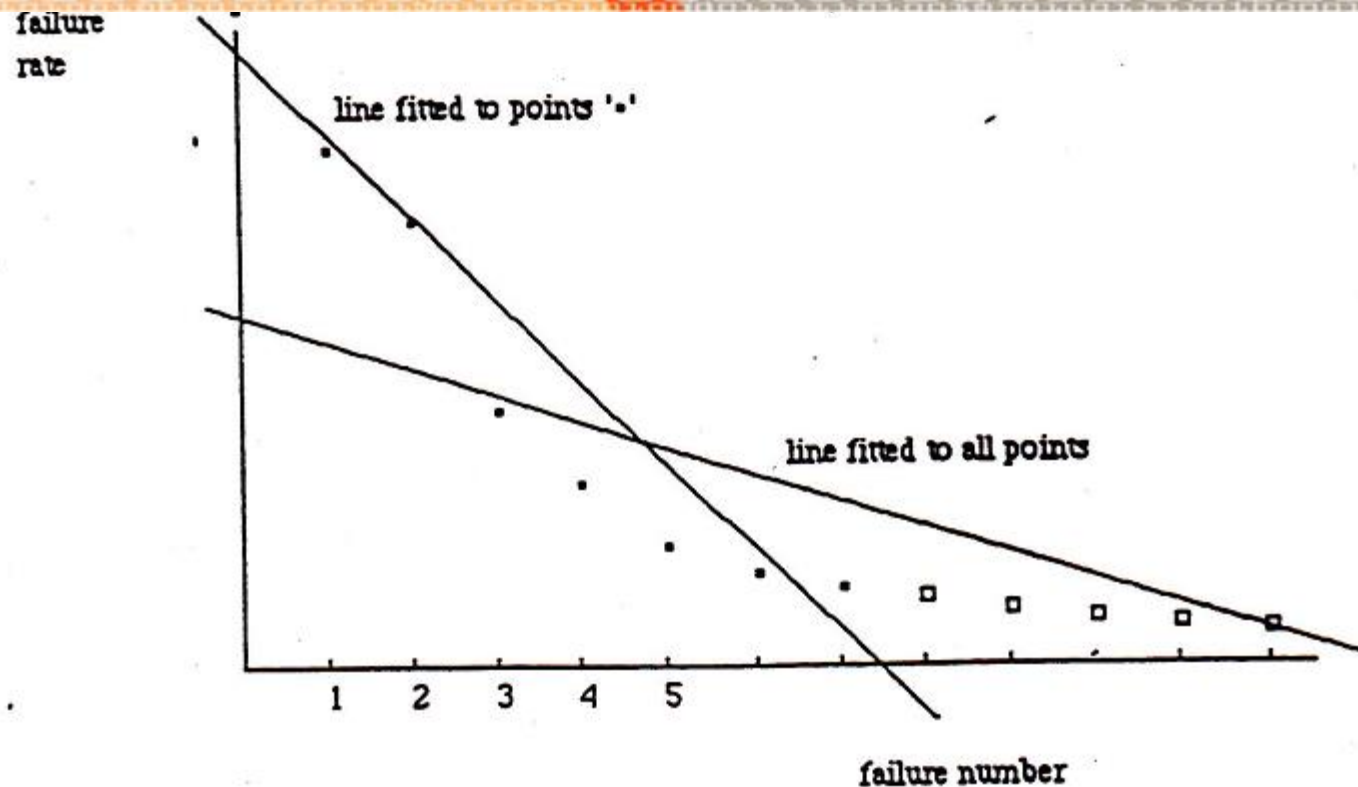
# Why is JM always optimistic ? (1)

In virtually all data we have analysed, JM/Musa model gives answers which are too optimistic. This is the result of assuming all faults are 'equal' when in fact they are not:

If faults have different sizes (rates) and we incorrectly assume they are equal, what kind of errors should we see?



# Why is JM always optimistic ? (2)



- vertical intercept is estimate of  $N\phi$ : we should see this *decreasing overall*
- $(-1) \times$  slope is  $\phi$ : we should also see this decreasing overall

# Why is JM always optimistic ? (3)

- Example: Musa System 1 data

| $i$ | $N\phi$ | $\phi$ |
|-----|---------|--------|
| 20  | .00350  | 1.35   |
| 25  | .00300  | 0.85   |
| 30  | .00317  | 0.99   |
| 35  | .00252  | 0.68   |
| 40  | .00196  | 0.41   |
| 45  | .00189  | 0.39   |
| 50  | .00172  | 0.32   |

- similar results on other data sets

# Criticisms of [JM] model

- Foundational assumptions unrealistic: true fault rates differ by orders of magnitude.
- Parameter estimates of  $N$  (by ML) have poor properties:
  - often seriously underestimate  $N$  (even  $N!$  – factorial!)
  - sometimes goes to infinite (essentially) when no evidence of reliability growth in the data)
- Shall show reliability predictions are poor: usually grossly “optimistic”.

# Musa model [M]

- Assumptions similar to the Jelinski-Moranda model.
- Parameters definition:  $M_0$  = number of faults in the software;  $N_0$  = number of failures;  $B$  = fault reduction factor: number of faults / number of failures;  $C$  = compression factor (execution time in operation / in test);  $\phi$  = fault manifestation rate.
- $\lambda(i) = B C \phi (N_0 - i + 1)$
- $N(t) = N_0 [1 - \exp (-B C \phi t)]$  = number of failures observed at  $t$  ( execution time)

# Littlewood [L] Model

- The main hypotheses are the following:
  - At a failure, the fault is removed with certainty
  - Faults manifest themselves at times that are independently exponentially distributed
  - The rates of these faults come from  $\Gamma(\alpha, \beta)$  distribution
- Notations and relations:
  - $N$  is the initial number of faults
  - $\phi_i$  represents the (random variable) rate associated with fault  $i$  (in arbitrary labelling).
  - $\lambda_i = \phi_1 + \phi_2 + \dots + \phi_{N-i+1}$       ROCOF is  $\lambda(t) = \frac{(N-i+1)^\alpha}{\beta + \sum_{j=1}^{i-1} t_j + t}$

# Littlewood (L) Model

This model( Littlewood, 1981), is an attempt to answer the criticism of JM . The major drawback of JM is that it treats debugging as a deterministic process: each fix is effective with certainty and all fixes have the same effect on the reliability. In detail, the model assumes, as before

$$f(t_k|\Lambda_k = \lambda_k) = \lambda_k e^{-\lambda_k t_k} \quad (1)$$

where the random variable  $\{\Lambda_k\}$  represent the successive ROCOFs arising from the gradual elimination of faults. Here

$$\Lambda_k = \Phi_1 + \Phi_2 + \dots + \Phi_{N-k+1} = \sum_{j=1}^{N-k+1} \Phi_j \quad (2)$$



where  $\Phi_0$  is the initial number of faults and  $\Phi_j$  represents the (random variable) rate associated with fault  $j$  (in arbitrary labelling).

The initial rates  $\Phi_1, \dots, \Phi_N$  are assumed to be independent, identically distributed gamma( $\alpha, \beta$ ).

$$f(\phi) = \text{gamma}(\alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \phi^{\alpha-1} e^{-\beta\phi}, \quad \phi \in (0, \infty) \quad (3)$$

When the program has executed for a total time  $x$ , use of Bayes' theorem shows that the *remaining rates* are independent identically distributed gamma( $\alpha, \beta+x$ ) random variables.

and MTTF :

$$E[T_k] = \int_0^{\infty} t f(t|x_{k-1}) dt = \frac{\beta + x_{k-1} + t}{[(N-k+1)\alpha - 1]} \quad (11)$$

which does exist under the condition:

$$(N - k + 1)\alpha - 1 > 0 \quad (12)$$

The median is:

$$\tau_{n+1} = (\beta + x_n) \left[ e^{\frac{\ln 2}{(N-n)\alpha}} - 1 \right] \quad (13)$$

The restrictions for  $N$ ,  $\alpha$ ,  $\beta$  are :

$$N \in \mathbf{N} , \alpha > 0 , \beta > 0 , N \geq n \quad (14)$$

The ML

$$\ln L = \sum_{k=1}^n [\ln(N - k + 1)] - (\alpha + 1) \sum_{k=1}^n \ln(\beta + x_k) + n \ln \alpha + N \alpha \ln \beta - (N - n) \alpha \ln(\beta + x_n) \quad (15)$$

and the equations:

$$\begin{aligned} \frac{\partial \ln L}{\partial \alpha} &= \frac{n}{\alpha} + N \ln \beta - (N - n) \ln(\beta + x_n) - \sum_{k=1}^n \ln(\beta + x_k) = 0 \\ \frac{\partial \ln L}{\partial \beta} &= N \frac{\alpha}{\beta} - (N - n) \frac{\alpha}{\beta + x_n} - (\alpha + 1) \sum_{k=1}^n \frac{1}{\beta + x_k} = 0 \\ \frac{\partial \ln L}{\partial N} &= \alpha \ln \frac{\beta}{\beta + x_n} + \sum_{k=1}^n \frac{1}{N - k + 1} = 0 \end{aligned} \quad (16)$$

We can eliminate  $\alpha$  from the first equation and the problem is reduced to a system of 2 equations. Using the following parameters  $N, \alpha, \beta$  we can estimate the following functions:  $f_{n+1}(t), R_{n+1}(t), \lambda_{n+1}(t), h(x), \tau_{n+1}$ .

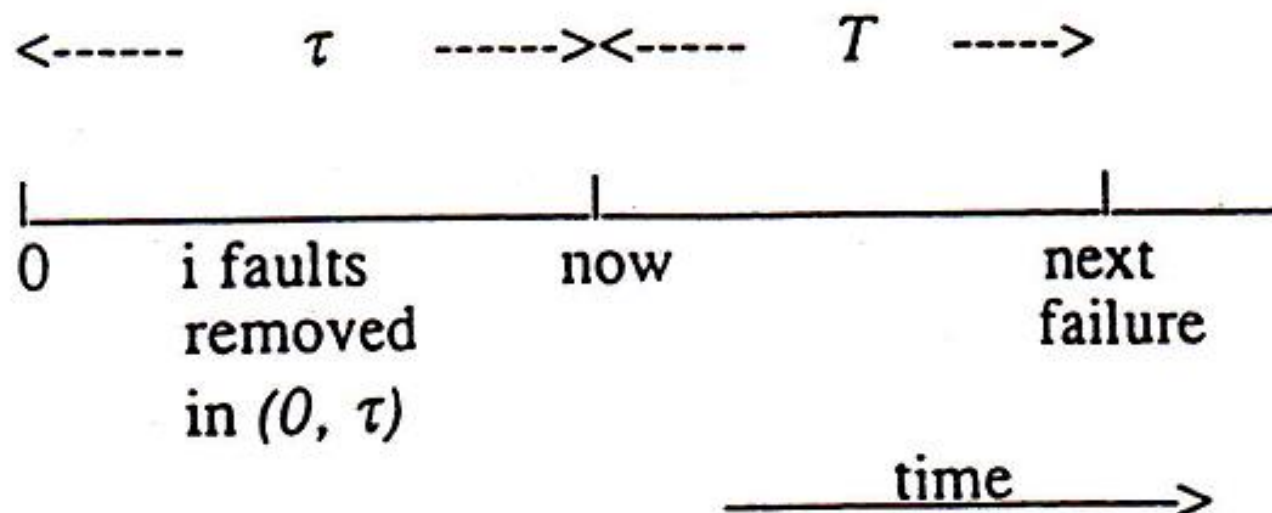
# [L] - LS Estimates

The LSE are those  $N, \alpha, \beta$  chosen to minimize

$$S(N, \alpha, \beta) = \sum_{i=1}^n \left[ X_i - \frac{\beta + t_{i-1}}{(N-i+1)\alpha - 1} \right]^2.$$

The system of equations to be solved in order to find LS estimates is:

$$\begin{cases} \sum_{i=1}^n \frac{X_i}{(N-i+1)\alpha - 1} = \sum_{i=1}^n \frac{\beta + t_{i-1}}{[(N-i+1)\alpha - 1]^2}, \\ \sum_{i=1}^n \frac{X_i(\beta + t_{i-1})}{[(N-i+1)\alpha - 1]^2} = \sum_{i=1}^n \frac{(\beta + t_{i-1})^2}{[(N-i+1)\alpha - 1]^3}, \\ \sum_{i=1}^n \frac{X_i(N-i+1)(\beta + t_{i-1})}{[(N-i+1)\alpha - 1]^2} = \sum_{i=1}^n \frac{(N-i+1)(\beta + t_{i-1})^2}{[(N-i+1)\alpha - 1]^3}. \end{cases}$$



Key idea:

- *faults removed in  $(0, \tau)$  will tend to be bigger than the ones remaining*
  - a "law of diminishing returns"

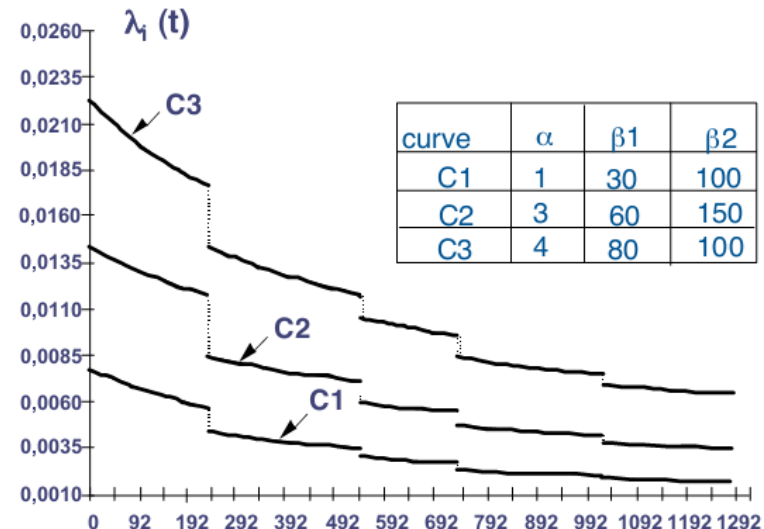


# Littlewood-Verrall [LV] Model

- *Stochastic relationship* between the successive failure rates. During correction it is possible to remove a fault or to introduce a new one.
- *Randomness of inputs*:  $f(t_i | \lambda_i) = \lambda_i \exp(-\lambda_i t_i)$ , the probability density function for  $\lambda_i$  is  $\Gamma(\alpha, \Psi(i))$ , where  $\Psi(i)$  captures the programming difficulty and the programmer skills. Usually,  $\Psi(i) = \beta_1 + \beta_2 i$ .
- Parameters:  $\alpha, \beta_1, \beta_2$
- Relations:

$$\lambda_i(t) = \alpha / (t + \Psi(i)),$$

$$MTTF_i = \Psi(i) / (\alpha - 1).$$





# [LV] – LS estimates

The LSE's are obtained by the minimization of:

$$S(\alpha, \beta_1, \beta_2) = \sum_{i=1}^n (X_i - E\{X_i\})^2.$$

In the case of linear assumptions for  $\psi(i)$ , the following system of equations is necessary to be solved:

$$\left\{ \begin{array}{l} \frac{\partial S}{\partial \alpha} = \alpha - 1 - \frac{\sum_{i=1}^n \psi^2(i)}{\sum_{i=1}^n X_i \psi(i)} = 0, \\ \frac{\partial S}{\partial \beta_1} = \sum_{i=1}^n X_i - \frac{n\beta_1}{\alpha - 1} - \frac{\beta_2 n(n+1)}{2(\alpha - 1)} = 0, \\ \frac{\partial S}{\partial \beta_2} = \sum_{i=1}^n X_i i - \frac{n(n+1)\beta_1}{2(\alpha - 1)} - \frac{\beta_2 \sum_{i=1}^n i^2}{\alpha - 1} = 0. \end{array} \right.$$

# Keiller-Littlewood [KL]

[KL] is similar to [LV], except that reliability growth is induced via the shape parameter of gamma distribution for the rates:

$$f(\lambda_i) = \frac{\beta^{\psi(i)} \lambda_i^{\psi(i)-1} e^{-\beta \lambda_i}}{\Gamma(\psi(i))}$$

$$\psi(i) = \psi(i, \alpha) = \frac{1}{\alpha_1 + \alpha_2 i}$$

Predictions of the unknown parameters can be obtained by Maximum Likelihood approach.

# The Poisson model (time related)

- The main hypotheses are the following:
  1.  $N(0) = 0$
  2. The occurrence of an fault is independent of previous faults; the future is independent of the past
  3. Not more than one fault can occur in the time interval  $(t, t+dt)$ ; simultaneous events are ‘impossible’
  4. The rate of occurrence of failures (ROCOF) is

$$\lim_{dt \rightarrow 0} \frac{P[1 \text{ event in } (t, t + dt)]}{dt} = \lambda(t)$$

# The NHPP distribution

- The occurrence of faults are described by the non-homogeneous (NHPP) distribution:

$$P[N(t) = n] = \frac{m(t)^n}{n!} e^{-m(t)}$$

where  $m(t)$  is the mean (expected) number of faults occurring in the interval  $(0, t)$ :

$$m(t) = \int_0^t \lambda(s) ds$$

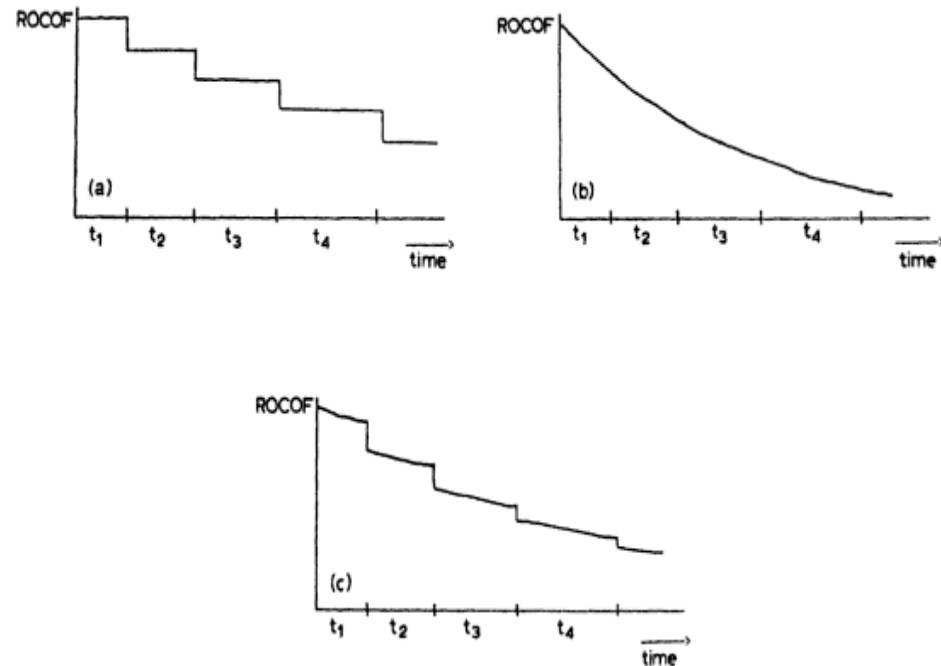
# Goel and Okumoto [GO]

- GO model is a NHPP variant of JM model.
- ROCOF is

$$\lambda(t) = \mu\phi \exp(-\phi t),$$

where  $t$  is total elapsed time since debugging began,  $\mu$  is the final number of faults that can be detected by the testing process, and  $\phi$  is a constant of proportionality, can be interpreted as the *failure occurrence rate per fault*.

- Prediction involves ML estimation of  $\mu$  and  $\phi$ , then substitution.



ROCOF plots for various models: (a) JM model; (b) NHPP models; (c) LV and L models.

# Goel-Okumoto [GO] – Assumptions (1)

- The software is operated in a similar manner as the anticipated operational usage.
- The number of errors ( $f_1, f_2, \dots, f_m$ ) detected in each of the respective time intervals  $(0, t_1), (t_1, t_2), \dots, (t_{m-1}, t_m)$  are independent for any finite collection of increasing sequence of times.
- Every error has the same chance of being detected and is of the same severity as any other error.



# Goel-Okumoto [GO] – Assumptions(2)

- The cumulative number of errors detected at any time  $t$ ,  $N(t)$ , follows a Poisson distribution with mean  $m(t)$ . The mean  $m(t)$  is such that the expected number of error occurrences for any time  $(t, t+\Delta t)$  is proportional to the expected number of undetected errors at time  $t$ .
- The expected cumulative number of errors function  $m(t)$  is assumed to be a bounded, nondecreasing function of  $t$  with  $m(t)=0, t=0$ ;  $m(t) = a, t$  goes to infinity:  $m(t)=a(1-\exp(-bt))$ ,  $b$  is a constant of proportionality.

## Goel and Okumoto model

This is a NHPP version of JM (although GO did not seem to realise this! [Goel and Okumoto 1979])

- ROCOF is  $\lambda(t) = \mu\phi e^{-\phi t}$  where  $t$  is total elapsed time since debugging began
- prediction involves ML estimation of  $\mu$  and  $\phi$ , then substitution

# [GO] – ML estimates

Let  $f_i = N(t_i) - N(t_{i-1})$ ,  $\Pr\{N(t) = n\} = m(t)^n \exp(-m(t)) / n!$ . The likelihood function is:

$$L(f_1, f_2, \dots, f_m) = \prod_{i=1}^m \frac{[m(t_i) - m(t_{i-1})]^{f_i} \exp\{m(t_{i-1}) - m(t_i)\}}{f_i!}$$

The MLEs are the solutions of a system based on the following relations:

$$0 = \frac{\partial \ln L}{\partial a} = \frac{\sum_{i=1}^m f_i}{a} - (1 - e^{-bt_m}),$$

$$0 = \frac{\partial \ln L}{\partial b} = \sum_{i=1}^m \frac{f_i t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}}}{e^{-bt_{i-1}} - e^{-bt_i}} - at_m e^{-bt_m}.$$

## Duane model

Originally a model for *hardware* reliability growth. Duane [Duane 1964] claims to have observed that a plot of cumulative number of failures against elapsed time on *log-log* paper was linear. This empirical observation about the ROCOF was taken up by Crow [Crow 1977] with the addition of an NHPP assumption.

- $ROCOF = \alpha \beta t^{\beta-1}$  (decreasing if  $\beta < 1$ , constant if  $\beta = 1$ , otherwise increasing)
- prediction by substitution of ML estimates, but *these are available in closed form* so no need for search routine unlike previous NHPPs

## Musa and Okumoto model [Musa and Okumoto 1984]

This is an attempt take account of faults having different rates, i.e. later fixes have less effect than earlier ones.

- ROCOF is  $\alpha/(\beta+t)$
- prediction by substitution of ML estimates of parameters



## Littlewood NHPP model

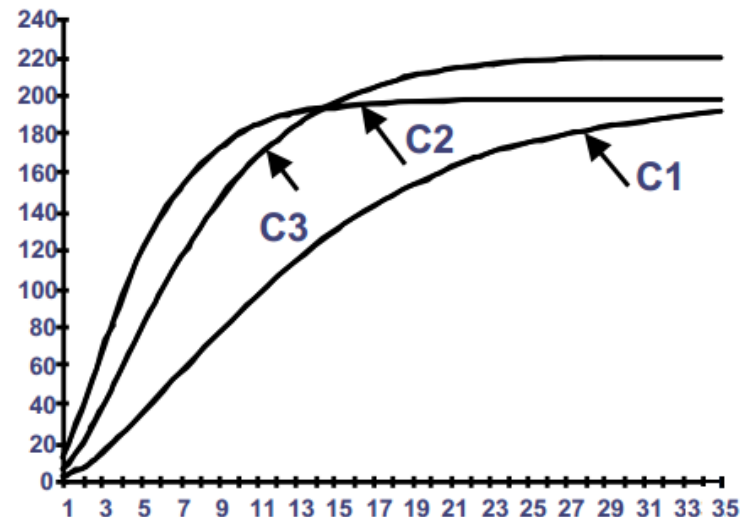
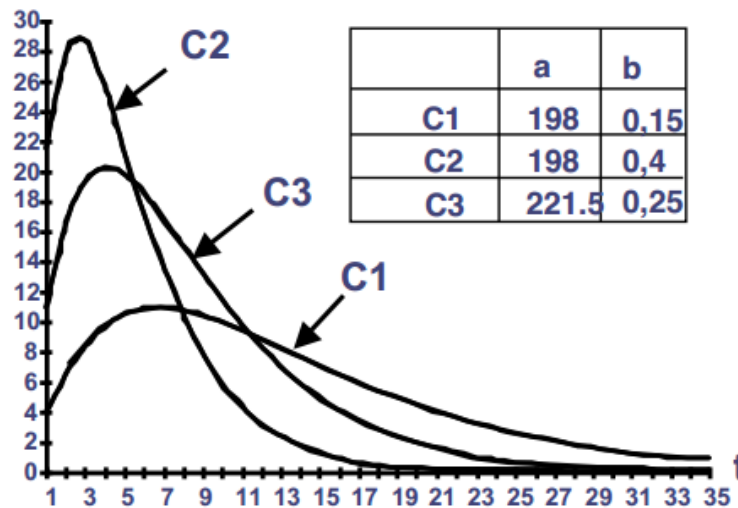
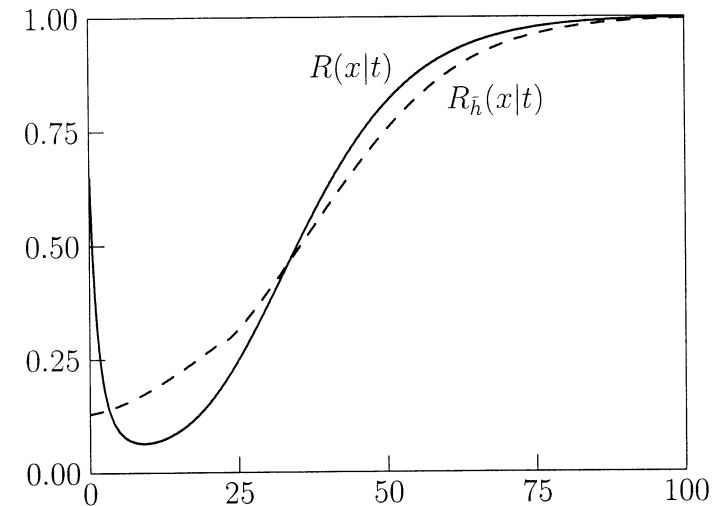
NHPP version of L

- ROCOF is  $\lambda(t) = \mu\alpha\beta^\alpha/(\beta+t)^{\alpha+1}$
- prediction by substitution of ML estimates of parameters



# NHPP – S Shaped Model

- Failure intensity:  
 $\lambda(t) = a b^2 t \exp(-bt)$
- Parameters to be estimated:  $a, b$
- Cumulative number of failures:  
 $M(t) = a[1 - (1 + bt)\exp(-bt)]$



# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
3. Measuring Software Reliability
- 4. Software Reliability Techniques and Tools**
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
7. Lessons Learnt
8. Background Literature

## 4. Software Reliability Techniques and Tools

- Kolmogorov-Smirnov (KS) Test
- The U-plot
- The Y-plot
- The Prequential Likelihood Ratio
- The Laplace test, Running Average, TTT, MIL HD Test, Noise
- Recalibration
- Combination of predictions

# Kolmogorov-Smirnov (KS) Statistics

- Uses the absolute vertical distance between two CDFs to measure goodness of fit.
- Depends on the fact that:

$$D_n = \sup_{-\infty < x < \infty} |\hat{F}_n(x) - F_0(x)|$$

where  $F_0$  is a known, continuous CDF, and  $\hat{F}_n$  is the sample CDF, is distribution free.

(CDF – Cumulative Distribution Function)

- $D_n$  is independent on  $\hat{F}_n$

# Critical Values for KS-test:

If  $D_n$  is less than the established criteria, the model fits the data adequately.

| Sample size<br>$n$ | Level of a significance for $D_n = \sup \hat{F}_n(x) - F_0(x) $ |                 |                 |                 |
|--------------------|---|-----------------|-----------------|-----------------|
|                    | $\alpha = 0.20$   | $\alpha = 0.10$ | $\alpha = 0.05$ | $\alpha = 0.01$ |
| 1                  | 0.9000  | 0.9500          | 0.9750          | 0.9950          |
| 2                  | 0.6838  | 0.7764          | 0.8419          | 0.9293          |
| 3                  | 0.5648  | 0.6360          | 0.7076          | 0.8290          |
| 4                  | 0.4927  | 0.5652          | 0.6239          | 0.7342          |
| 5                  | 0.4470  | 0.5095          | 0.5633          | 0.6685          |
| 6                  | 0.4104  | 0.4680          | 0.5193          | 0.6166          |
| 7                  | 0.3815  | 0.4361          | 0.4834          | 0.5758          |
| 8                  | 0.3583  | 0.4096          | 0.4543          | 0.5418          |
| 9                  | 0.3391  | 0.3875          | 0.4300          | 0.5133          |
| 10                 | 0.3226  | 0.3687          | 0.4093          | 0.4889          |
| 11                 | 0.3083  | 0.3524          | 0.3912          | 0.4677          |
| 12                 | 0.2958  | 0.3382          | 0.3754          | 0.4491          |
| 13                 | 0.2847  | 0.3255          | 0.3614          | 0.4325          |
| 14                 | 0.2748  | 0.3142          | 0.3489          | 0.4176          |
| 15                 | 0.2659  | 0.3040          | 0.3376          | 0.4042          |
| 16                 | 0.2578  | 0.2947          | 0.3273          | 0.3920          |
| 17                 | 0.2504  | 0.2863          | 0.3180          | 0.3809          |
| 18                 | 0.2436  | 0.2785          | 0.3094          | 0.3706          |
| 19                 | 0.2374  | 0.2714          | 0.3014          | 0.3612          |
| 20                 | 0.2316  | 0.2647          | 0.2941          | 0.3524          |
| 25                 | 0.2079  | 0.2377          | 0.2640          | 0.3166          |
| 30                 | 0.1903  | 0.2176          | 0.2417          | 0.2899          |
| 35                 | 0.1766  | 0.2019          | 0.2243          | 0.2690          |
| Over 35            | $1.07/\sqrt{n}$   | $1.22/\sqrt{n}$ | $1.36/\sqrt{n}$ | $1.63/\sqrt{n}$ |

# The u-plot (1)

**The “u-plot” can be used to assess the predictive quality of a model**

- Given a predictor,  $\hat{F}_i(t)$ , that estimates the probability that the time to the next failure is less than  $t$ . Consider the sequence

$$u_i = \hat{F}_i(t_i)$$

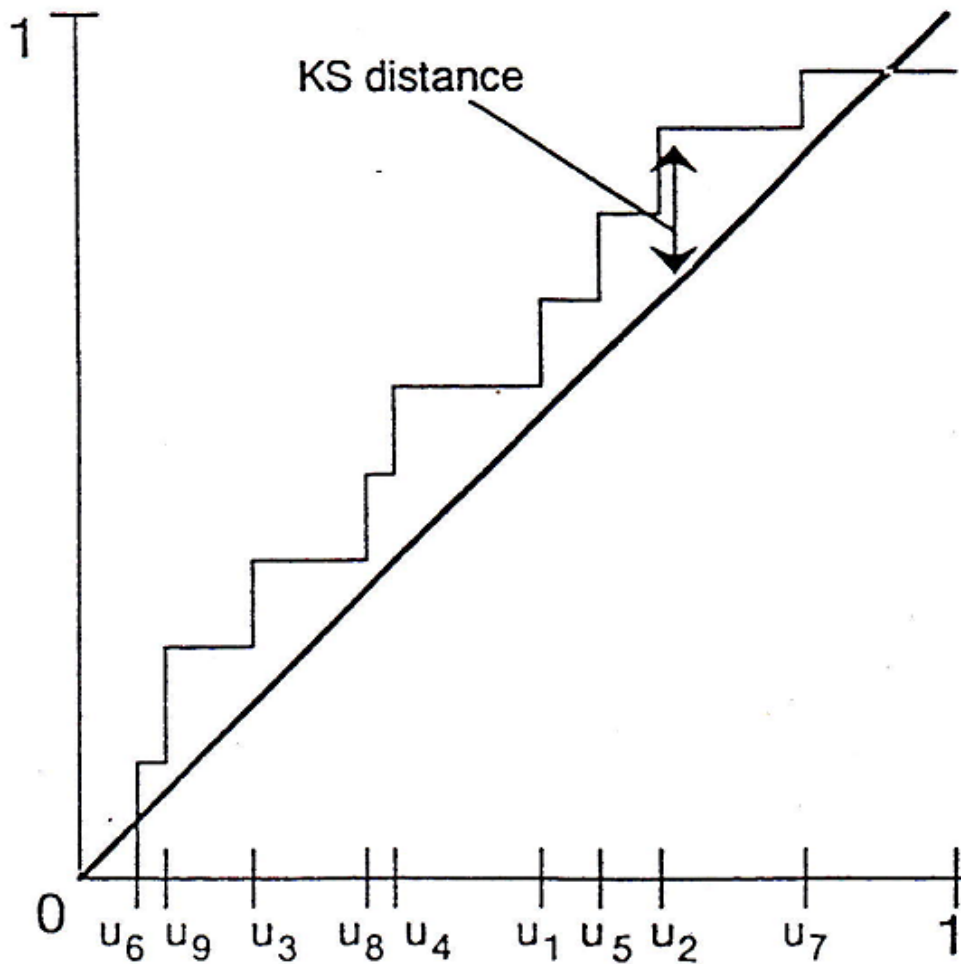
where each  $u_i$  is a probability integral transform of the observed  $t_i$  using the previously calculated predictor  $\hat{F}_i$  based upon  $t_1, t_2, \dots, t_{i-1}$ .



# The u-plot (2)

- If each  $\hat{F}_i$  were identical to the true  $F_i$  then the  $u_i$  would be realizations of independent random variables with a uniform distribution in  $[0,1]$ .
- **The simplest question to answer is whether their distribution is close to U by plotting their sample cdf: we call this the u-plot.**

# How to draw a u-plot



For  $n$   $u_i$  s, step  
size is  $\frac{1}{n+1}$   
(here  $n = 9$ )

# Y-plot

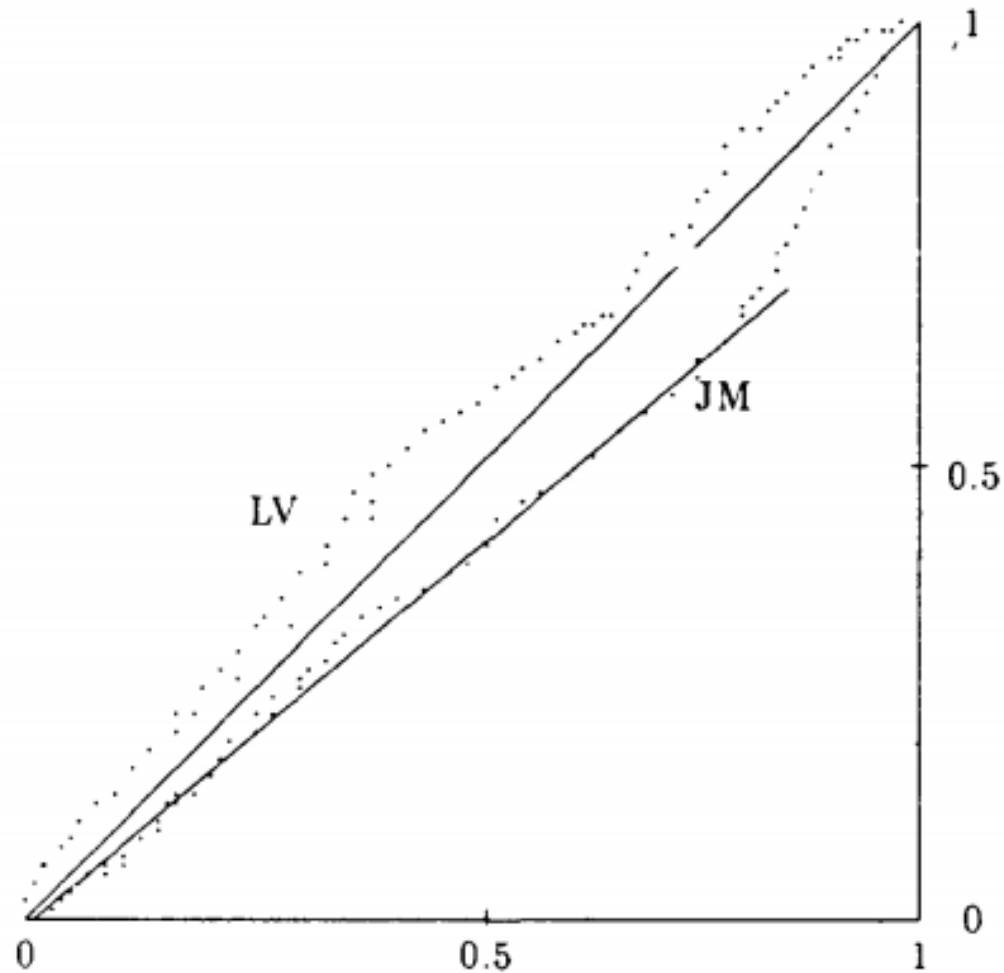
To preserve the temporal information lost in producing the bias plot, we have to perform two additional transformations after producing the random variable  $u_i$ . The transformation sequence is:

$$x_i = -\ln(1 - u_i) \quad (1)$$

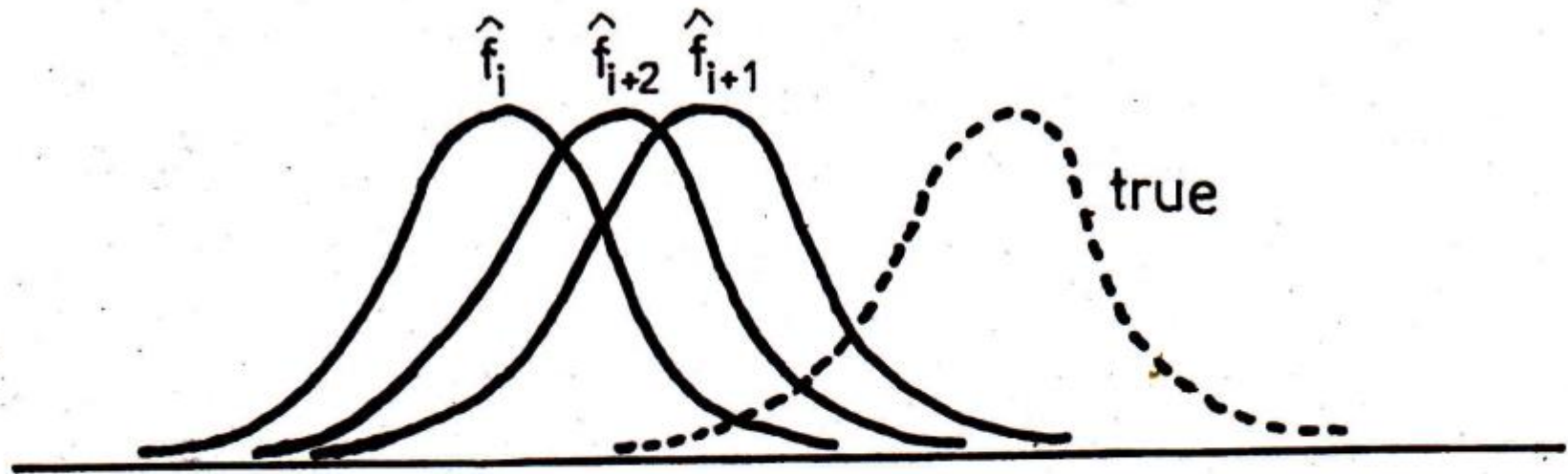
$$y_i = \frac{\sum_{j=1}^i x_j}{\sum_{j=1}^n x_j} \quad (2)$$

where  $n$  is the total number of failures observed. The cdf of the  $y_i$  is drawn, as was done for the  $u$ -plot. This  $y$ -plot reveals temporal trends in the  $u_i$ .

# The y-Plot for the LV and JM models (Littlewood, 1981)

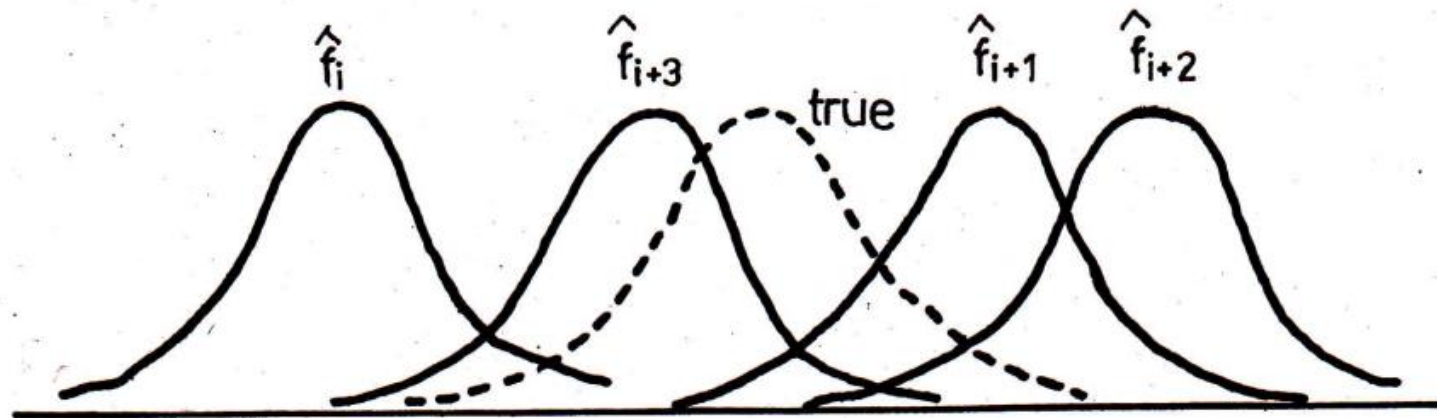


# Detecting *consistent* 'bias' and *inappropriate* 'noisiness' in a prediction system (1)



These predictions have high 'bias' and low 'noise'

# Detecting *consistent* 'bias' and *inappropriate* 'noisiness' in a prediction system (2)



These predictions have high 'noise' and low 'bias'



# Prequential Likelihood Ratio

- The pdf for  $\hat{F}_i(t)$  for  $T_i$  is based on observations  $t_1, t_2, \dots, t_{i-1}$ .  
 $\hat{f}_i(t) = d\hat{F}_i(t) / dt$

- For one-step ahead predictions of  $T_{j+1}, T_{j+2}, \dots, T_{j+n}$ , the prequential likelihood is:

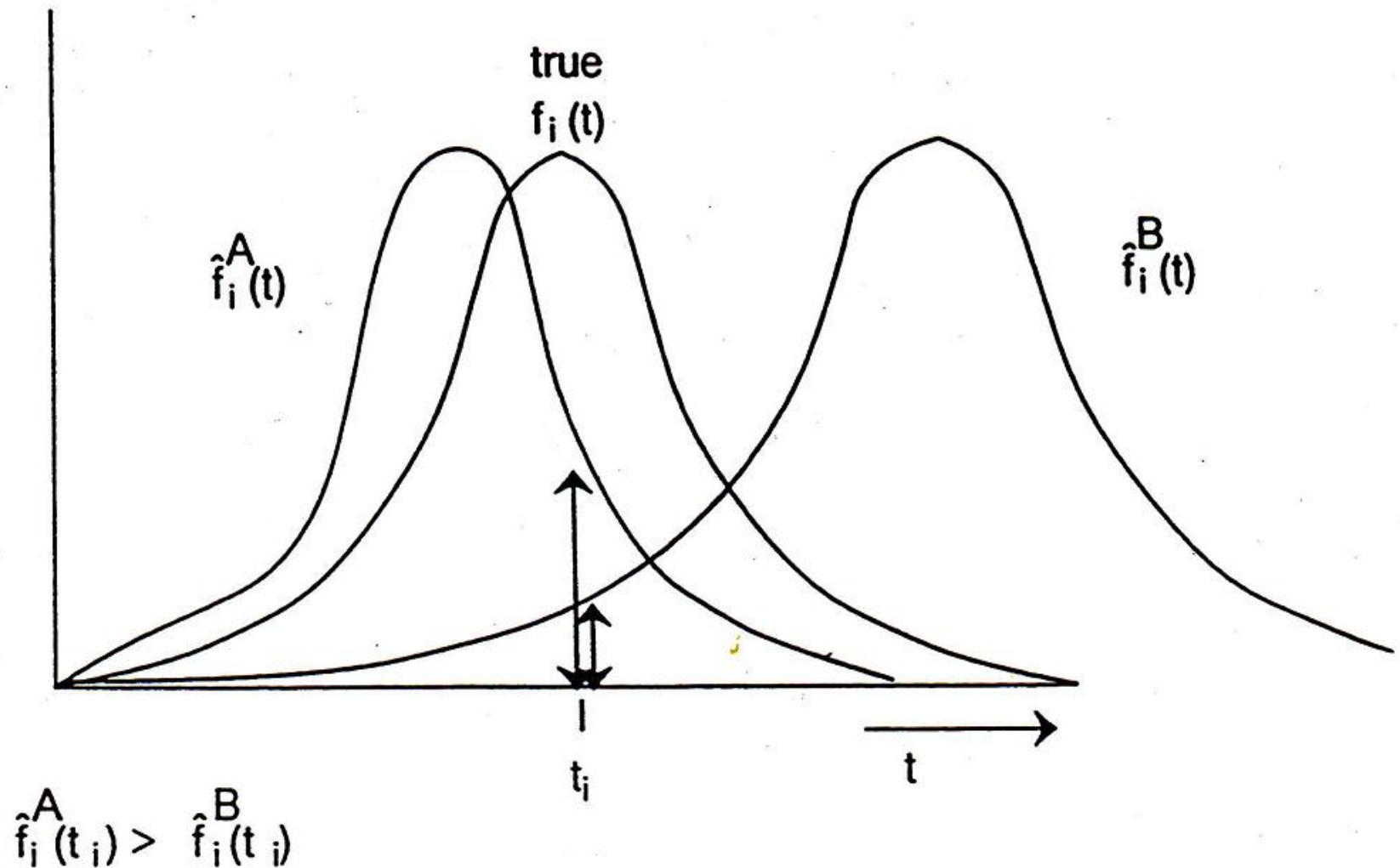
$$PL_n = \prod_{i=j+1}^{j+n} \hat{f}_i(t)$$

- Two prediction systems, A and B, can be evaluated by computing the prequential likelihood ratio:

$$PLR_n = \frac{\prod_{i=j+1}^{j+n} \hat{f}_i^A(t_i)}{\prod_{i=j+1}^{j+n} \hat{f}_i^B(t_i)}$$

- If  $PLR_n$  approaches infinity as  $n$  approaches infinity, B is discarded in favor of A.

# Using PLR as a device for comparing two prediction systems, A and B



# Reliability trend analysis – Laplace test

The expression for the Laplace test factor  $u(k)$  is:

$$u(k) = \frac{c - m}{s_k} \sqrt{12(k-1)}, k = 2, \dots, n$$

where

$$c = \frac{1}{k-1} \sum_{i=1}^{k-1} s_i$$

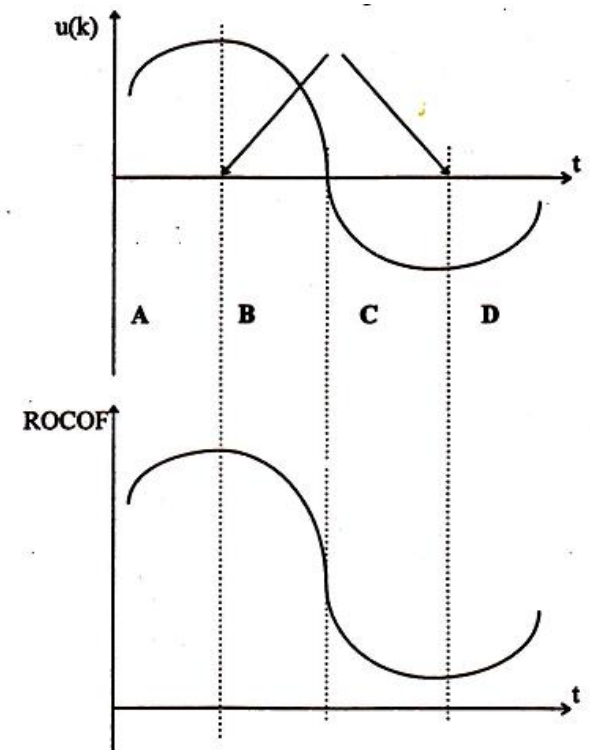
$$m = \frac{s_k}{2}$$

$$s_i = \sum_{j=1}^i t_j$$

$n$  = total number of failures

$t_k$  = interfailure time between failures  $k-1$  and  $k$

$s_i$  = time of occurrence of failure  $i$



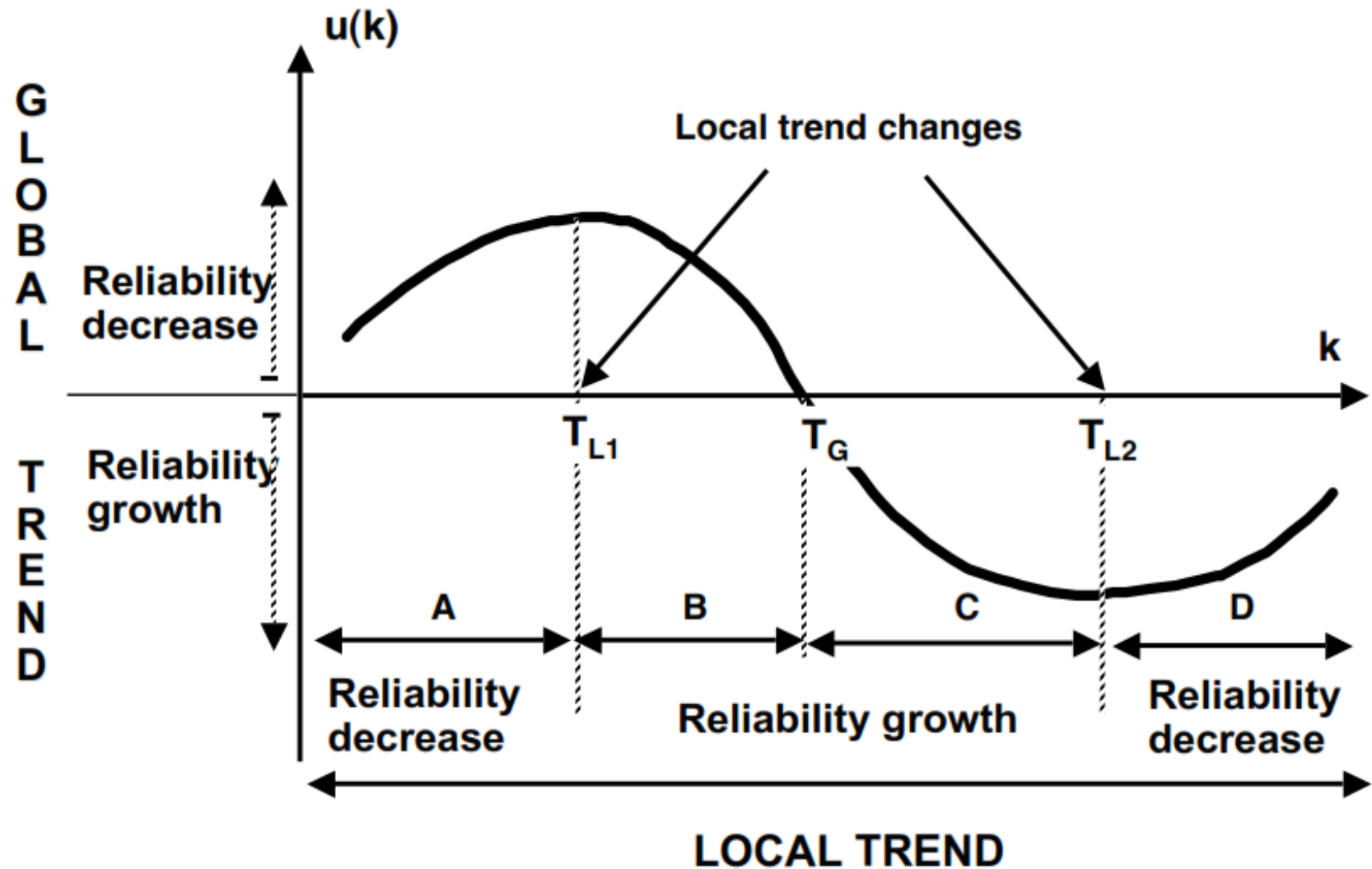
# Laplace test - Interpretation

This test analyses the trend of the failures. One can extract two types of information from such a graph : *local and global changes*.

When the values are *positive (resp. negative)*, the reliability is *globally increasing (resp. decreasing)*. On the other side, when the *values are increasing (resp. decreasing)*, we have *local variations* of the reliability.

- If  $U$  is approximately equal to zero, it indicates a lack of trend,
- If  $U$  is greater than zero, the TBFs are decreasing,
- If  $U$  is less than zero, the TBFs are increasing.

# General trend – Local trend



## Running Arithmetic Average of Time Between Failures/Failure Counts

For time between failures data, the running arithmetic average after the  $i^{th}$  failure has been observed,  $r(i)$ , is given by:

$$r(i) = \frac{\sum_{j=1}^i \theta_j}{i}, \quad (1)$$

where  $\theta_j$  is the observed time between the  $(j-1)^{st}$  and the  $j^{th}$  failures. For failure counts data, the running arithmetic average after the  $i^{th}$  test interval has been completed,  $r(i)$ , is given by:

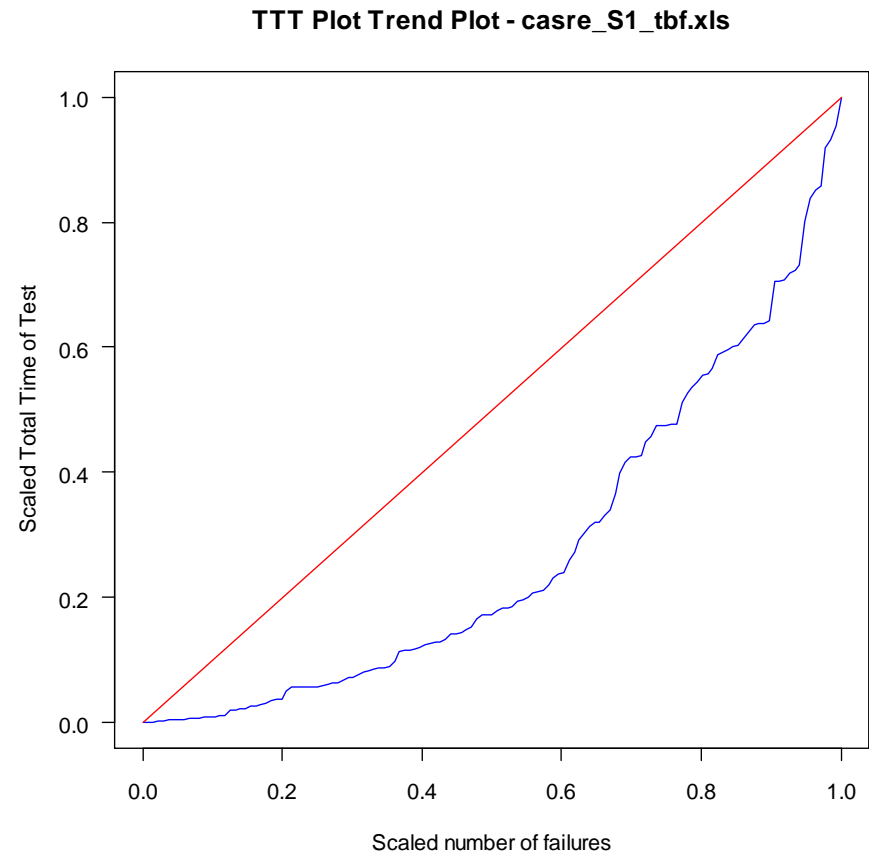
$$r(i) = \frac{\sum_{j=1}^i n_j}{i}, \quad (2)$$

where  $n_j$  is the number of failures that have been observed in the  $j^{th}$  test interval.



# TTT plot – *Total Time on Test*

- The TTT plot is basically a scaled version of the graph consisting of the points  $(i, T_i)$ .
- Is defined as a plot of the points  $(i/n, T_i/T_n)$ , for  $i=1, 2, \dots, n$ .
- A necessary but not sufficient condition for this notion of reliability growth is that the graph of the TTT plot should be below the diagonal.



# The MIL-HDBK-189 Test (1)

The MIL-HDBK-189 trend test is a conditional statistical test based on the power-law process:

$$\mu(t) = \frac{b}{a} \left( \frac{t}{a} \right)^{b-1}$$

where  $a$  and  $b$  are the model parameters which are positive.

If  $b < 1$  then  $\mu(t)$  decreases, meaning that the failures tend to occur less frequently (and the system shows reliability growth).

If  $b > 1$ , then the system shows reliability decrease. When  $b = 1$  the homogeneous Poisson process case is obtained.

# The MIL-HDBK-189 Test (2)

Considering the event  $\{T_n = t_n\}$ , the MLE of  $b$  of a failure truncated power-law process is given by

$$\hat{b} = \frac{n}{\sum_{i=1}^{n-1} \log(t_n / T_i)}$$

Under the null hypothesis of  $b=1$  it follows that  $2n/\hat{b} \sim \chi^2(2(n-1))$ . If the alternative hypothesis is two-sided, then the null hypothesis is rejected if

$$\hat{b} > \frac{2n}{\chi_{1-\alpha/2}^2(2(n-1))} \quad \text{or} \quad \hat{b} < \frac{2n}{\chi_{\alpha/2}^2(2(n-1))},$$

where  $\chi_{\gamma}^2(\nu)$  denotes the  $\gamma$ -percentile of the chi-squared distribution with  $\nu$  degrees of freedom. For large values of  $n$  the null hypothesis is rejected in favor of reliability growth.

# Model noise

Model noise is defined by the following equation:

$$\text{Model Noise} = \sum_i \left| \frac{t_i - t_{i-1}}{t_{i-1}} \right|$$

The quantity  $t_i$  is the prediction for the  $i^{\text{th}}$  time between failure made by the model.

# Recalibrating Software Reliability Models

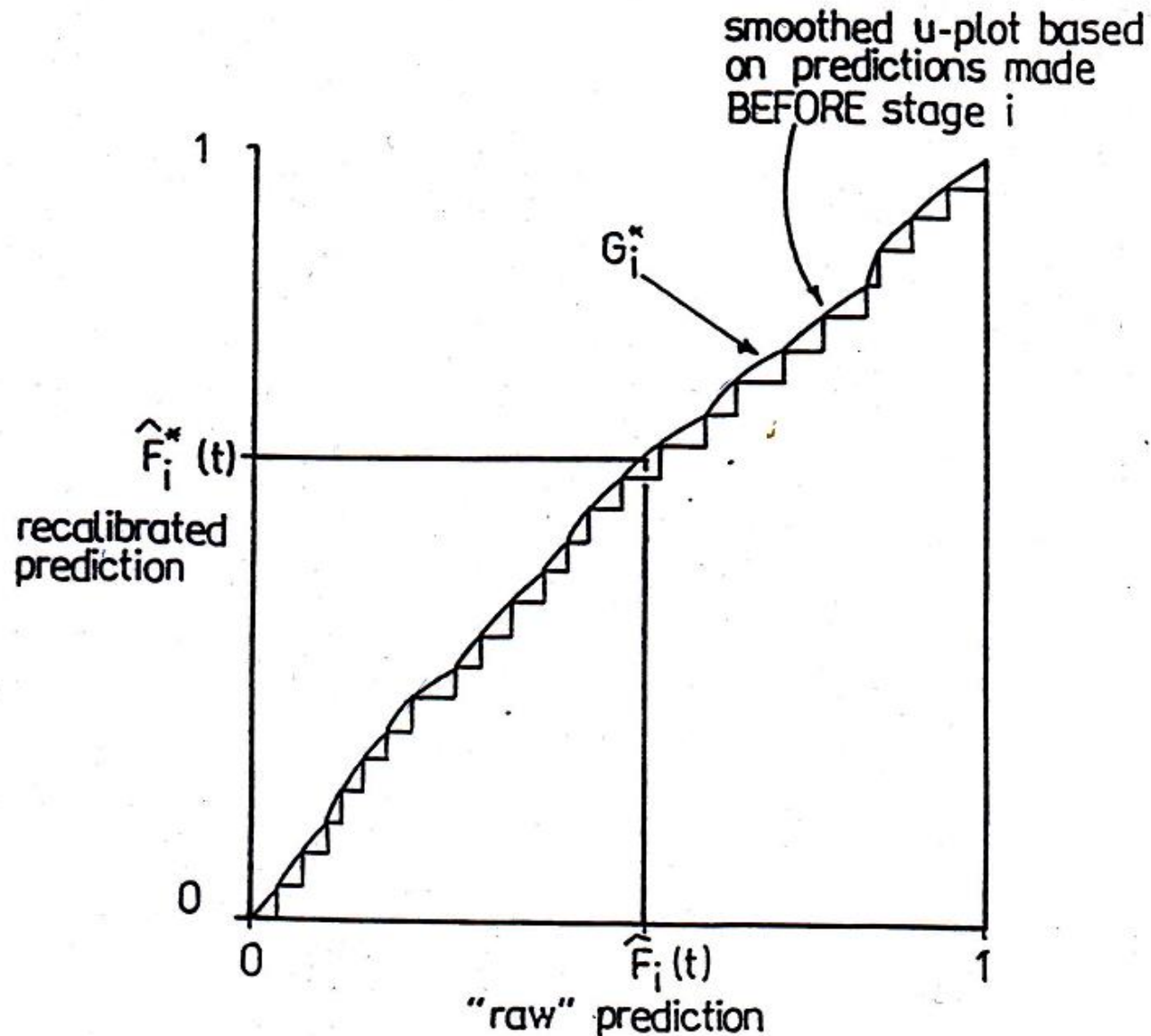
This method was introduced by Brocklehurst, Chan, Littlewood & Snell (NASA-CR-166407), and can be summarized as follows.

The relation between true distribution  $F_i(t)$  of the random variable  $T_i$ , and the predicted one,  $F_i(t)$ , can be represented through a relation function  $G_i$  as  $F_i(t) = G_i(F_i(t))$ , where  $G_i$  is only slowly changing function with  $i$ . Since  $G_i$  is not known, it will be approximated with an estimate  $G^*$  which will lead to a new prediction:

$$\hat{F}_i^*(t) = G_i^*(F_i(t))$$

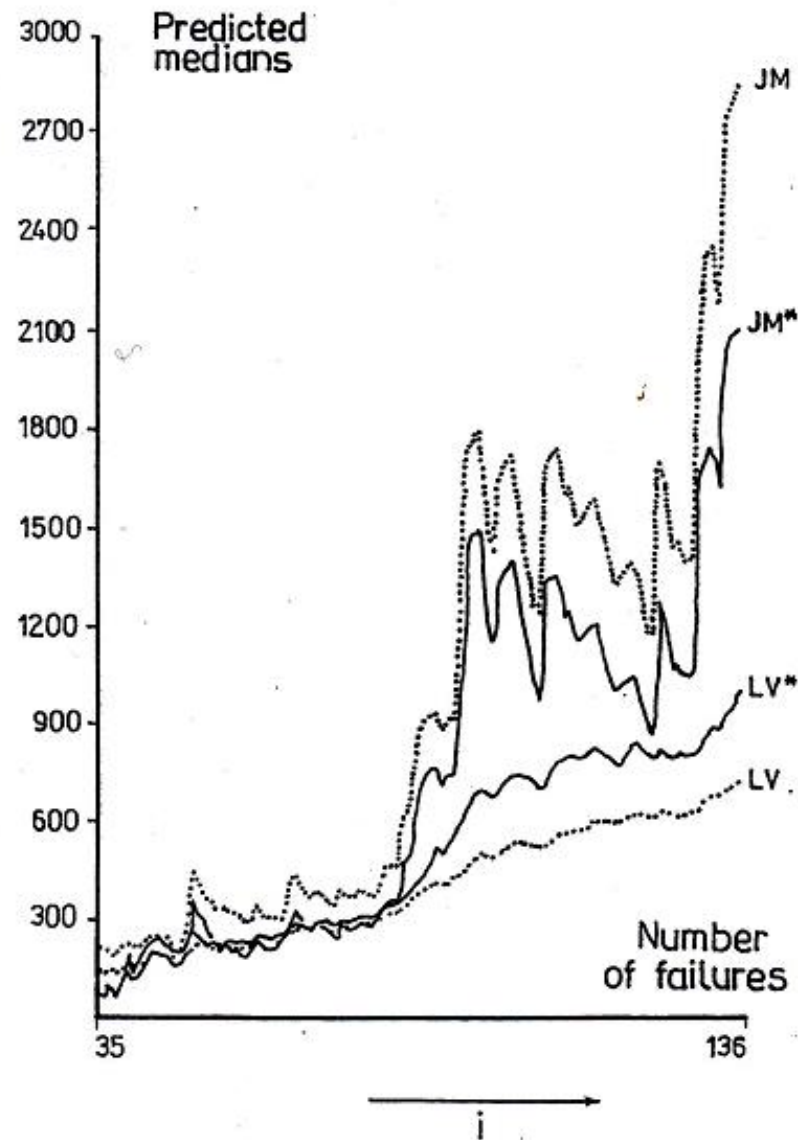
This technique recalibrates the raw model output  $F_i(t)$  related to the accuracy of past predictions.

Recalibration; use u-plot analysis of past predictions to improve future predictions





## Median predictions for JM, LV, JM\*, LV\* on data of Table 1



# Combination of predictions

- Our previous recommendation was: ‘pick the model which had performed best in the past, and use it for the next prediction’.
- This seems unduly ‘rejecting’ of models which are only slightly inferior to ‘the best’.
- Why not combine predictions from different models in some optimal way? cf pooling of ‘expert opinion’.
- For two candidate models, A and B, could take as a ‘n  $\hat{F}_j(t) = w_1 \hat{F}_j^A(t) + w_2 \hat{F}_j^B(t)$ ,  $w_1 + w_2 = 1$  than two.

# How should the weights be selected ‘optimally’?

- For a prediction at stage  $i$ , i.e. of  $t_i$ , let  $\{w_k\}$  take values that maximise the PL of the combined predictor over previous predictions

$$\begin{aligned} \text{i.e. } \max \prod_{j=1}^{i-1} \hat{F}_j(t) \\ = \max \prod_{j=1}^{i-1} (w_1 \hat{F}_j^A(t) + w_2 \hat{F}_j^B(t)) \end{aligned}$$

in the case of two prediction systems.

- This is computationally intensive, but seems to work quite well.

# How well do these work?

- Sometimes dramatic disagreement between model predictions on the same data source.
- No universally accurate model.
- No way of selecting a model a priori and being confident that it will be accurate on a particular data source.
- Remember we have a prediction triad: in principle we could separately examine models and inference procedures.
- In fact this is too difficult: we are forced to examine directly the accuracy of the different available models on each data source and somehow decide which, if any, is giving accurate results.

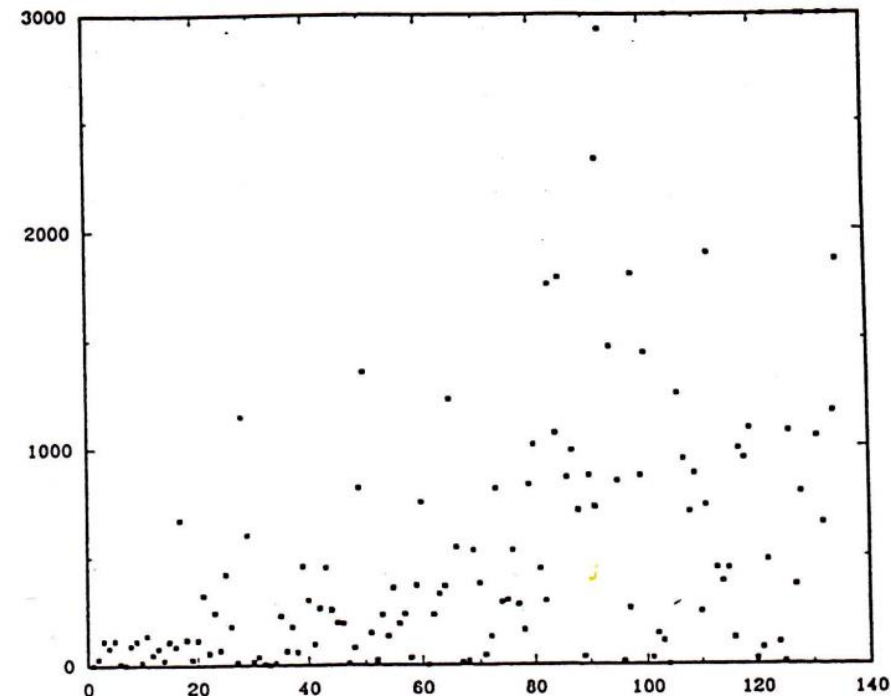
# Reliability models in practice - examples

Table 1. Data set - SYS1 (136 inter-failure times)

(read left to right)

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 3.    | 30.   | 113.  | 81.   | 115.  |
| 9.    | 2.    | 91.   | 112.  | 15.   |
| 138.  | 50.   | 77.   | 24.   | 108.  |
| 88.   | 670.  | 120.  | 26.   | 114.  |
| 325.  | 55.   | 242.  | 68.   | 422.  |
| 180.  | 10.   | 1146. | 600.  | 15.   |
| 36.   | 4.    | 0.    | 8.    | 227.  |
| 65.   | 176.  | 58.   | 457.  | 300.  |
| 97.   | 263.  | 452.  | 255.  | 197.  |
| 193.  | 6.    | 79.   | 816.  | 1351. |
| 148.  | 21.   | 233.  | 134.  | 357.  |
| 193.  | 236.  | 31.   | 369.  | 748.  |
| 0.    | 232.  | 330.  | 365.  | 1222. |
| 543.  | 10.   | 16.   | 529.  | 379.  |
| 44.   | 129.  | 810.  | 290.  | 300.  |
| 529.  | 281.  | 160.  | 828.  | 1011. |
| 445.  | 296.  | 1755. | 1064. | 1783. |
| 860.  | 983.  | 707.  | 33.   | 868.  |
| 724.  | 2323. | 2930. | 1461. | 843.  |
| 12.   | 261.  | 1800. | 865.  | 1435. |
| 30.   | 143.  | 108.  | 0.    | 3110. |
| 1247. | 943.  | 700.  | 875.  | 245.  |
| 729.  | 1897. | 447.  | 386.  | 446.  |
| 122.  | 990.  | 948.  | 1082. | 22.   |
| 75.   | 482.  | 5509. | 100.  | 10.   |
| 1071. | 371.  | 790.  | 6150. | 3321. |
| 1045. | 648.  | 5485. | 1160. | 1864. |
| 4116. |       |       |       |       |

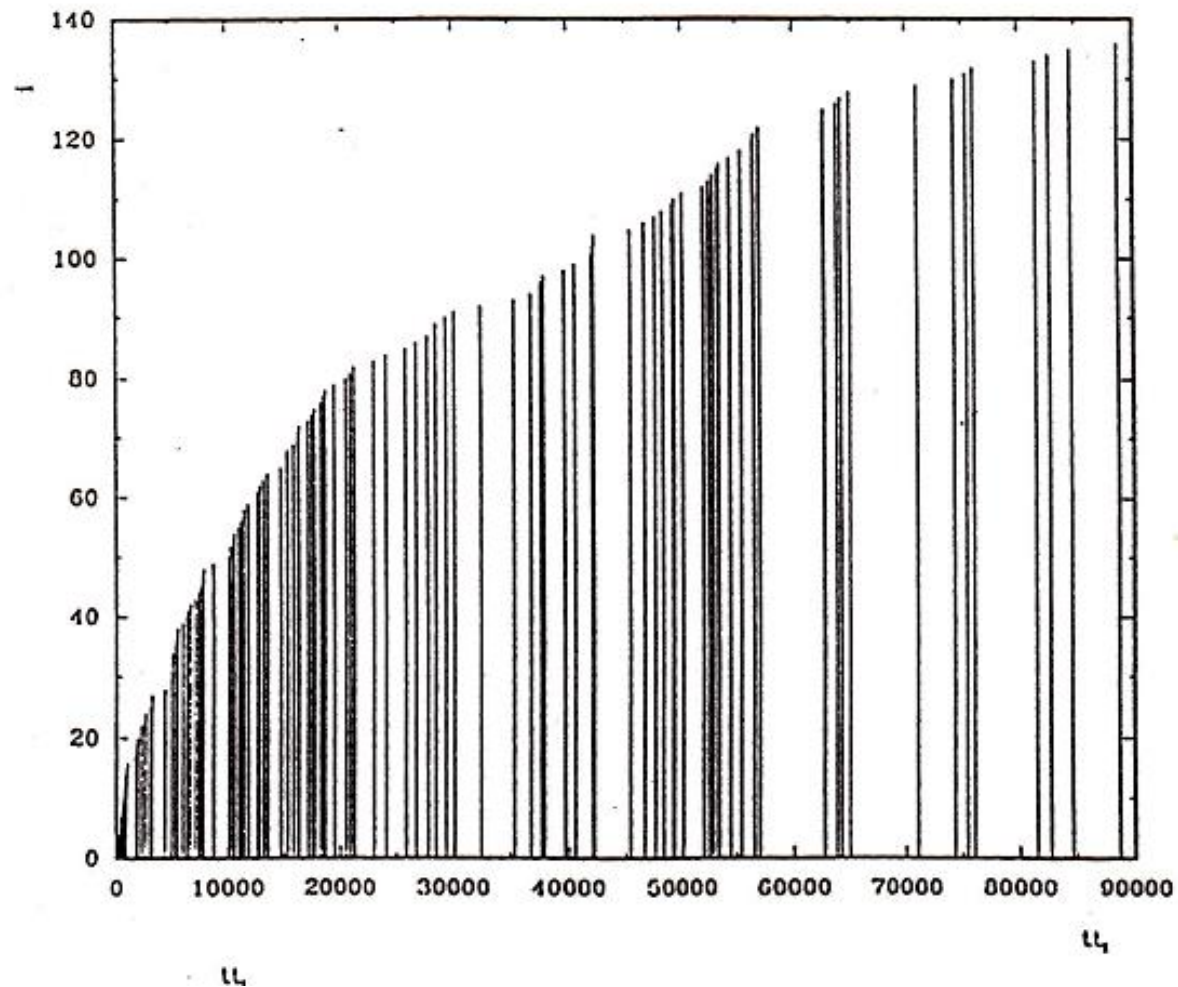
Inter-failure time vs failure no : sys1





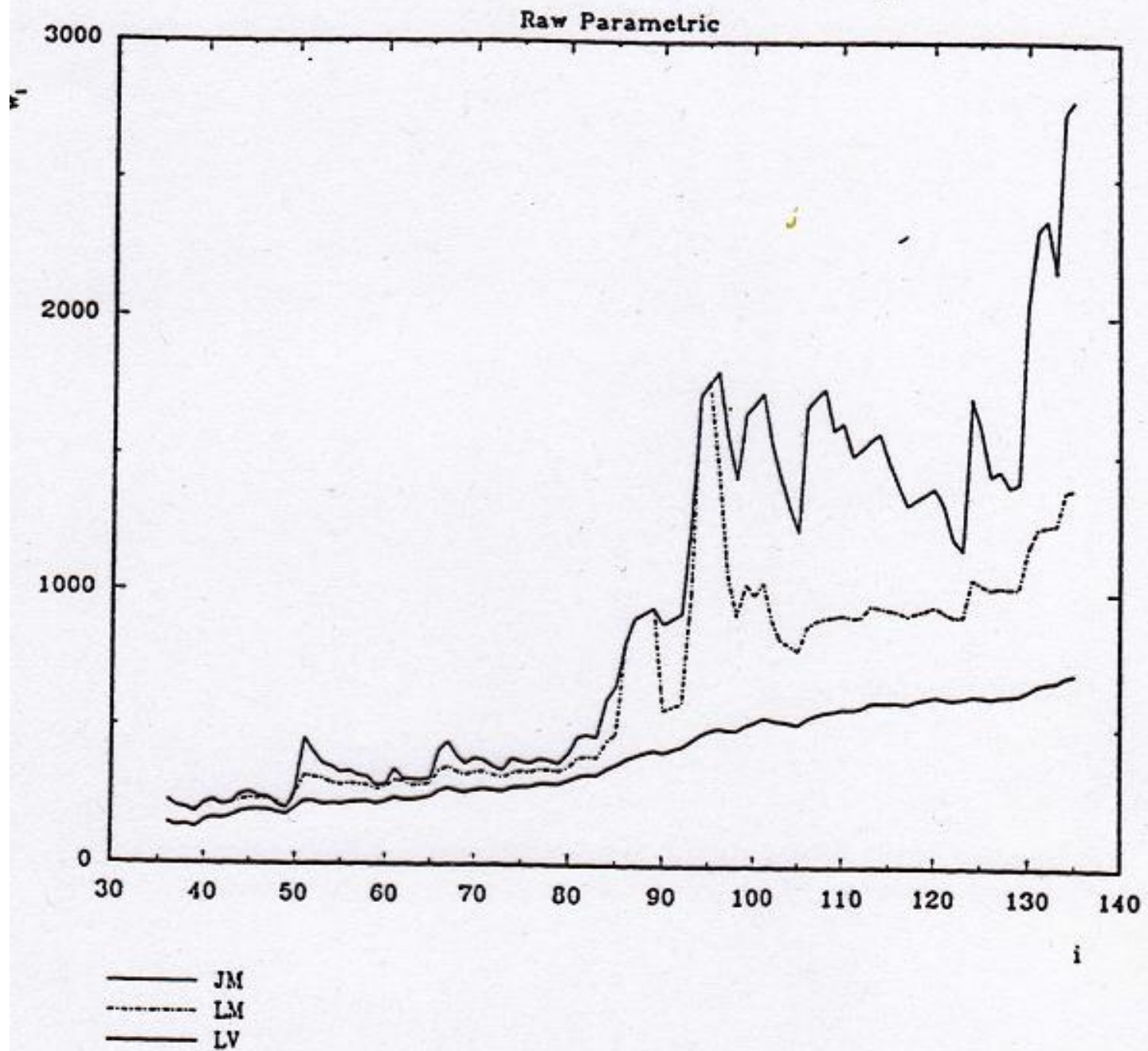
# Data set: Sys1

Cumulative failures vs total elapsed time : sys1



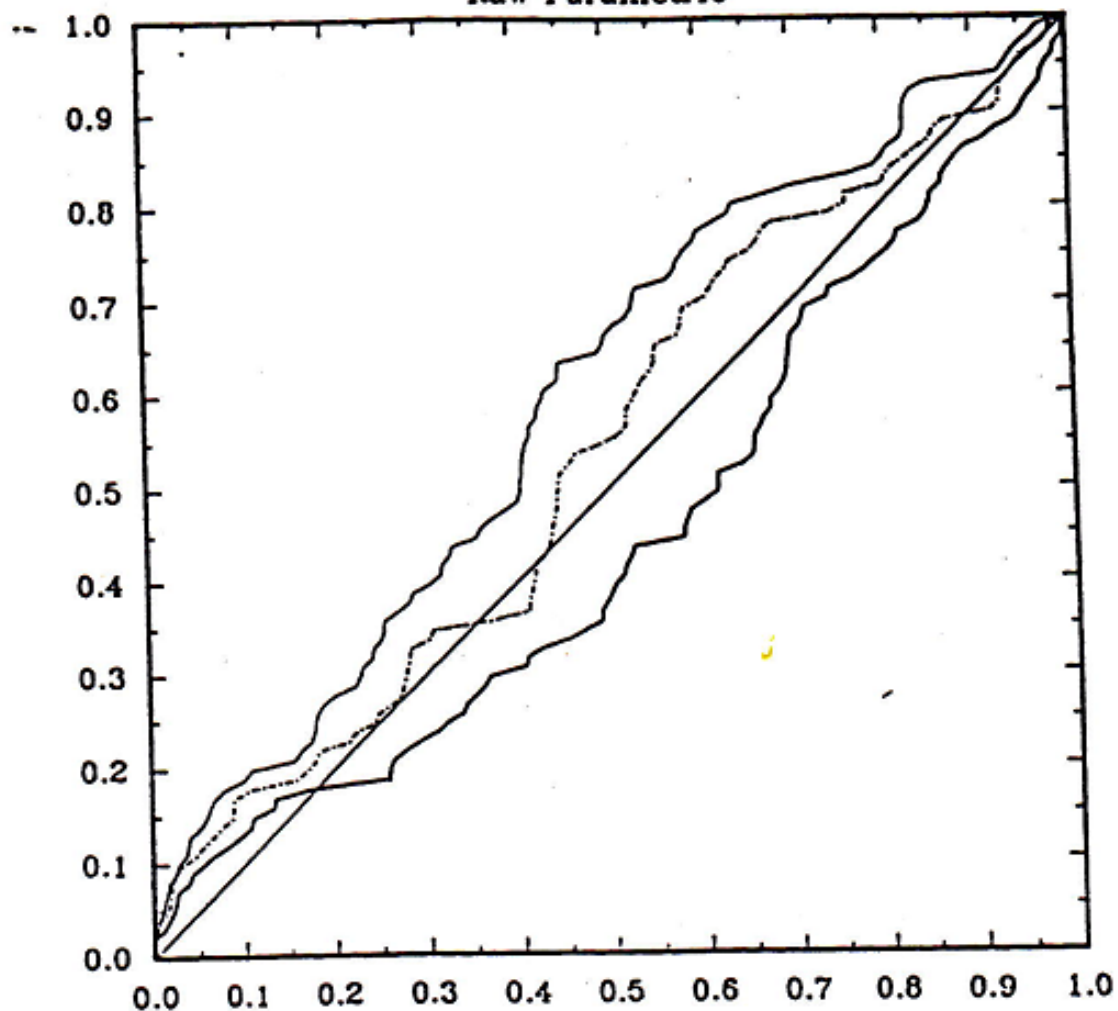


# Medians versus i for data set sys1



# u-plot for data set sys1

Raw Parametric

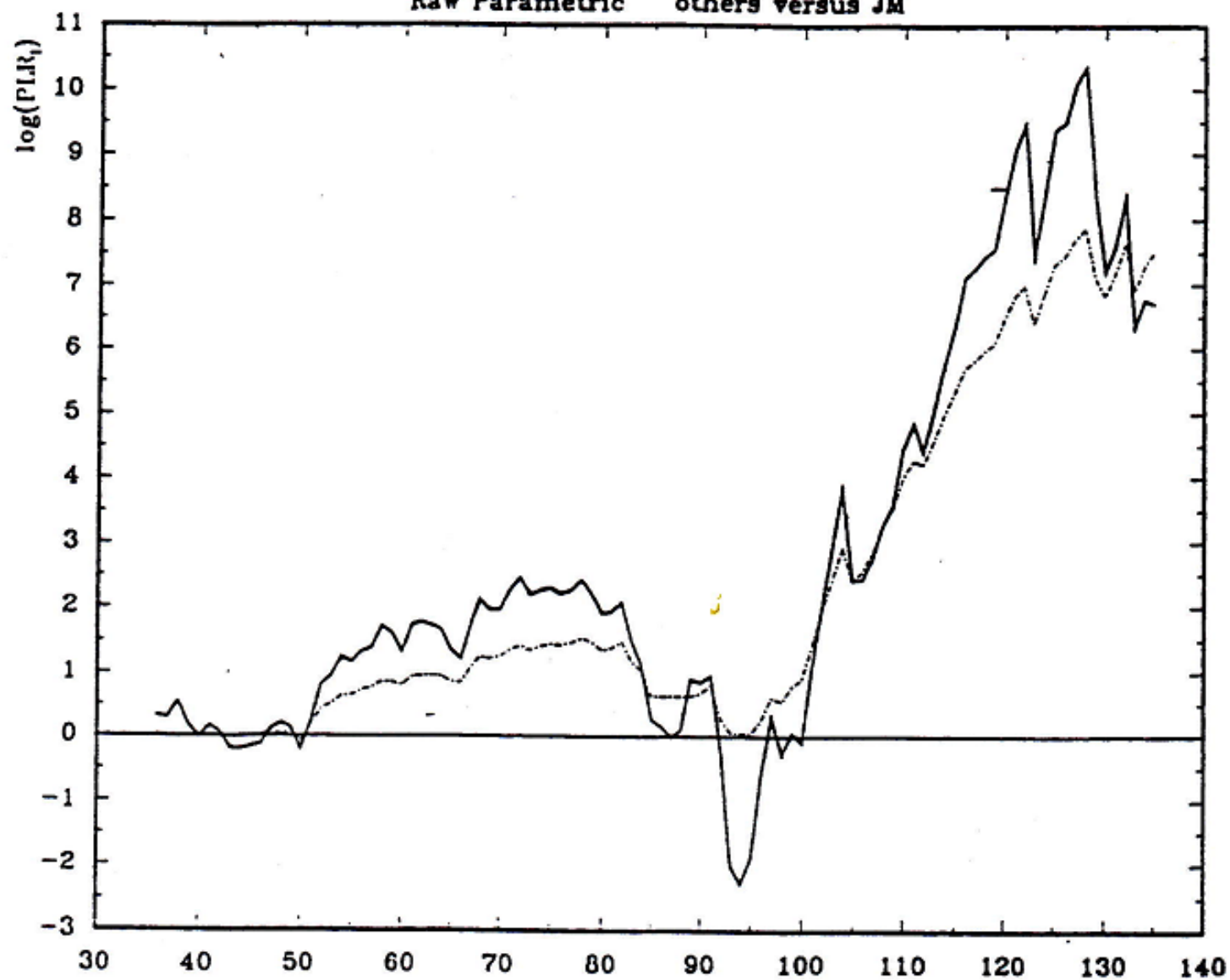


ks distance siglev

|        |              |        |
|--------|--------------|--------|
| — JM   | 1.873824e-01 | >1%    |
| - - LM | 1.095623e-01 | 10-20% |
| — LV   | 1.436834e-01 | 2-5%   |

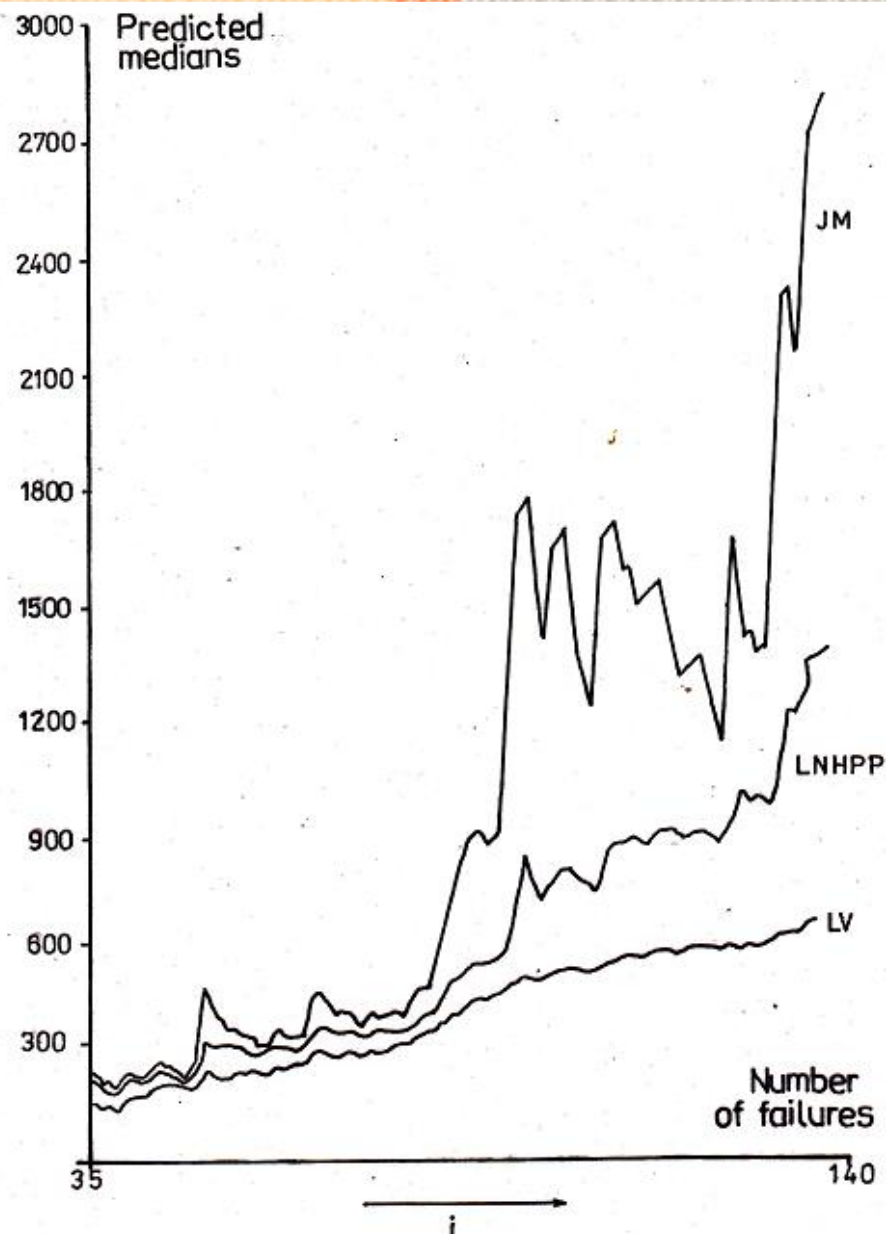
# Log(PLR) versus i for data set sys1

Raw Parametric    others versus JM

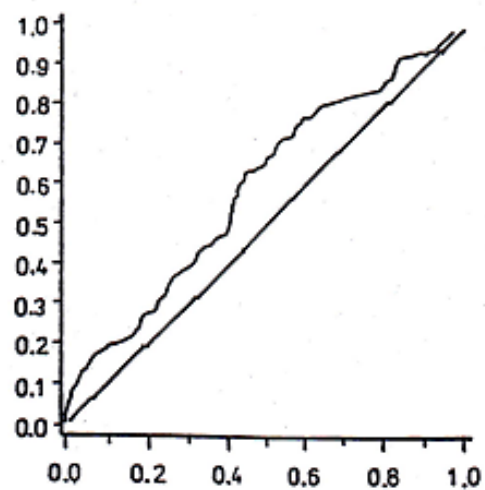


— JM  
- - LM  
— IV

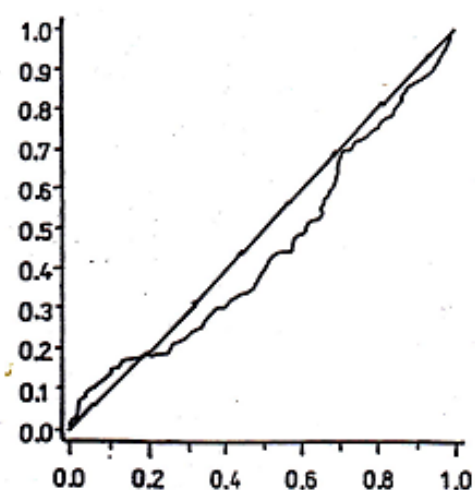
Median plots for JM, LNHPP, LV, data from Table 1. Plotted here are predicted median of  $T_i$  ( based on  $t_1, t_2, \dots, t_{i-1}$  ) against  $i$



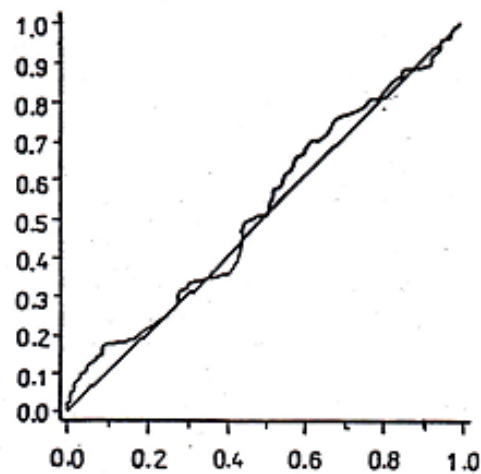
u-Plots for the data of Table 1 : (a) JM, maximum deviation 0.1874; (b) LV, maximum deviation 0.1437; (c) LNHPP, maximum deviation 0.0805



(a)



(b)



(c)



Prequential likelihood ratio comparison of JM, LV and LNHP models operating on the data of Table 1

| JM and LV models |         | LNHP and LV models |         | LNHP and JM models |         |
|------------------|---------|--------------------|---------|--------------------|---------|
| n                | $PLR_n$ | n                  | $PLR_n$ | n                  | $PLR_n$ |
| 10               | 1.19    | 10                 | 1.16    | 10                 | 0.975   |
| 20               | 0.318   | 20                 | 0.593   | 20                 | 1.86    |
| 30               | 0.252   | 30                 | 0.759   | 30                 | 3.01    |
| 40               | 0.096   | 40                 | 0.502   | 40                 | 5.23    |
| 50               | 0.745   | 50                 | 1.83    | 50                 | 2.46    |
| 60               | 6.50    | 60                 | 7.56    | 60                 | 1.16    |
| 70               | 0.088   | 70                 | 5.75    | 70                 | 65.34   |
| 80               | 0.00177 | 80                 | 1.24    | 80                 | 700.56  |
| 90               | 0.00008 | 90                 | 0.66    | 90                 | 8118.08 |
| 100              | 0.00119 | 100                | 30.85   | 100                | 25924.3 |



# Optimistic/pessimistic

- If u-plot is everywhere above the line of unit slope the predictions are '*too optimistic*'; if they are everywhere below the line, '*too pessimistic*'.
- Here JM is far too optimistic, which confirms suspicion from median plot; LV is slightly *too pessimistic*.
- This poor u-plot performance for JM probably explains the poor PLR performance versus LM, LV.
- The median plot, and PLR, seem to show that there is little to choose between the three in the early stages, but u-plot aggregates over the *whole sequence of predictions*.

# Summary

- These models seem to perform almost as well as the best parametric models for most data sets.
- They seem robust: whilst the performance of the parametric models varies considerably from one data set to another, these seem fairly consistent.
- In most cases the best performing model is usually a parametric one.
- Since we can select the best model via analysis of predictive accuracy, it is still best to be eclectic in model choice and 'let the data decide'.
- Some of these models are very computationally intensive (but who cares?!).

**'Prediction is very difficult, especially of the future'**  
(Niels Bohr)

# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
- 5. Experiences in Software Reliability**
6. Software Reliability Engineering Practice
7. Lessons Learnt
8. Background Literature

# 5. Experiences in Software Reliability

- Limits in software reliability
- Reliability & Availability Guidelines
- A Case Study from the Nuclear Industry
- How might we gain confidence in ultrahigh reliability?
- The law of diminishing returns: ‘Heroic debugging’ does not work
- Adams effect
- Exercises

# Limits to reliability measurement (1)

- The dependence upon computers in safety-critical applications is accelerating:
  - A330-340 flight control:  $10^{-9}$  failures per hour stated requirement
  - Sizewell B reactor protection:  $10^{-4}$  prob of failure on demand
  - Air traffic control: 3 seconds per annum
  - Chemical plant: risks comparable to nuclear
  - Robotics (e.g. surgical assistance): surprisingly modest requirements
  - Automobiles (engine management, ABS, 4WS)
  - Railway signalling and control (TGV):  $10^{-12}$  prob of fail per hour

# Limits to reliability measurement (2)

- Can we build these to the reliability levels needed?
- *How do we convince ourselves* that the reliability targets *have been achieved* when software plays a critical role?
- What are the limits to the levels of reliability we can measure? Are these just limits to the current measurement techniques, or are they intrinsic?



# How much confidence should we place in a system that has not failed at all?

(1)

- Let the random variable  $T$  represents the time to next failure, and let us assume that this program has been on test for a period  $t_0$ , during no failures have occurred.
- $T$  is exponential with rate  $\lambda$
- And mean  $\theta = \lambda^{-1}$
- Assume, in general,  $x$  failures of the program during the period of testing  $t_0$ .

# How much confidence should we place in a system that has not failed at all?

(2)

- Bayes theorem states

$$p(\lambda \mid \text{data}) \propto p(\lambda) p(\text{data} \mid \lambda),$$

where the distribution  $p(\lambda)$  represents the prior belief on occurrence of the failures,  $\lambda$ , and  $p(\lambda \mid \text{data})$  represent the posterior belief after seeing the data.

- Assuming the sequence of failures as a Poisson process, then  $p(\text{data} \mid \lambda)$  is proportional to  $\lambda^x \cdot \exp(-\lambda t_0)$ .

# How much confidence should we place in a system that has not failed at all?

(3)

- The form of  $p(\lambda)$  (**a conjugate family of distributions, like Gamma**) permits some homogeneity.
- The prior belief - Gamma ( $a, b$ ), for some suitable choice of **a** and **b**.
- $p(\lambda \mid x, t_0)$  is represented by Gamma ( $a+x, b+t_0$ ).
- Under conjugacy both posterior distribution and prior will be a member of the same family: for example the expected value,  $E(\lambda)$ , changes from  $a/b$  to  $(a+x)/(b+t_0)$ , so that *observing a small number of failures,  $x$ , in a long time  $t_0$ , will cause the posterior expected value to be smaller than the prior.*

## How much confidence should we place in a system that has not failed at all?

3)

We can now make probability statements about  $\lambda$ , and about  $T$  itself:

$$\begin{aligned} p(t \mid x, t_0) &= \int p(t \mid \lambda) p(\lambda \mid x, t_0) d\lambda \\ &= \frac{(a+x)(b+t_0)^{a+x}}{(b+t_0+t)^{a+x+1}} \end{aligned}$$

It follows that the reliability function is

$$\begin{aligned} R(t \mid x, t_0) &= P(T > t \mid x, t_0) \\ &= \left( \frac{b+t_0}{b+t_0+t} \right)^{a+x} \end{aligned}$$

and in our case, when  $x = 0$ ,

$$= \left( \frac{b+t_0}{b+t_0+t} \right)^a$$

# How much confidence should we place in a system that has not failed at all?

Ignoring prior distribution ?!

(1)

- Modelling “*total ignorance*” is difficult.
- To represent *initial ignorance*, we should take *a* and *b* as small as possible.
- The posterior distribution is approximately Gamma ( $x, t_0$ ), with the approximation improving as  $a, b \rightarrow 0$ .
- We could informally think of Gamma ( $x, t_0$ ) as the posterior in which the data “*speak for themselves*”.
- When  $x=0$  the posterior distribution for the rate is proportional to  $\lambda^{-1}$ , and is thus *improper* (i.e., it yields a total probability mass greater than 1).
- Worse, *the predictive distribution for  $T$  is also improper*, and is thus useless for prediction.

## How much confidence should we place in a system that has not failed at all?

Ignoring prior distribution ?! (2)

Here is a possible way forward. Choose the improper prior

$$p(\lambda) \equiv 1$$

giving the posterior

$$p(\lambda \mid 0, t_0) = t_0 \exp(-\lambda t_0)$$

which is a proper distribution. More importantly, the predictive distribution for  $T$  is also proper:

$$\begin{aligned} p(t \mid 0, t_0) &= \int p(t \mid \lambda) p(\lambda \mid 0, t_0) d\lambda \\ &= t_0 / (t_0 + t)^2 \end{aligned}$$

The reliability function is

$$\begin{aligned} R(t \mid 0, t_0) &\equiv P(T > t \mid 0, t_0) \\ &= t_0 / (t + t_0) \end{aligned}$$

and in particular,  $R(t_0 \mid 0, t_0) = 1/2$ : i.e. we have a 50:50 chance of seeing a further period of failure-free working as has already been observed.



How much confidence should we place in a system that has not failed at all?

What prior belief is needed to arrive at a posterior belief in ultra-high reliability?  
Is such belief reasonable? (1)

- The conclusion here is that *observing a long period of failure-free working does not **in itself** allow us to conclude that a system is ultra-reliable*. It must be admitted that the prior distribution here is rather unrealistic.
- Let us consider the case where the observer has genuine prior beliefs about  $\lambda$ .
- **Example:** the reliability requirement is that the median time to failure is  $10^6$  hours, and the trust has shown failure-free working for  $10^3$  hours, **what prior belief would the observer have needed in order to conclude that the requirement had been met?**

**How much confidence should we place in a system that has not failed at all?**

**What prior belief is needed to arrive at a posterior belief in ultra-high reliability?**

**Is such belief reasonable? (2)**

- From above, (a, b) must satisfy

$$\frac{1}{2} = \left( \frac{b + 10^3}{b + 10^3 + 10^6} \right)^a$$

which implies, since  $b > 0$ , that  $a > 0.1003288$ .

- It is instructive to examine what is implied by prior beliefs in this solution set.

How much confidence should we place in a system that has not failed at all?

What prior belief is needed to arrive at a posterior belief in ultra-high reliability?

Is such belief reasonable? (3)

- Consider, for example,  $a=0.11$ ,  $b=837.2$ .

- *Is this a “reasonable” prior belief? Not, since the prior probability that  $T>10^6$  is 0.458.*

- The observer must believe *a priori* that there is almost 50:50 chance by surviving for  $10^6$  hours.

- If  $a = 0.50$ ,  $b=332333$ , the prior  $P(T>10^6)$  is 0.499. As  $a$  increases this problem becomes worse.

To believe that “this is a  $10^6$  system” after seeing only  $10^3$  hours of failure-free working, we must *initially* believe it was a  $10^6$  system.

To end up with a very high confidence in a system, *when we can see only a modest amount of testing, we must bring to the problem the relevant degree of belief.*

# Orders of magnitude less than ...

You can argue with the *details* of all this, but I think you are struck with the ball-park order-of-magnitude representing by this argument:

- *For the amounts of testing that are practically feasible, the confidence to be gained solely from such information is **orders of magnitude** less than is represented by, for example,  $10^{-9}$  failures/hr.*
- **Are there other sources of information, in addition to testing, that could allow us to gain higher confidence?** (e.g. by allowing us to *justifiably* have strong prior beliefs)

# Reliability Guidelines

| Typical ROCOF<br>(Failures/Hr) | Time Between<br>Failures |
|--------------------------------|--------------------------|
| $10^{-9}h^{-1}$                | <i>114,000 years</i>     |
| $10^{-6}h^{-1}$                | <i>114 years</i>         |
| $10^{-3}h^{-1}$                | <i>6 weeks</i>           |
| $10^{-2}h^{-1}$                | <i>100 hours</i>         |
| $10^{-1}h^{-1}$                | <i>10 hours</i>          |

# Use Availability Guidelines

| <b>Acceptable Down Time</b>            | <b>Availability</b>      |
|--|--------------------------|
| <i>5 minutes/year</i>                  | <i>5 nines (0.99999)</i> |
| <i>5 minutes/month or 1 hour/year</i>  | <i>4 nines (0.9999)</i>  |
| <i>10 minutes/week or 1 shift/year</i> | <i>3 nines (0.999)</i>   |



# Software Reliability in Safety Critical Applications: A Case Study from the Nuclear Industry (1)

- Software is being widely used in various *safety critical industries such as automobile, medical, petrochemical, nuclear, railways, etc.*
- The increase in software-based systems for safety functions requires systematic evaluation of software reliability. *Software reliability estimation is still an unresolved issue and existing approaches have limitations and assumptions that are not acceptable for safety applications.*

# Software Reliability in Safety Critical Applications: A Case Study from the Nuclear Industry (2)

- Existing reliability estimation techniques *require a sufficient and accurate history of software failures, which is not available for new software products.* A novel idea uses *mutation testing and software verification.* The approach has been demonstrated through a case study from the nuclear industry (specifically, the core temperature monitoring system of a nuclear reactor).

# It need to ...

- Firstly it needs to be emphasised that we do need to express our dependability requirements in the language of probability.
  - The sources of uncertainty we have met earlier are still present:
    - Operational environment
    - Incomplete knowledge of possible behaviour
  - Informally we need to have sufficient confidence that the system will fail sufficiently infrequently (or, for a one-shot system, with sufficiently low probability, etc).

# How might we gain confidence in ultrahigh reliability?

- Direct observation of operational behaviour of the system (e.g. in test or simulation) is not going to give assurance of ultra-high reliability:
  - The problem of ‘representativeness’ of input cases
  - The law of diminishing returns ...
- Aids to be used to obtain confidence in software designs:
  - Past experience with similar products, or products of the same process
  - Structural reliability modelling
  - Proofs and formal methods
  - Combination of different kinds of evidence
  - Validation by stepwise improvement of a product

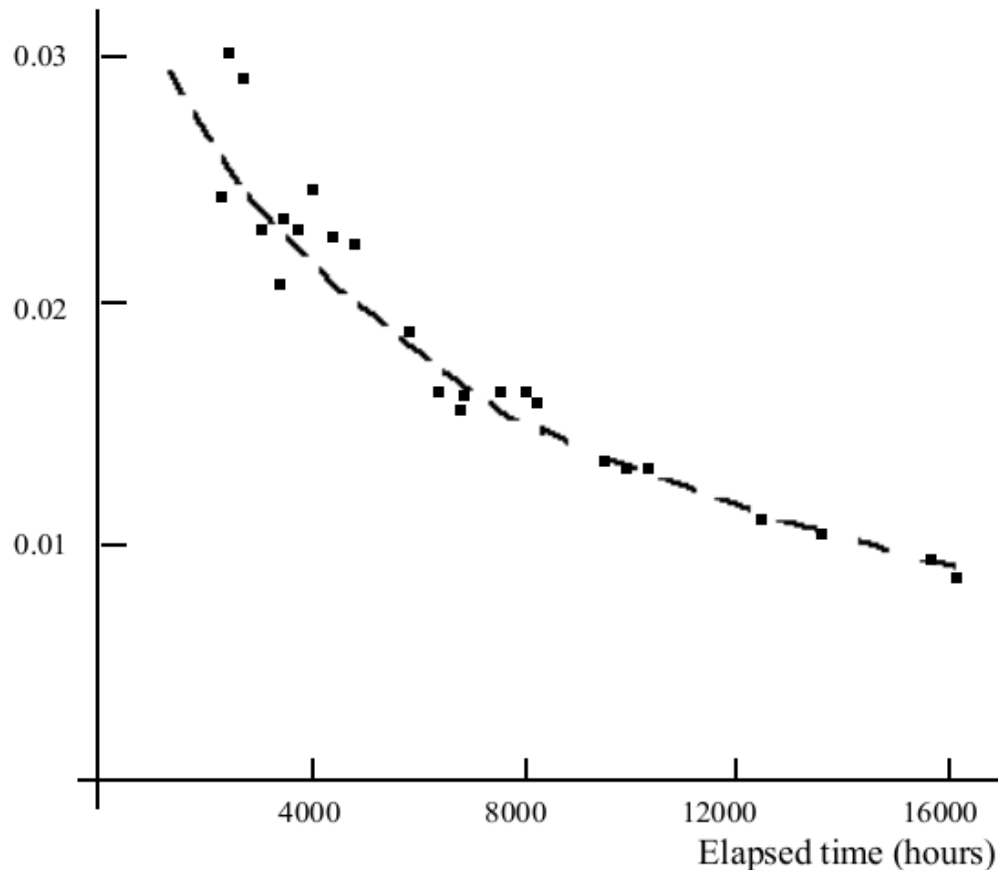
# ‘Heroic debugging’ does not work

| sample size, $i$ | elapsed time, $t_i$ | achieved mttf, $m_i$ | $t_i/m_i$ |
|------------------|---------------------|----------------------|-----------|
| 40               | 6380                | 288.8                | 22.1      |
| 50               | 10089               | 375.0                | 26.9      |
| 60               | 12560               | 392.5                | 32.0      |
| 70               | 16186               | 437.5                | 37.0      |
| 80               | 20567               | 490.4                | 41.9      |
| 90               | 29361               | 617.3                | 47.7      |
| 100              | 42015               | 776.3                | 54.1      |
| 110              | 49416               | 841.6                | 58.7      |
| 120              | 56485               | 896.4                | 63.0      |
| 130              | 74364               | 1054.1               | 70.1      |

**Table 1** An illustration of the law of diminishing returns in heroic debugging. Here the total execution time (seconds) required to reach a particular mean time to failure is compared with the mean itself.

Later improvements in the MTTF require proportionally longer testing.

# Adams effect



**Figure 2** Estimates of the rate of occurrence of failures for a system experiencing failures due to software faults and hardware design faults. The broken line here is fitted by eye. Once again, the rate is not recomputed at each data point here: the plotted points represent only occasional recomputation of the rate during the observation of several hundred failures.

- Field data on many copies of a system undergoing failures as a result of both software and hardware design faults; points are 'current rate' estimates from LV, curve fitted by eye.
- Again a strong law of diminishing returns
- To get very low rate will take extremely long time even if achievable (what is asymptote?)
- Adams effect: rates of faults differ by order of magnitude; system eventually is depleted of the 'large' ones; unreliability then comes from many small faults; fixes have little effect upon unreliability.



# Exercises( Course Work)

## Questions 1

1. Imagine that your job is to evaluate the reliability of the software in a company. This means setting up an in-house testing service, whereby test cases would be generated in such a way as to provide input to the reliability growth models described in the lectures. How would you propose to go about your task when the software in question is

- a) a word-processing system
- b) a air-traffic control system
- c) a car engine management system
- d) a controller for a domestic washing machine

In each case discuss if you could justify the test procedure you and advocating as the only testing to be carried out on the particular product, bearing in mind that there is a fixed budget from which all testing must funded.

# Reliability Course Solutions 1

There are three questions to consider for each example:

1. What is a failure , and what classification of different failure severity levels is appropriate?
2. How should execution time be defined and measured?
3. How can the true operational profile be approximated for the purpose of random testing?

These suggest something like the following proposals for the four examples.( Although credit may be given other well reasoned suggestions.)

**Q: a) A word-processing system**

**A: a)** Some market research would help to establish frequency of use of various functions. For example, the proportion of users who would require mathematical notation. This research could be used as the basis for a representative selection of documents for input and editing by several testers possessing varying level of typing skill. Alternative execution time measures: hands on time ; number of words typed; processor time consumed by word processing software. The latter two measures presume the availability of some automated data collection function. The following and categories of failure , roughly in decreasing order of seriousness; loss of data; corruption ( misprinting of document ); user-interface difficulties. No safety critical failure . It is likely that reliability requirements could be assured by such random testing alone.



**Q: b) A air-traffic control system**

**A: b)** Far higher potential cost of failures for this safety critical system , allows aircraft to collide; fails to maintain track of aircraft; makes unnecessary demand on memory or attention of human operators. Execution time measure: processor time. Operational profile: examine pattern of traffic on existing ATC systems, model and simulate with real operators participating at screens in lifelike simulation. ( This will be extremely costly to do well. ) Random testing insufficient alone to assure ultra-high reliability . Deliberately concentrate some testing effort on critical scenarios. Possibly some static analysis and formal verification of code.(There is some controversy as to whether even all this is adequate, particularly if system contains much design novelty over preceding systems.)

**Q: c) A car engine management system**

**A: c) Safety critical failures: engine cut-out at speed; sudden, unrequested acceleration. Other failures in order of decreasing cost: overheats and damages engine; degraded performance of fuel – efficiency.**

**Execution time: total engine running time.**

**Operational profile: examine driving patterns; Test with selected drivers on simulation ring or road.**

**Require stress test in addition to random testing.**



Q. d) A controller for a domestic washing machine

A: d) Possible safety critical software failure : door allowed to open while drum rotating ( and with hot wash? )- unless this is prevented by mechanical interlock, as is likely to be the case. Costly failures : floods floor; damages delicate fabrics; inadequate wash. Execution time: accumulated wash time. Operational profile: Market research then parallel testing of many machines using different loads and programs. Perhaps exhaustive testing of software possible, depending on complexity ? Random testing alone sufficient.



Q• d) A controller for a domestic washing machine

A: d) Possible safety critical software failure : door allowed to open while drum rotating ( and with hot wash? )- unless this is prevented by mechanical interlock, as is likely to be the case. Costly failures : floods floor; damages delicate fabrics; inadequate wash. Execution time: accumulated wash time. Operational profile: Market research then parallel testing of many machines using different loads and programs. Perhaps exhaustive testing of software possible, depending on complexity ? Random testing alone sufficient.

# APPENDIX A

Table 1. Data set 1 (162 inter-failure times)

(read from left to right)

|       |       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 60.   | 30.   | 607.  | 40.   | 28.   | 53.   | 4.    | 16.   | 94.   | 15.   |
| 5.    | 90.   | 77.   | 68.   | 15.   | 160.  | 1.    | 104.  | 16.   | 9.    |
| 22.   | 14.   | 82.   | 6.    | 79.   | 43.   | 12.   | 87.   | 138.  | 108.  |
| 30.   | 295.  | 113.  | 36.   | 50.   | 81.   | 89.   | 157.  | 14.   | 166.  |
| 36.   | 5.    | 69.   | 50.   | 164.  | 392.  | 16.   | 154.  | 176.  | 247.  |
| 304.  | 5.    | 2.    | 135.  | 233.  | 156.  | 295.  | 152.  | 341.  | 103.  |
| 16.   | 83.   | 7.    | 221.  | 4.    | 38.   | 11.   | 17.   | 107.  | 59.   |
| 127.  | 20.   | 1.    | 21.   | 112.  | 23.   | 193.  | 1104. | 103.  | 318.  |
| 114.  | 1553. | 306.  | 245.  | 107.  | 2.    | 327.  | 696.  | 5.    | 63.   |
| 1054. | 495.  | 128.  | 482.  | 116.  | 35.   | 310.  | 110.  | 98.   | 60.   |
| 177.  | 65.   | 231.  | 62.   | 158.  | 1622. | 386.  | 70.   | 151.  | 809.  |
| 1710. | 745.  | 350.  | 592.  | 3569. | 1967. | 772.  | 3337. | 620.  | 3044. |
| 6.    | 1923. | 17.   | 71.   | 34.   | 1176. | 1040. | 38.   | 78.   | 1798. |
| 205.  | 2095. | 788.  | 1.    | 3812. | 726.  | 1452. | 5173. | 1957. | 3097. |
| 25.   | 1048. | 78.   | 33.   | 6725. | 1366. | 2859. | 556.  | 5028. | 537.  |
| 63.   | 113.  | 1236. | 1406. | 1580. | 3546. | 8575. | 1893. | 198.  | 3326. |
| 6372. | 124.  |       |       |       |       |       |       |       |       |

Fig. 1: Plot of cumulative number of failures against total elapsed time for data set 1.

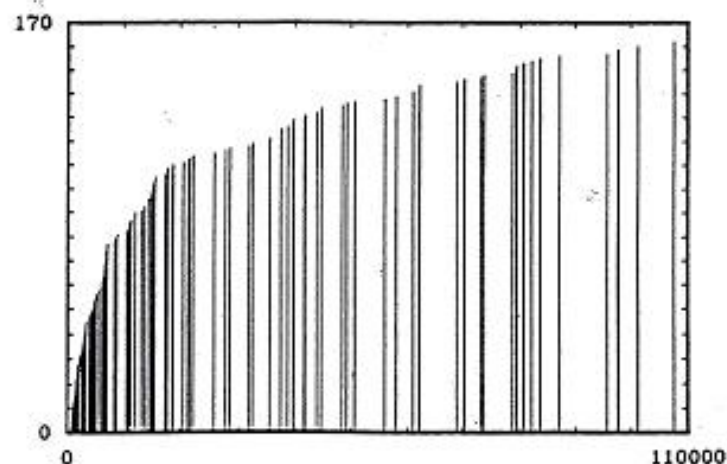


Fig. 2: Median predictions of  $T_{66}, \dots, T_{162}$ , from the raw models for data set 1

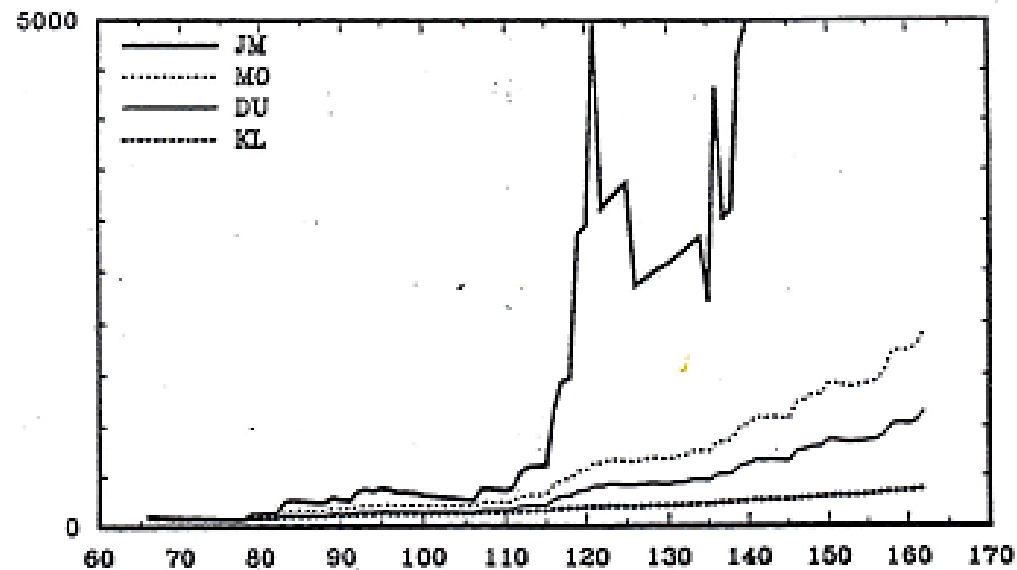


Fig. 3:  $u$ -plots from the raw models for data set 1

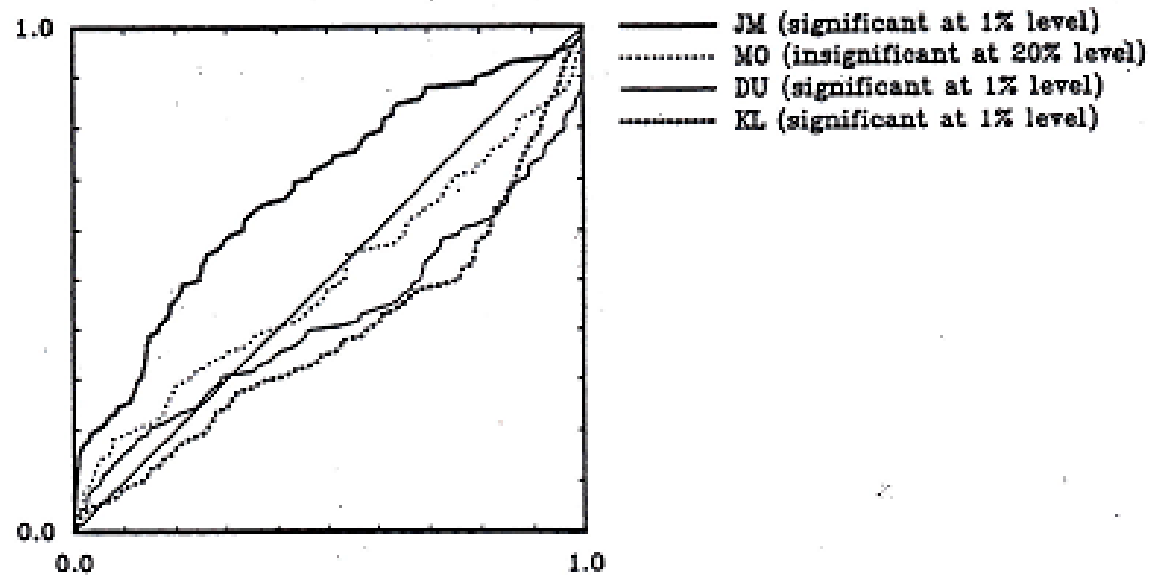




Fig. 4:  $\log(PLR)$ -plots for the raw models versus the  $DU$  model for data set 1

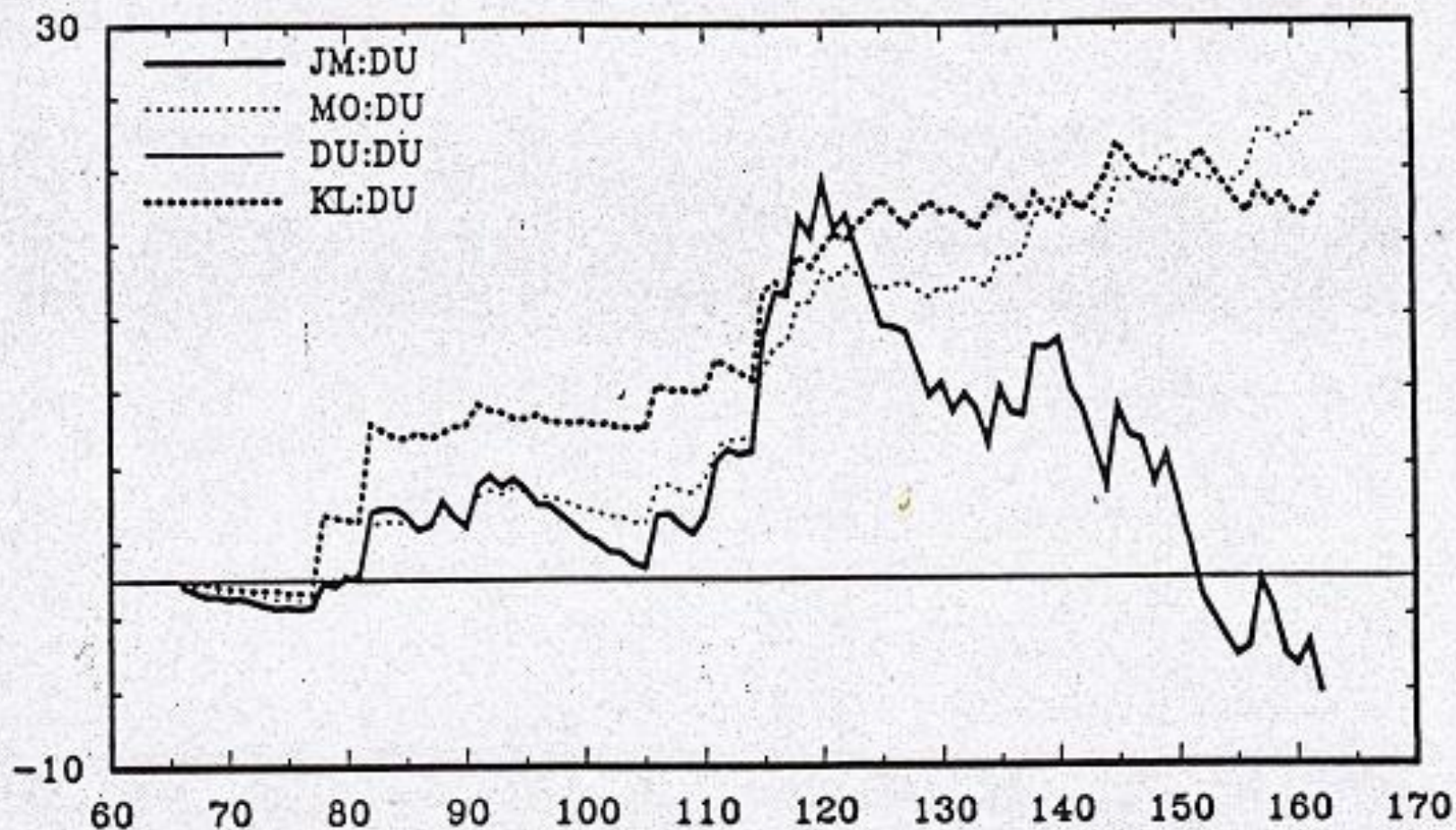


Table 1. Data set 1 (129 inter-failure times)

(read left to right)

|      |       |       |       |       |       |       |       |       |       |
|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 760. | 758.  | 303.  | 6.    | 22.   | 14.   | 42.   | 4.    | 84.   | 15.   |
| 221. | 14.   | 15.   | 41.   | 1.    | 153.  | 409.  | 54.   | 24.   | 44.   |
| 180. | 397.  | 19.   | 145.  | 36.   | 54.   | 1337. | 163.  | 8.    | 1.    |
| 17.  | 16.   | 87.   | 19.   | 29.   | 0.5   | 300.  | 360.  | 10.   | 11.   |
| 100. | 252.  | 460.  | 179.  | 3.    | 24.   | 253.  | 163.  | 54.   | 137.  |
| 328. | 3.    | 9.    | 12.   | 18.   | 9.    | 75.   | 15.   | 366.  | 428.  |
| 212. | 115.  | 264.  | 269.  | 276.  | 1.    | 999.  | 30.   | 495.  | 472.  |
| 344. | 550.  | 131.  | 47.   | 92.   | 863.  | 991.  | 35.   | 9549. | 249.  |
| 607. | 83.   | 614.  | 352.  | 673.  | 4179. | 111.  | 75.   | 407.  | 288.  |
| 894. | 1314. | 845.  | 55.   | 409.  | 36.   | 15.   | 1960. | 60.   | 19.   |
| 20.  | 79.   | 24.   | 1737. | 7984. | 10.   | 20.   | 338.  | 250.  | 1682. |
| 212. | 287.  | 56.   | 4973. | 3500. | 59.   | 98.   | 2439. | 1812. | 6203. |
| 385. | 3500. | 4892. | 687.  | 62.   | 2796. | 3268. | 3845. | 76.   |       |

Fig. 1: Plot of cumulative failures against total time for the data in table 1.

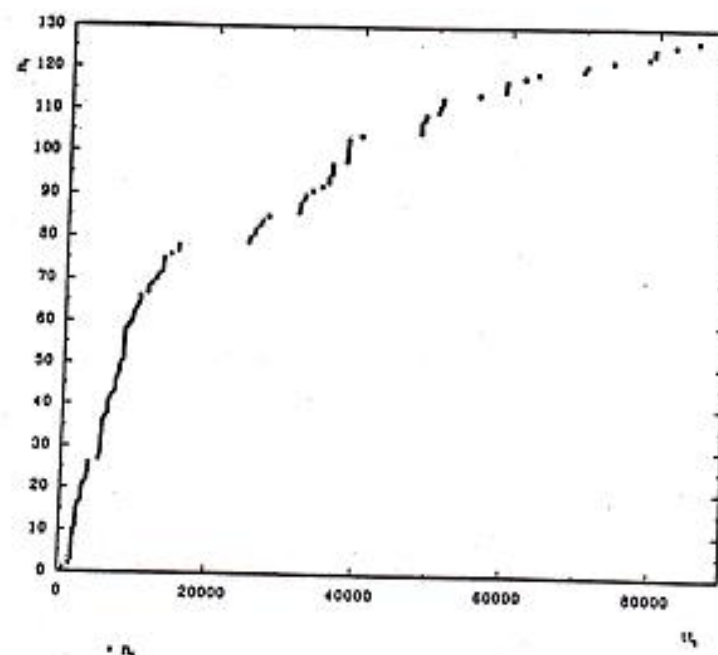


Fig. 2: Median predictions of  $T_{66}, \dots, T_{129}$ , from the raw models for data set 1

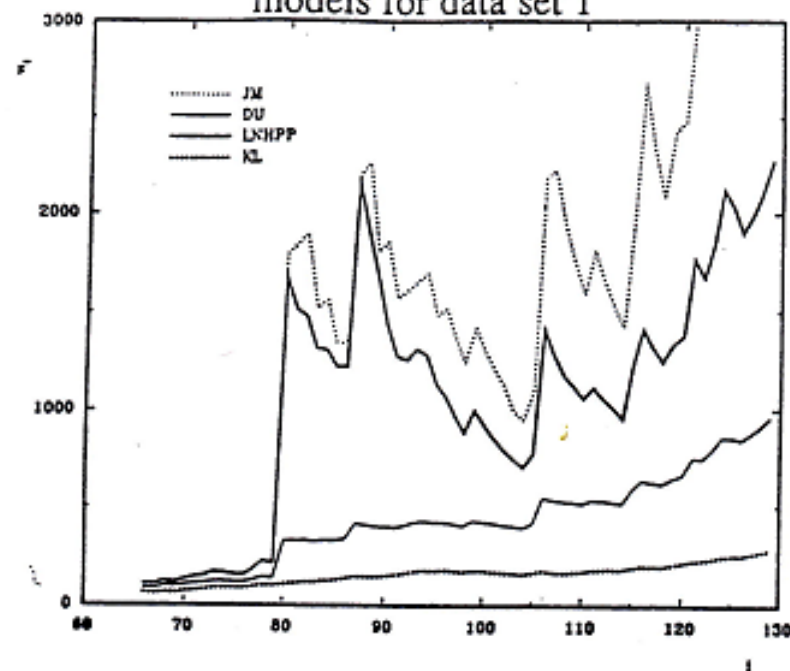


Fig. 3:  $u$ -plots from the raw models for data set 1

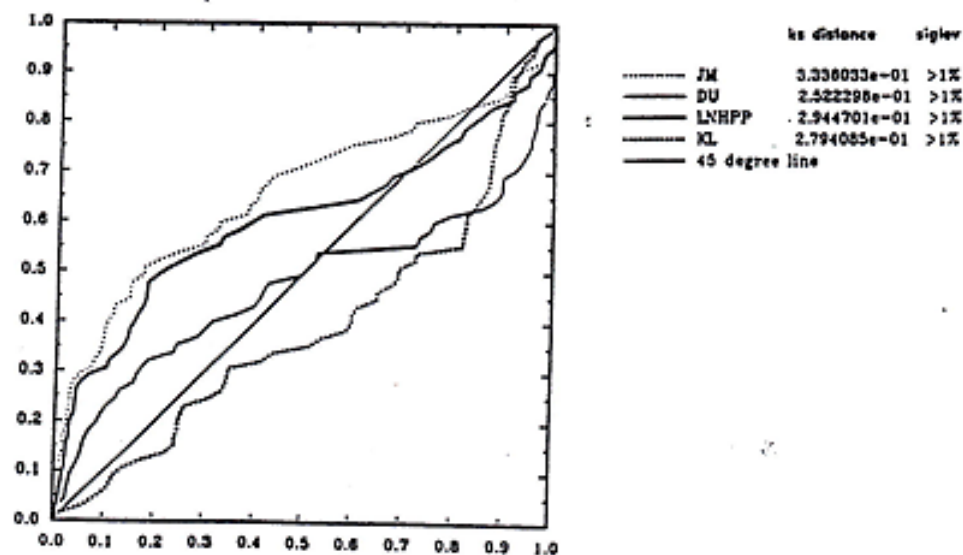




Fig. 4:  $\log(PLR)$ -plots for the raw models versus the  $DU$  model for data set 1

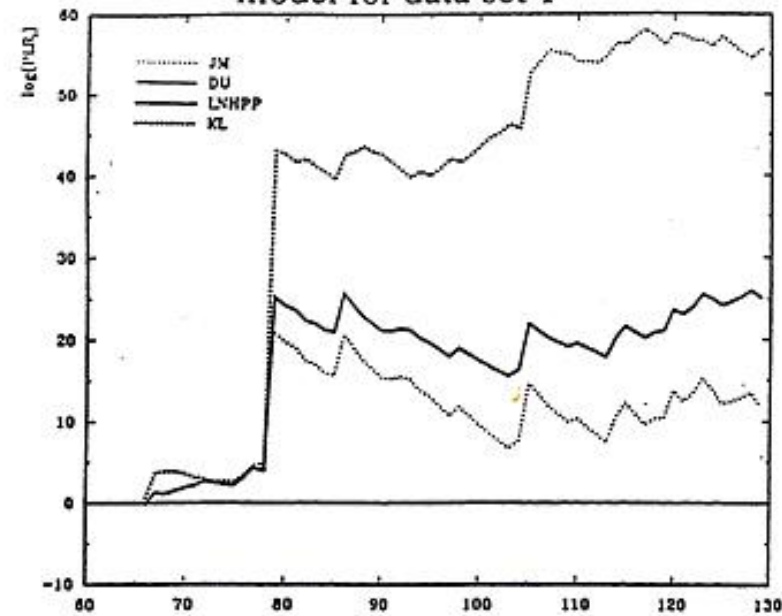


Fig. 5: Median predictions of  $T_{66}, \dots, T_{129}$ , from the recalibrated models for data set 1

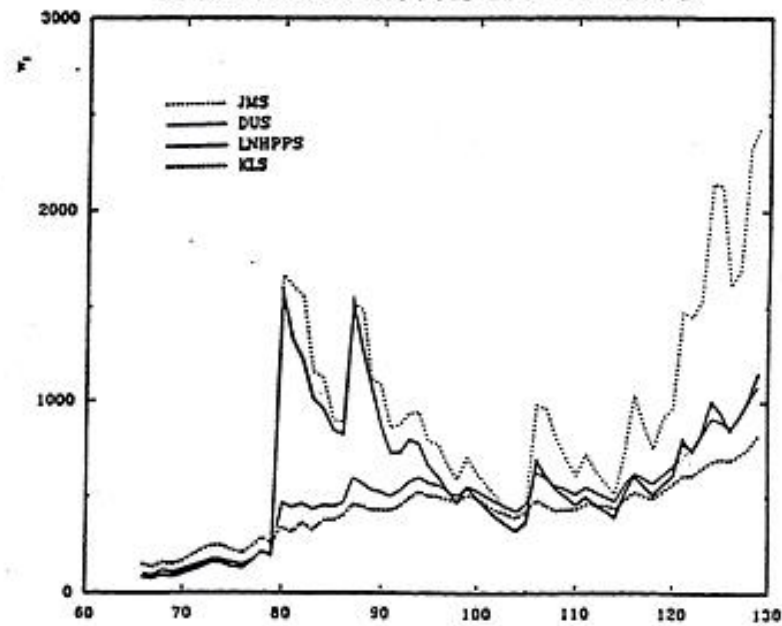


Fig. 6:  $u$ -plots from the recalibrated models for data set 1

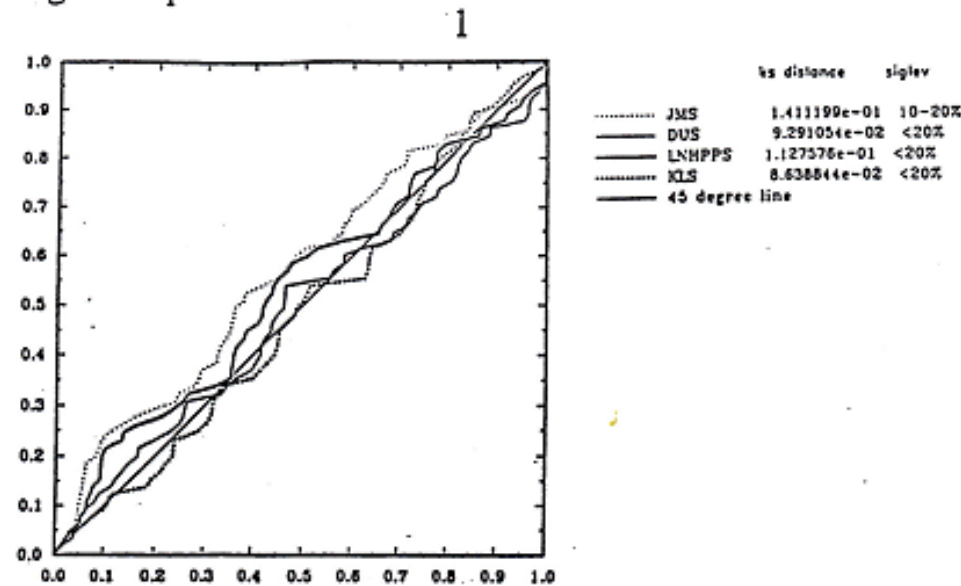


Fig. 7:  $\log(PLR)$ -plots for the recalibrated models versus the *DUS* model for data set 1

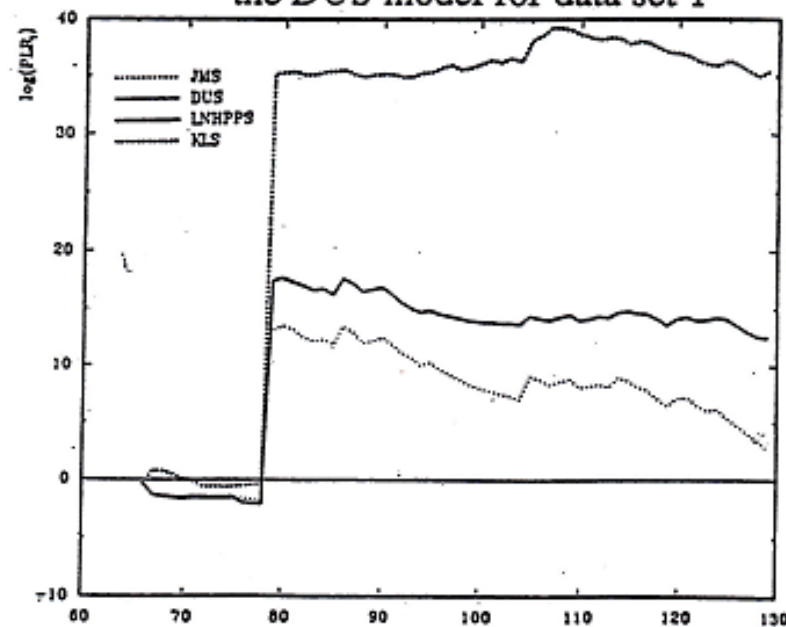
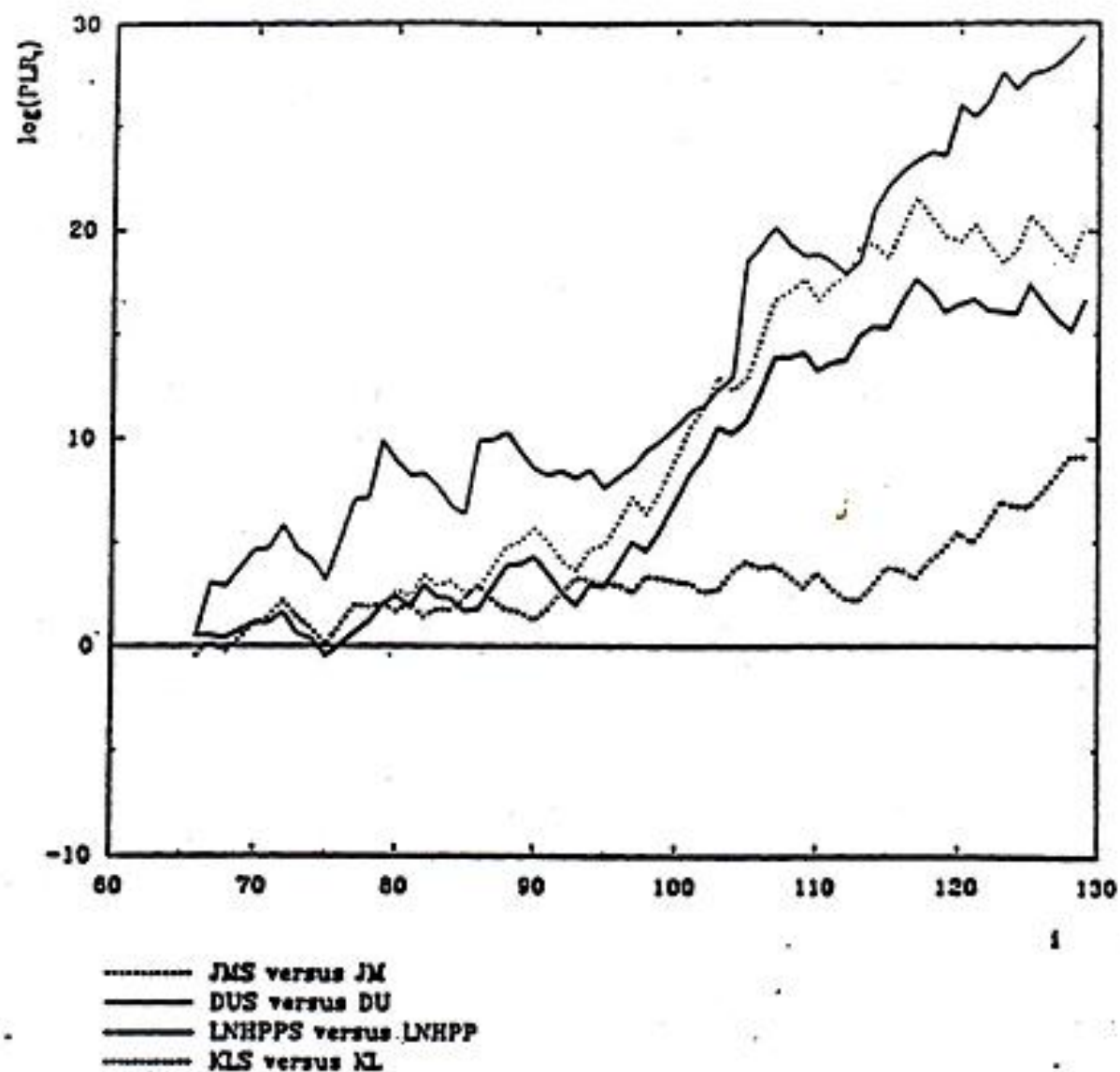


Fig. 8:  $\log(PLR)$ -plots for the recalibrated models versus the raw models for data set 1



# Table 1. Data set - SYS1 (136 inter-failure times)

(read left to right)

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| 3.    | 30.   | 113.  | 81.   | 115.  |
| 9.    | 2.    | 91.   | 112.  | 15.   |
| 138.  | 50.   | 77.   | 24.   | 108.  |
| 88.   | 670.  | 120.  | 26.   | 114.  |
| 325.  | 55.   | 242.  | 68.   | 422.  |
| 180.  | 10.   | 1146. | 600.  | 15.   |
| 36.   | 4.    | 0.    | 8.    | 227.  |
| 65.   | 176.  | 58.   | 457.  | 300.  |
| 97.   | 263.  | 452.  | 255.  | 197.  |
| 193.  | 6.    | 79.   | 816.  | 1351. |
| 148.  | 21.   | 233.  | 134.  | 357.  |
| 193.  | 236.  | 31.   | 369.  | 748.  |
| 0.    | 232.  | 330.  | 365.  | 1222. |
| 543.  | 10.   | 16.   | 529.  | 379.  |
| 44.   | 129.  | 810.  | 290.  | 300.  |
| 529.  | 281.  | 160.  | 828.  | 1011. |
| 445.  | 296.  | 1755. | 1064. | 1783. |
| 860.  | 983.  | 707.  | 33.   | 868.  |
| 724.  | 2323. | 2930. | 1461. | 843.  |
| 12.   | 261.  | 1800. | 865.  | 1435. |
| 30.   | 143.  | 108.  | 0.    | 3110. |
| 1247. | 943.  | 700.  | 875.  | 245.  |
| 729.  | 1897. | 447.  | 386.  | 446.  |
| 122.  | 990.  | 948.  | 1082. | 22.   |
| 75.   | 482.  | 5509. | 100.  | 10.   |
| 1071. | 371.  | 790.  | 6150. | 3321. |
| 1045. | 648.  | 5485. | 1160. | 1864. |
| 4116. |       |       |       |       |



## Question 2

Using the data of **Table 1** and one of the more accurate (see **Fig.2e** and **Fig.3e**) sets of predictions from the median plots of **Fig 1e** , obtain an approximate plot of achieved median against total elapsed test time. Comment on the shape of this. You have been asked to continue testing until you are confident that the median time to failure is 1000 hours (remember the raw data is in seconds); comment on the feasibility of this.

### Notes on figures

- i) "raw model" is used to make the distinction between the "raw" reliability models initially applied and the recalibrated version of these models, later applied.
- ii) The abbreviated model name (eg. DU) followed by and S (eg. DUS ) is used to refer to the recalibrated version of the model in question.
- iii) In **figures 3e, 6e and 9e**, the DU model ( the DUS model in the case of **Fig 9e** coincides with the zero axis (since the  $\log(\text{PLR})$  of a model versus itself will always be zero).
- iv) In **Fig 1e to 10e** it can be seen that some models tend to give similar predictions (eg. the LV and KL models will frequently give similar predictions). This is because the underlying assumptions for these models are similar. In the case of data set SYS1 the LNHPP model actually coincides with the MO model for the range of predictions shown and the L model also coincides with the MO and LNHPP models over most of the range shown.

### Question 3 and Reliability Coursework Solution 3

- Q: a) **Table 2e** (Data set TSW ) shows inter-failure time data ("hand-on" time measured in minutes) from software failures observed by a single user of a work station. From this data construct a plot of the cumulative number of failures against the elapsed time. Comment on the shape of this plot.
- A: a) This plot has the increasing, concave shape typical of the progressive removal of design faults. The failure rate ( slope of plot ) is decreasing. Also the decreasing rate at which the slope is decreasing indicates a "diminishing return".



Fig. 1e: Median predictions of  $T_{36}, \dots, T_{136}$ , from the raw models for data set SYS1

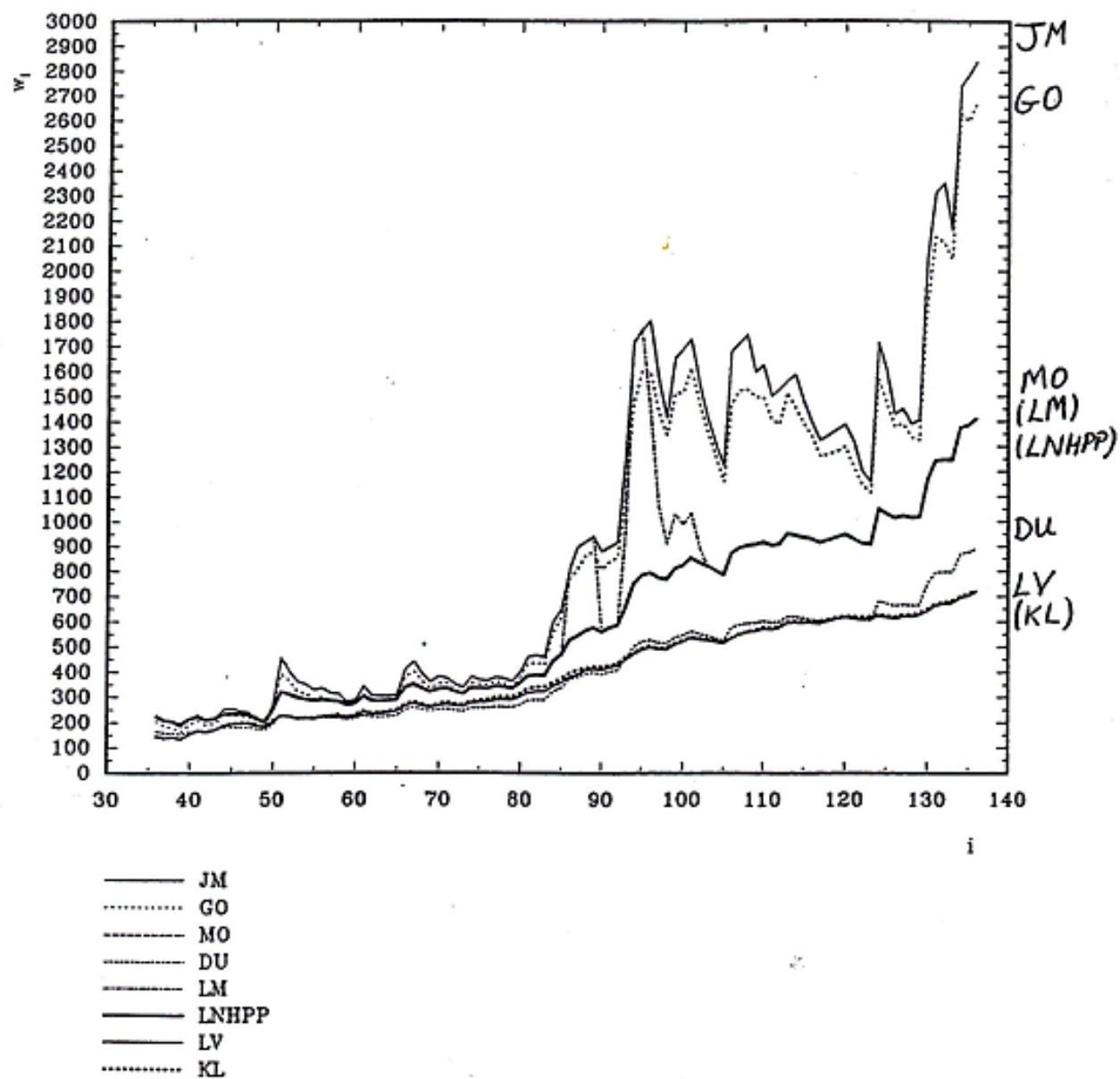
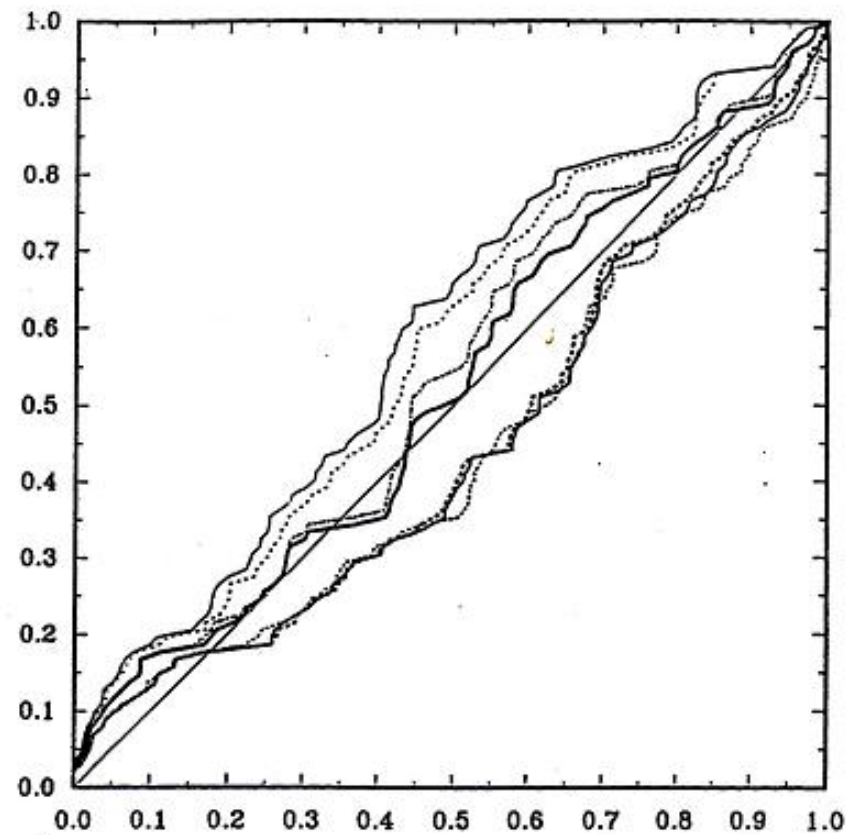
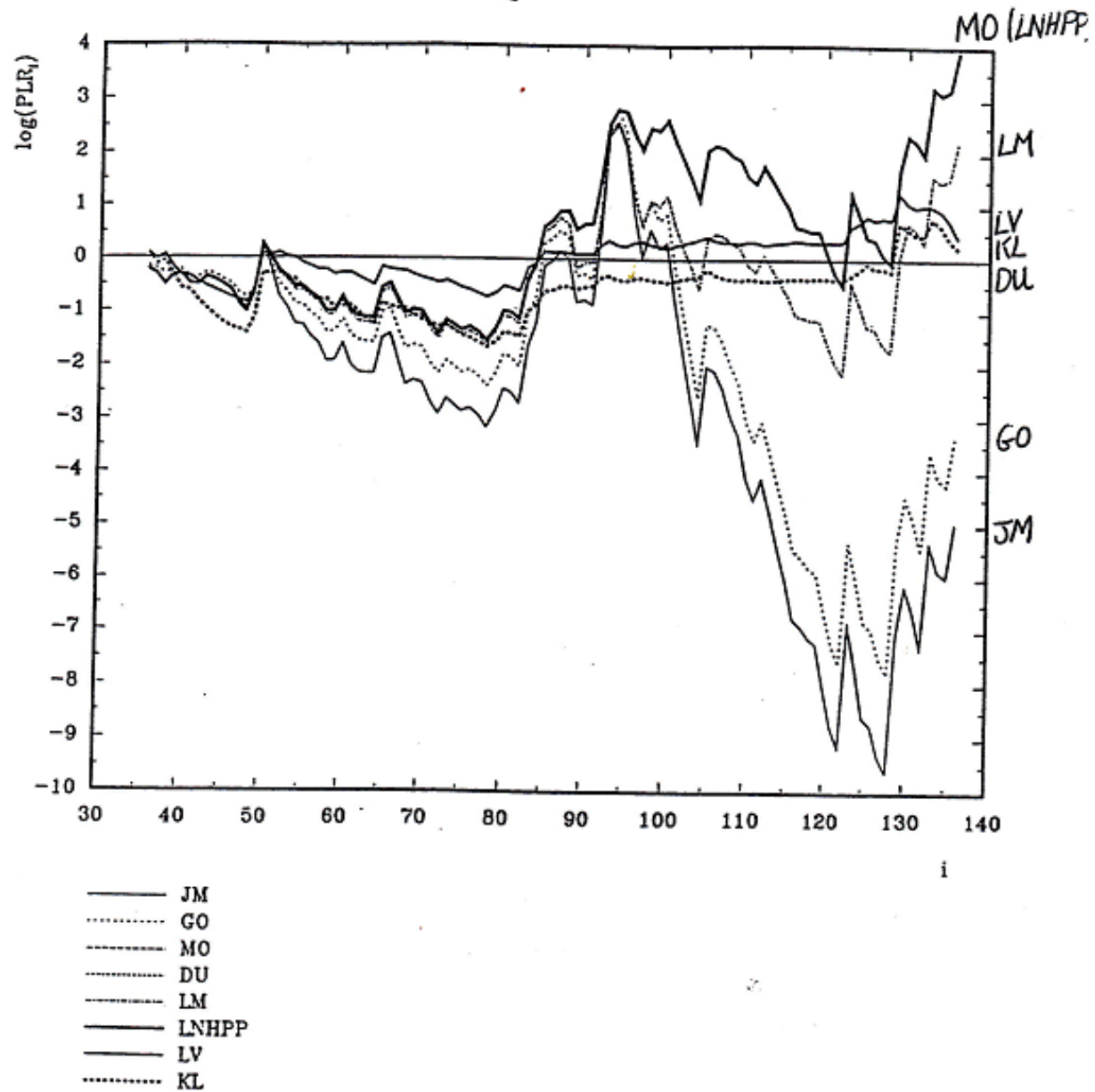


Fig. 2e:  $u$ -plots from the raw models for data set SYS1



|          | ks distance  | siglev |
|----------|--------------|--------|
| —— JM    | 1.811700e-01 | <1%    |
| ..... GO | 1.466002e-01 | 2-5%   |
| ----- MO | 7.886967e-02 | <20%   |
| ----- DU | 1.624068e-01 | <1%    |
| ----- LM | 1.027675e-01 | <20%   |
| —— LNHPP | 7.886767e-02 | <20%   |
| —— LV    | 1.479544e-01 | 2-5%   |
| ..... KL | 1.420824e-01 | 2-5%   |

Fig. 3e:  $\log(PLR)$ -plots for the raw models versus the  $DU$  model for data set SYS1





From the u-plot, the LNHPP and MO models perform best as predictors of time-to-next-failure (i.e plots closest to 45-line), as far as bias is concerned, with the LM performing almost equally well. The PLR plot yields a similar conclusion for overall performance of these predictors ( because the PLR plot vs. DU is closest to exhibiting a consistently increasing trend for these models). Therefore the median time-to-next-failure plot for LNHPP & MO is selected as the best available estimate of achieved median. (these two median plots are so close as to be indistinguishable.) The horizontal axis is now transformed in order to plot achieved median  $i$  against  $t_j$ , the cumulative elapsed time, rather than against  $i$ , the  $j=1$  cumulative number of failures used in the provided. The resulting plot can be fitted by eye as an increasing, approximately linear but perhaps slightly concave (i.e. decreasing slope ) function of elapsed time. The bottom left hand point of the fit is at just over 5,000 seconds elapsed time ( the time at which about 36 failures have been observed and prediction is started ) and about 200 seconds achieved median inter-failure time. At the other ( later ) end of the fit, the total elapsed time is approaching 90,000 seconds and the best achieved line fit is approximately .015. A rough conservative calculation shows that, at the same average rate of improvement in median, it will take approximately a further  $240,000,000 = (3,600,000 - 1,400) / .015$  seconds, i.e. about 7.6 years of continuous testing in order to achieve the required median of 3,600,000 seconds. If the plot is actually concave, the time will be even longer. This amount of testing is almost certainly infeasible in practice.

- Q: b) **Figure 4e** shows the median predictions which result when 8 raw reliability models are applied to this inter-failure time data. Briefly discuss what this plot tells us about these predictions. Discuss what the u-plot in **Fig 5e** tell us about the error in predictive accuracy in each of the models (i.e the bias in the predictions). By examining **Fig 4e and 5e** suggest which model (or models) may be giving the most accurate median predictions.
- A: b) The plots in **Fig 4e** confirm the improvement in reliability by the clear increasing trend in median time-to- next-failure prediction. However, the predictions from the different models are noticeable inconsistent. (Although, certain pair of models gives very similar predictions.) Clearly they cannot all be accurate, the predictions from the different models containing different amount of " noise ". ( Some models produce predictions which are less stable that others as each new data point is incorporated). The u-plot provides an indication of predictor bias. All of these models appear to be producing biased predictions since the u-plots deviate significantly from the 45-line in all eight cases. From those u-plot which deviate consistently below the 45-line, we can conclude that the corresponding model is over pessimistic, tending to underestimate the time- to-next-failure ( KL,LV). Conversely a plot above the line indicates excessive optimism ( JM, GO,LM,LNHPP). The bias, or otherwise , specifically of the median predictions is indicated by the vertical distance of the u- plot from the 45-line at 0.5 on the horizontal axis, since the vertical value of the u-plot here is the proportion of  $u_i$  which were less than  $1/2$ , i.e. the proportion of  $t_i$  which were less that the median of the predictive distribution function,  $F_i(t)$ . On this criterion the MO and DU predictors are best. The fact that, in **Figure 4e**, these two models produce medians lying between those of the other models appears to confirm the suggestion that these two models produce the most accurate medians.



Fig. 4e: Median predictions of  $T_{66}, \dots, T_{129}$ , from the raw models for data set TSW

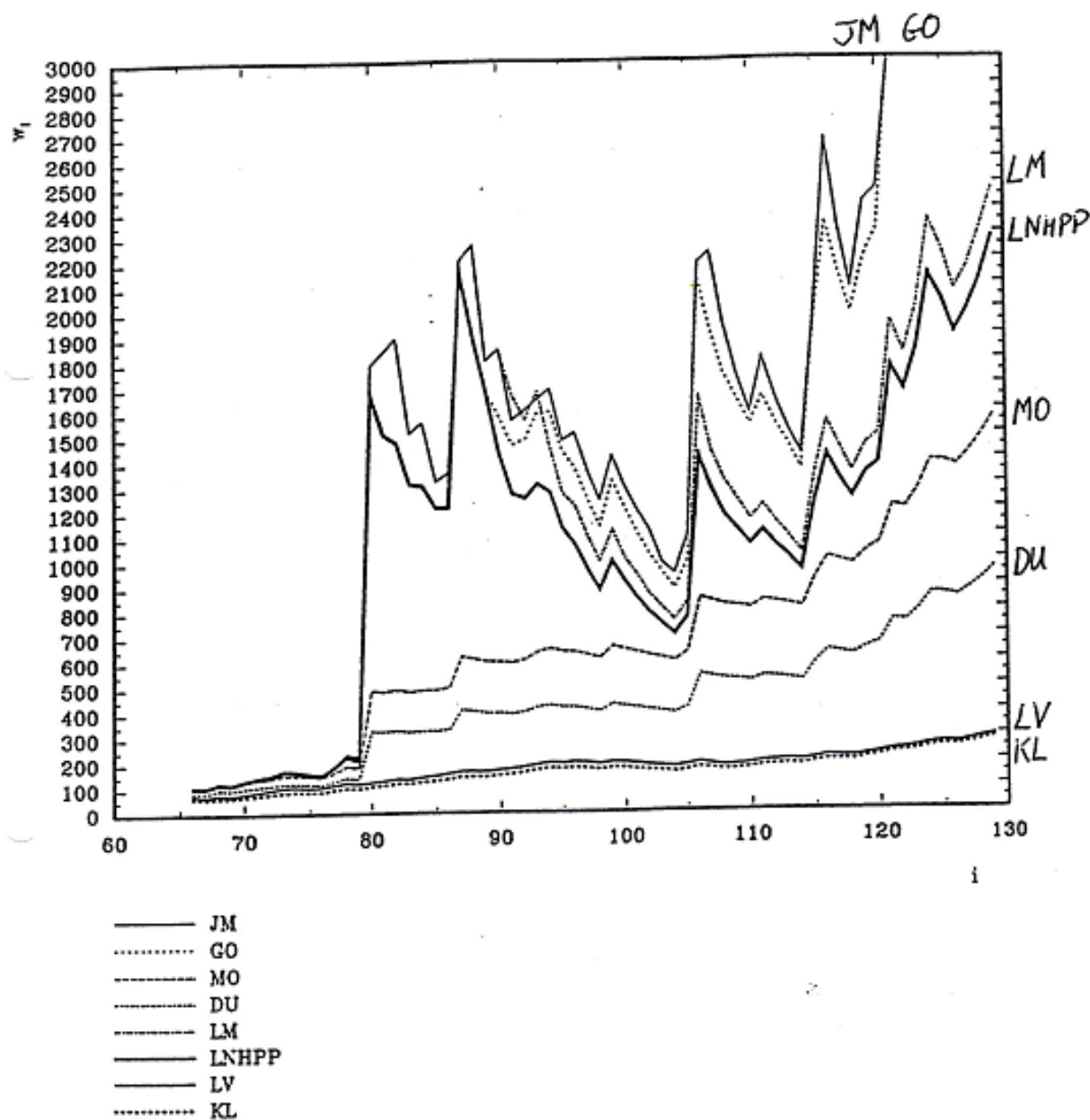
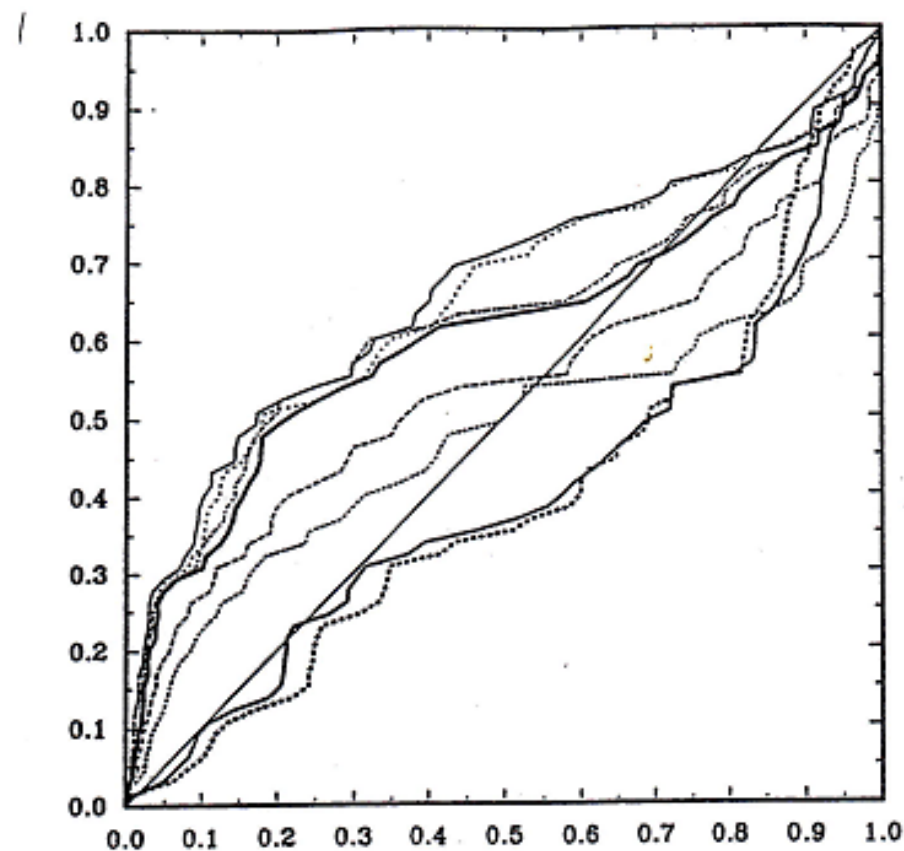




Fig. 5:  $\mu$ -plots from the raw models for data set TSW

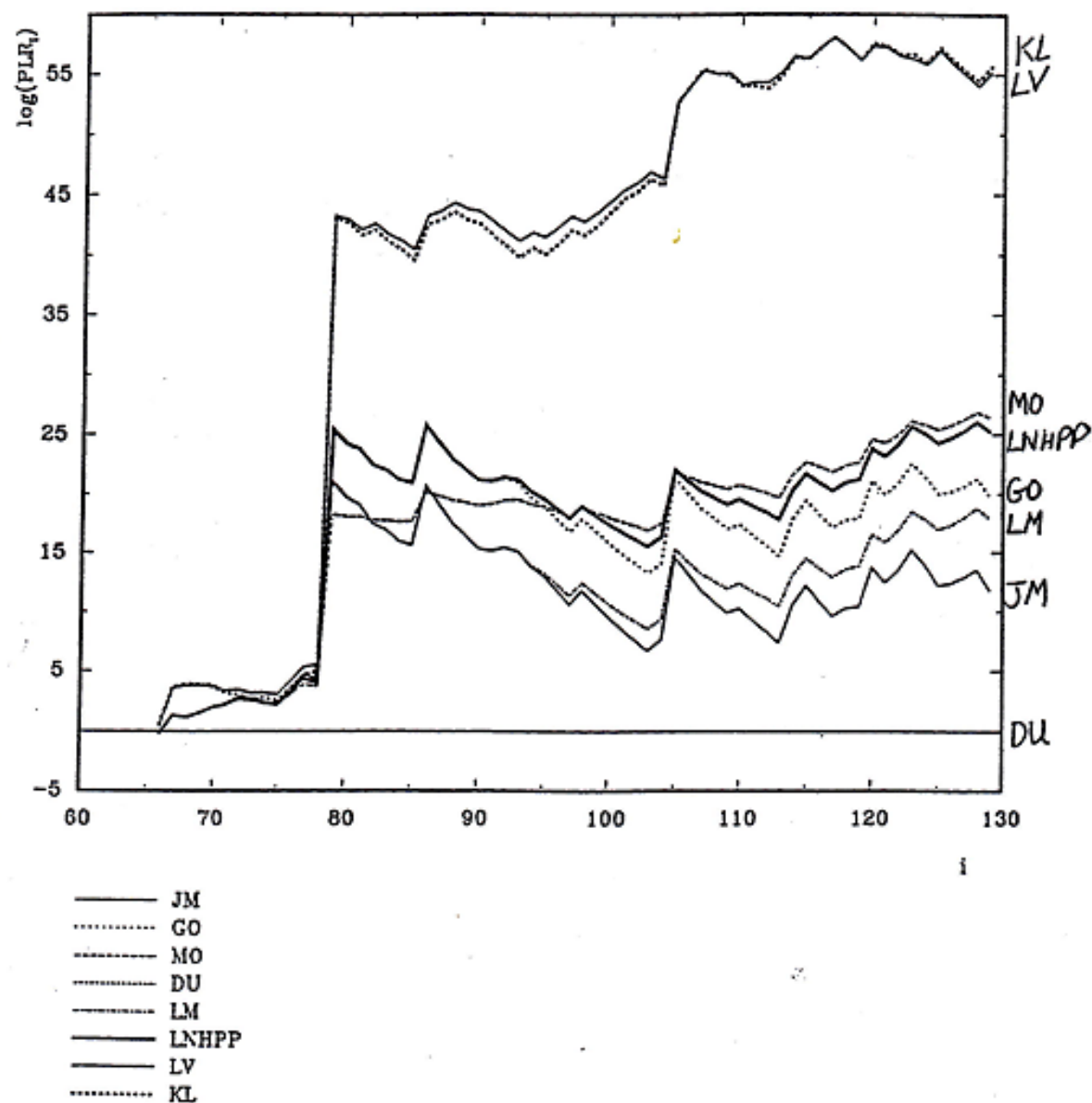


|          | ks distance  | siglev |
|----------|--------------|--------|
| — JM     | 3.336033e-01 | <1%    |
| ..... GO | 3.228883e-01 | <1%    |
| ----- MO | 1.877333e-01 | 1-2%   |
| ----- DU | 2.522298e-01 | <1%    |
| ----- LM | 3.188919e-01 | <1%    |
| — LNHPP  | 2.944701e-01 | <1%    |
| — LV     | 2.785918e-01 | <1%    |
| ..... KL | 2.794085e-01 | <1%    |

Q: c) **Fig 6e** shows the  $\log(\text{PLR})$  plot for each of the raw models when compared against the DU model. The large jumps in these plots coincides with the 79<sup>th</sup> inter-failure (ie.  $t_{79}=9549$ ). By examining **Table 2e** what do you notice is special about this inter-failure time? Comment briefly about the predictions of  $T_{79}$ .

A: c)  $t_{79}$  is extremely large compared to the other inter-failure times. It might be worth checking the recording this time in case there has been a clerical error. The large jump in PLR at  $i=79$  indicates that the KL and LV predictors allocate much more probability to their upper tails than does DU (ie. they assign higher probability to large  $T_i$  than DU does).

Fig. 6e:  $\log(PLR_i)$ -plots for the raw models versus the *DU* model for data set TSW





Q: **d) Figures 7e, 8e and 9e**, shows the corresponding plots for the recalibrated version of the models. Discuss each of these figures when compared with their raw equivalents (ie. compare **7e** with **4e**, **8e** with **5e** and **9e** with **6e**).

A: **d)** Comparing **Figures 4e and 7e** shows that the recalibrated versions of the predictors are more consistent with each other than are the raw predictors. This may well be due to a general convergence on the truth resulting from recalibration, ( at least for the medians). Comparing **Figures 5e and 8e** suggests that recalibration has tended to reduce bias. Comparing **Figures 6e and 9e** we notice a striking similarity in performance between DUS and MOS. The PLR is almost horizontal ( apart from the jump at the outlier,  $t_{79}$ .) Also DU appears to have been improved more by recalibration than several of the other models. (The trends in **Figure 9e** are "less upwards" than the corresponding trends in **Figure 6e** ).

Fig. 7e: Median predictions of  $T_{66}, \dots, T_{129}$ , from the recalibrated models for data set TSW

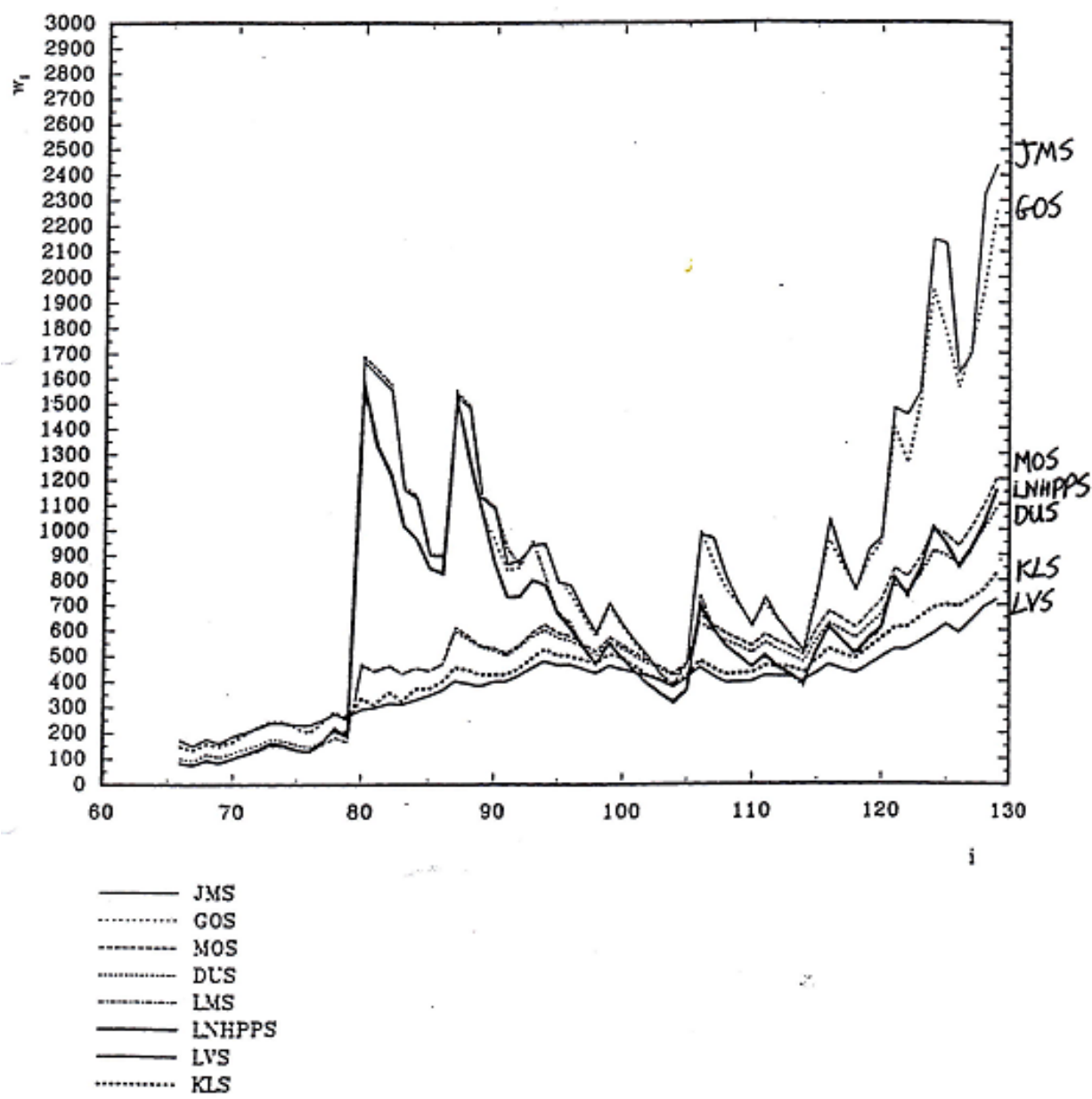
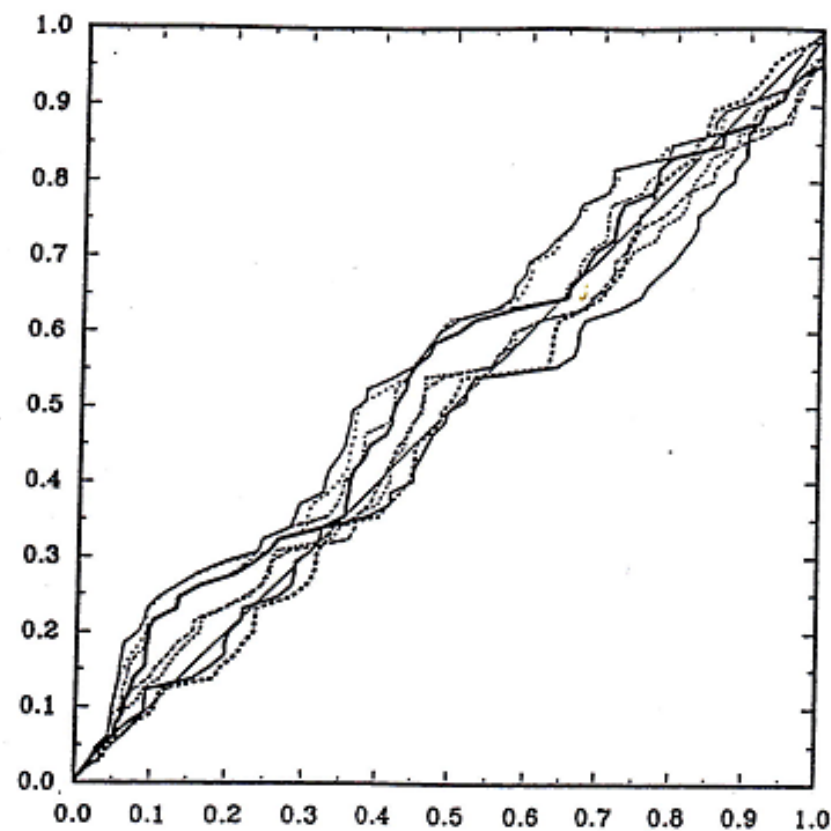


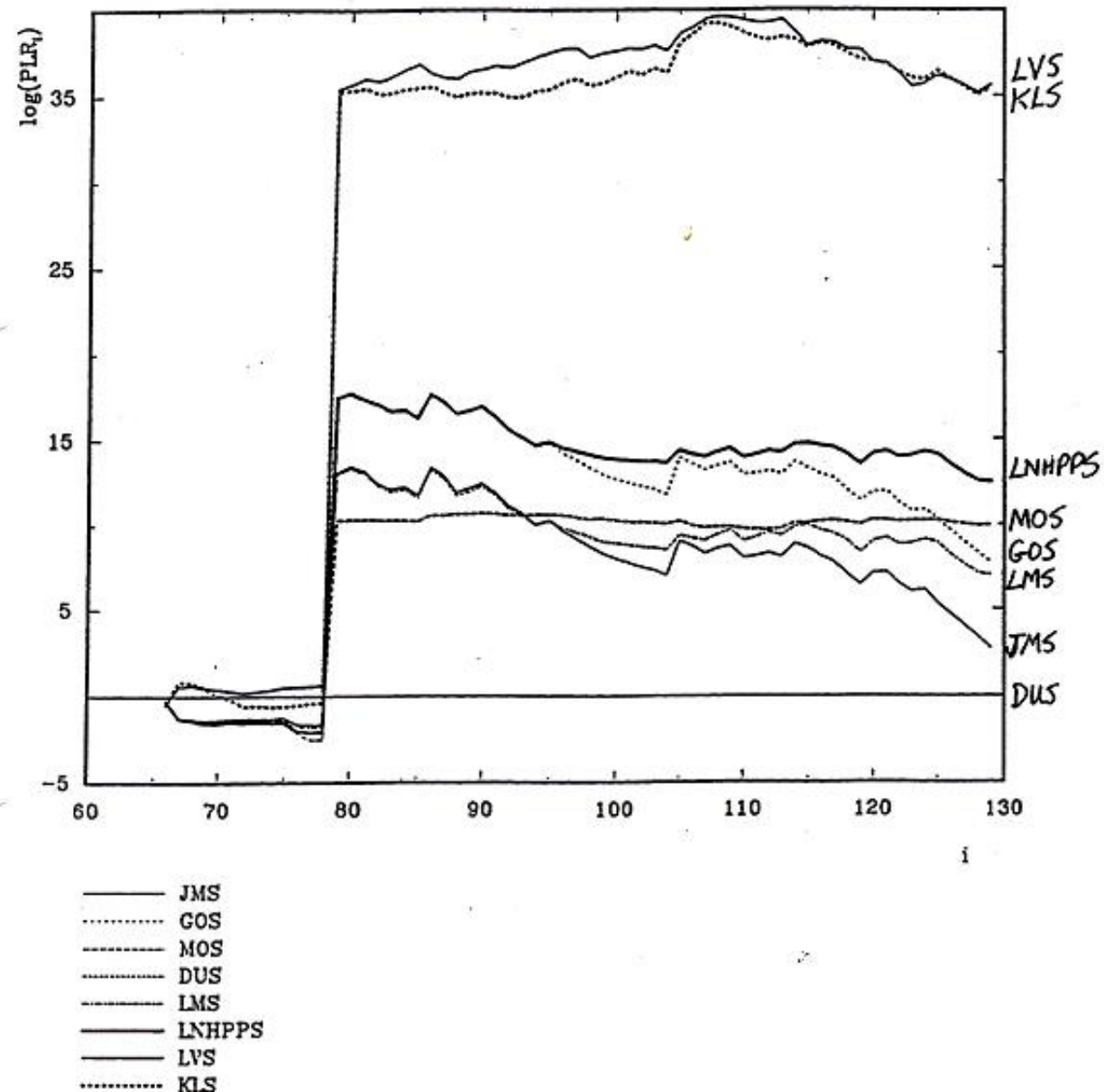
Fig. 8:  $\mu$ -plots from the recalibrated models for data set TSW



|           | ks distance  | siglev |
|-----------|--------------|--------|
| — JMS     | 1.411199e-01 | 10-20% |
| - - - GOS | 1.340998e-01 | 10-20% |
| - - - MOS | 8.471752e-02 | <20%   |
| - - - DUS | 9.291054e-02 | <20%   |
| - - - LMS | 1.200516e-01 | <20%   |
| — LNHPPS  | 1.127576e-01 | <20%   |
| — LVS     | 1.239827e-01 | <20%   |
| - - - KLS | 8.638844e-02 | <20%   |



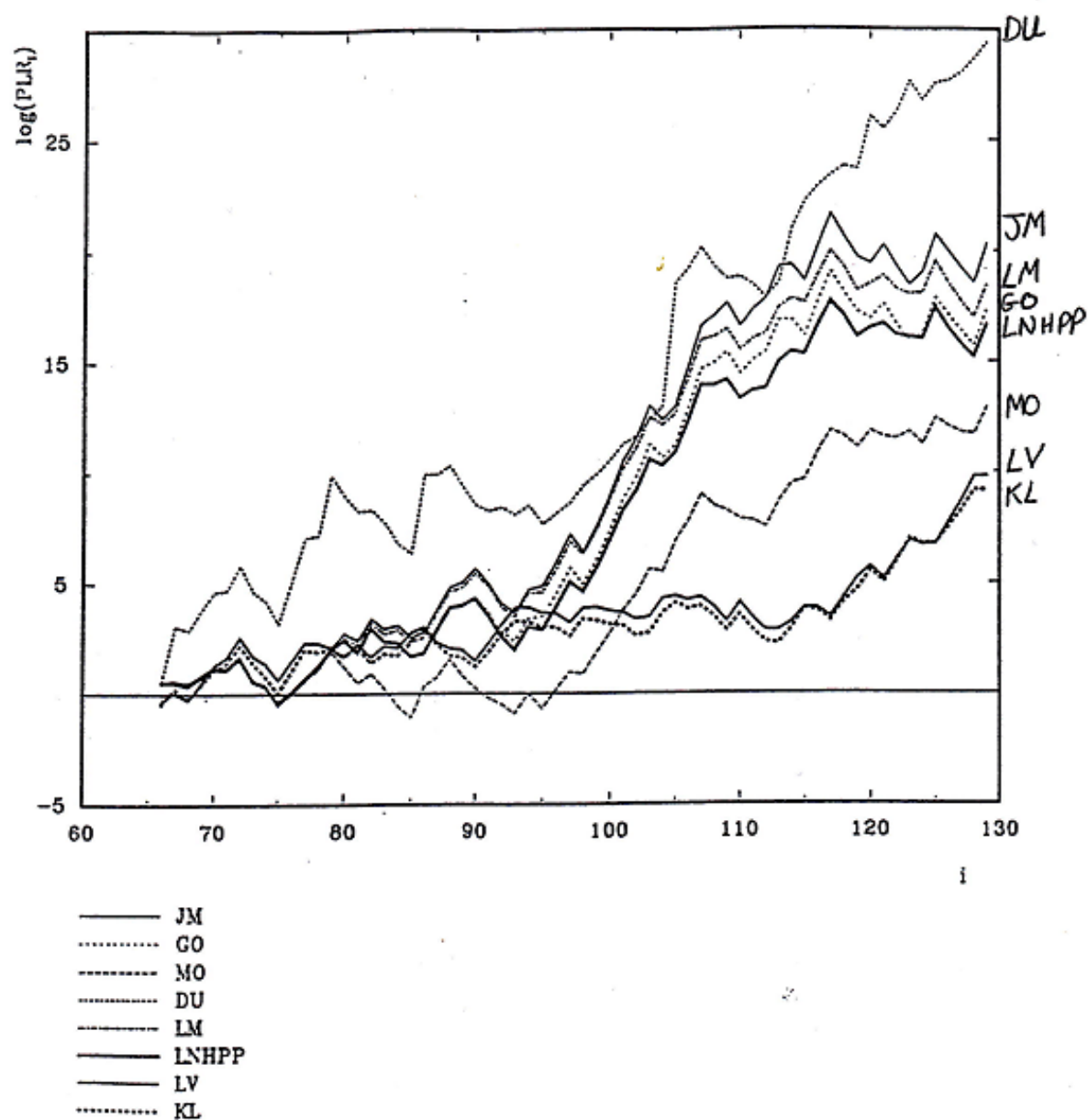
Fig. 9a:  $\log(PLR_i)$ -plots for the recalibrated models versus the *DUS* model for data set TSW



**Q: e) Figure 10e shows the log(PLR) plot for the recalibrated predictions versus the raw predictions (ie. recalibrated JM versus raw JM, etc.). Comment on this plot.**

**A: e) Recalibration seems to improve all the predictors, and especially the DU.**

Fig. 10a:  $\log(PLR)$ -plots for the recalibrated models versus the raw models for data set TSW



Q: f) Based on all the plots state which prediction system (or systems) you use for the next reliability prediction (ie. the 130<sup>th</sup> inter-failure time,  $T_{130}$ ) for this data set? Discuss briefly what led you to make this choice.

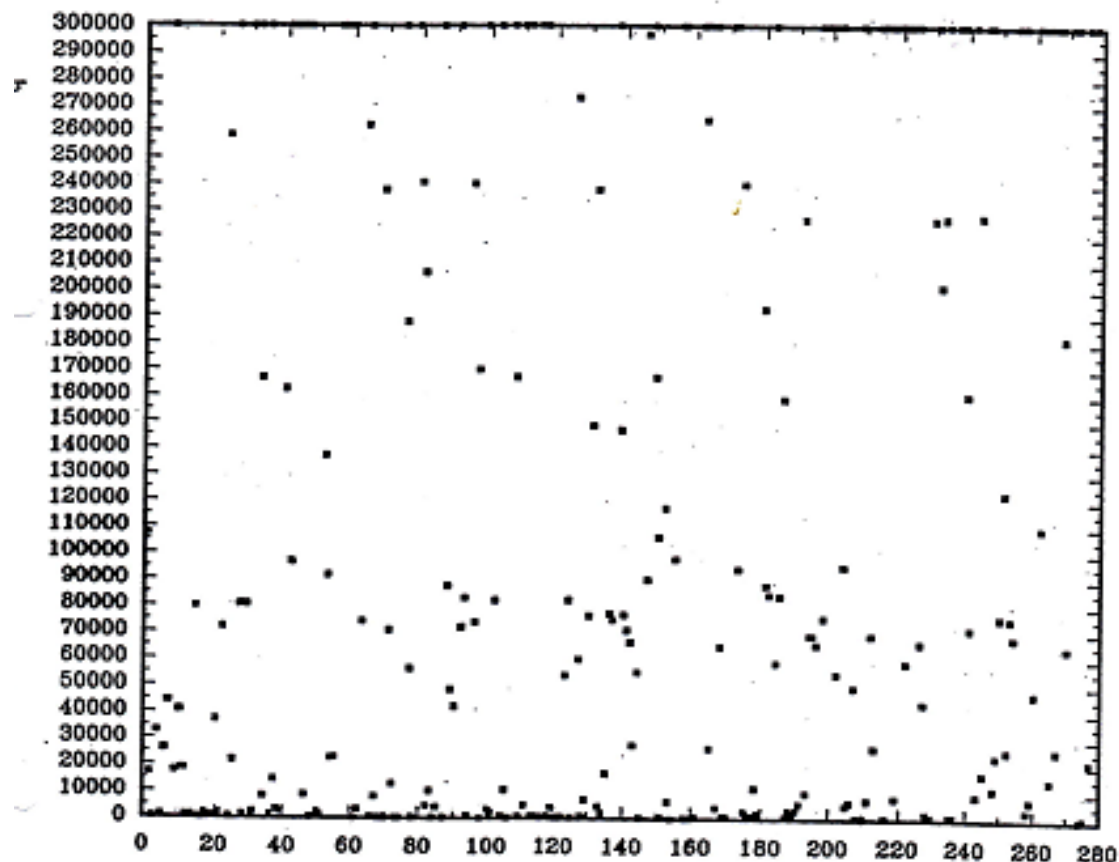
A: f) MOS or DUS best on PLR and have good u-plots. These would be favored over LVS and KLS because, from **Figure 9e**, there is evidence that MOS and DUS have performed better recently (i.e. for the last 25 predictions).



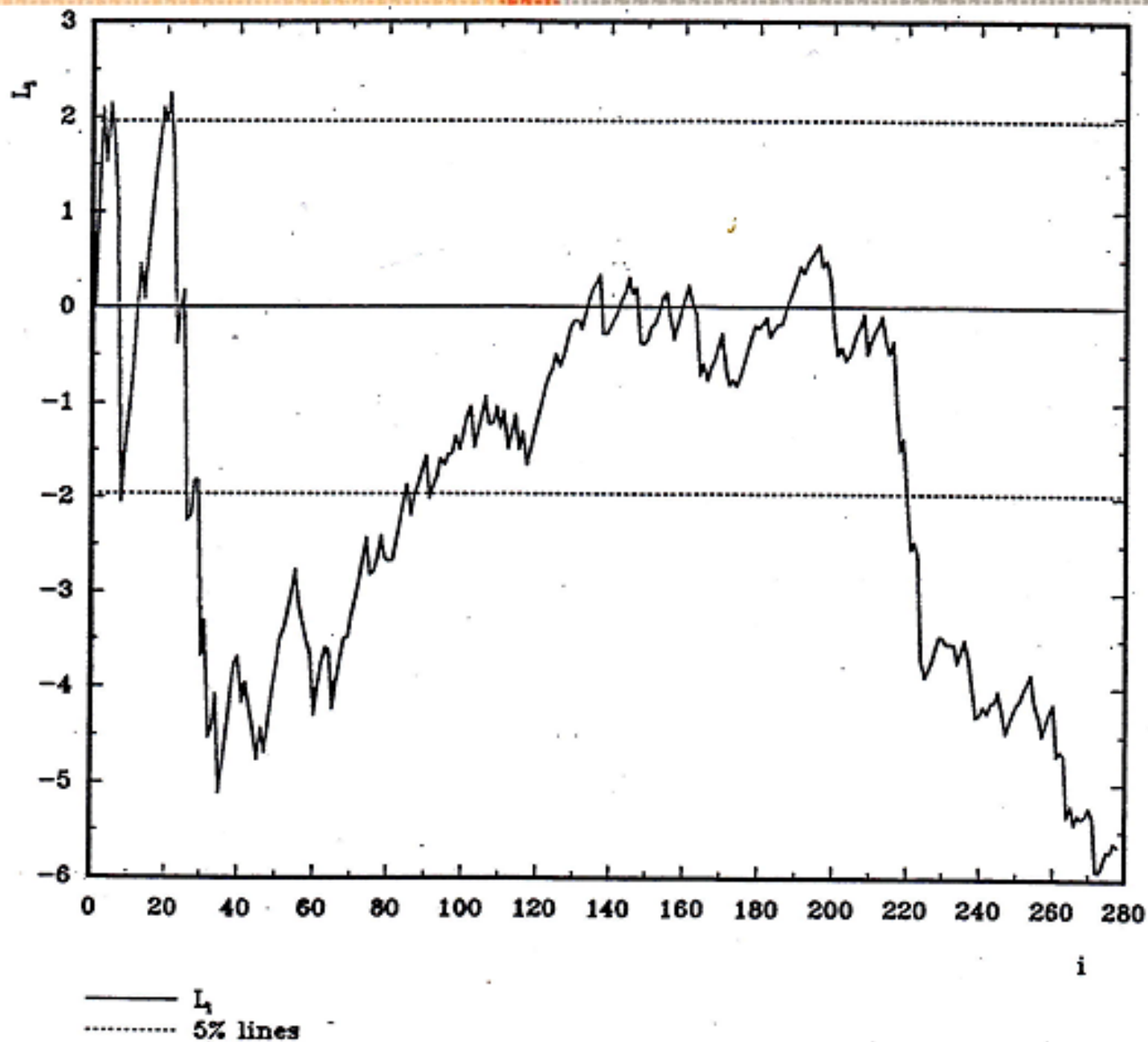
SS3 interfailure data set

|         |         |        |         |         |
|---------|---------|--------|---------|---------|
| 107400  | 17220   | 180    | 32880   | 960     |
| 26100   | 44160   | 333720 | 17820   | 40860   |
| 18780   | 960     | 960    | 79860   | 240     |
| 120     | 1800    | 480    | 780     | 37260   |
| 2100    | 72080   | 258704 | 480     | 21900   |
| 478620  | 80760   | 1200   | 80700   | 688860  |
| 2220    | 758880  | 168820 | 8280    | 951354  |
| 1320    | 14700   | 3420   | 2520    | 162480  |
| 520320  | 96720   | 418200 | 434760  | 543780  |
| 8820    | 488280  | 480    | 540     | 2220    |
| 1080    | 137340  | 91860  | 22800   | 22920   |
| 473340  | 354901  | 369480 | 380220  | 848640  |
| 120     | 3416    | 74160  | 262500  | 879300  |
| 360     | 8160    | 180    | 237920  | 120     |
| 70800   | 12960   | 300    | 120     | 558540  |
| 188040  | 56280   | 420    | 414464  | 240780  |
| 206640  | 4740    | 10140  | 300     | 4140    |
| 472080  | 300     | 87600  | 48240   | 41940   |
| 576612  | 71820   | 83100  | 900     | 240300  |
| 73740   | 169800  | 1      | 302280  | 3360    |
| 2340    | 82260   | 559920 | 780     | 10740   |
| 180     | 430860  | 166740 | 600     | 376140  |
| 5100    | 549540  | 540    | 900     | 521252  |
| 420     | 518840  | 1020   | 4140    | 480     |
| 180     | 600     | 53760  | 82440   | 180     |
| 273000  | 59880   | 840    | 7140    | 76320   |
| 148680  | 237840  | 4560   | 1920    | 16860   |
| 77040   | 74760   | 738180 | 147000  | 76680   |
| 70800   | 66180   | 27540  | 55020   | 120     |
| 296796  | 90180   | 724560 | 167100  | 106200  |
| 480     | 117360  | 6480   | 60      | 97860   |
| 398580  | 391380  | 180    | 180     | 240     |
| 540     | 336900  | 264480 | 847080  | 26460   |
| 349320  | 4080    | 84880  | 840     | 540     |
| 589980  | 332280  | 94140  | 240060  | 2700    |
| 900     | 1080    | 11580  | 2160    | 192720  |
| 87840   | 84360   | 378120 | 58500   | 83880   |
| 158640  | 660     | 3180   | 1560    | 3180    |
| 5700    | 228580  | 9840   | 69060   | 68880   |
| 65460   | 402900  | 75480  | 380220  | 704968  |
| 505680  | 54420   | 319020 | 95220   | 5100    |
| 6240    | 49440   | 420    | 687320  | 120     |
| 7200    | 68940   | 26520  | 448620  | 339420  |
| 480     | 1042680 | 779580 | 8040    | 1158240 |
| 907140  | 58500   | 383940 | 2039460 | 522240  |
| 66000   | 43500   | 2040   | 600     | 228320  |
| 327800  | 201300  | 226980 | 553440  | 1020    |
| 960     | 542760  | 819240 | 801660  | 160380  |
| 71640   | 363990  | 9090   | 227970  | 17190   |
| 597900  | 689400  | 11520  | 23850   | 75870   |
| 123030  | 26010   | 75240  | 68130   | 811050  |
| 498360  | 623280  | 3330   | 7290    | 47160   |
| 1328400 | 109800  | 343890 | 1615860 | 14940   |
| 680760  | 26220   | 376110 | 181690  | 64320   |
| 468180  | 1568580 | 333720 | 180     | 810     |
| 322110  | 21960   | 363600 |         |         |

Inter-failure time vs failure no : ss3

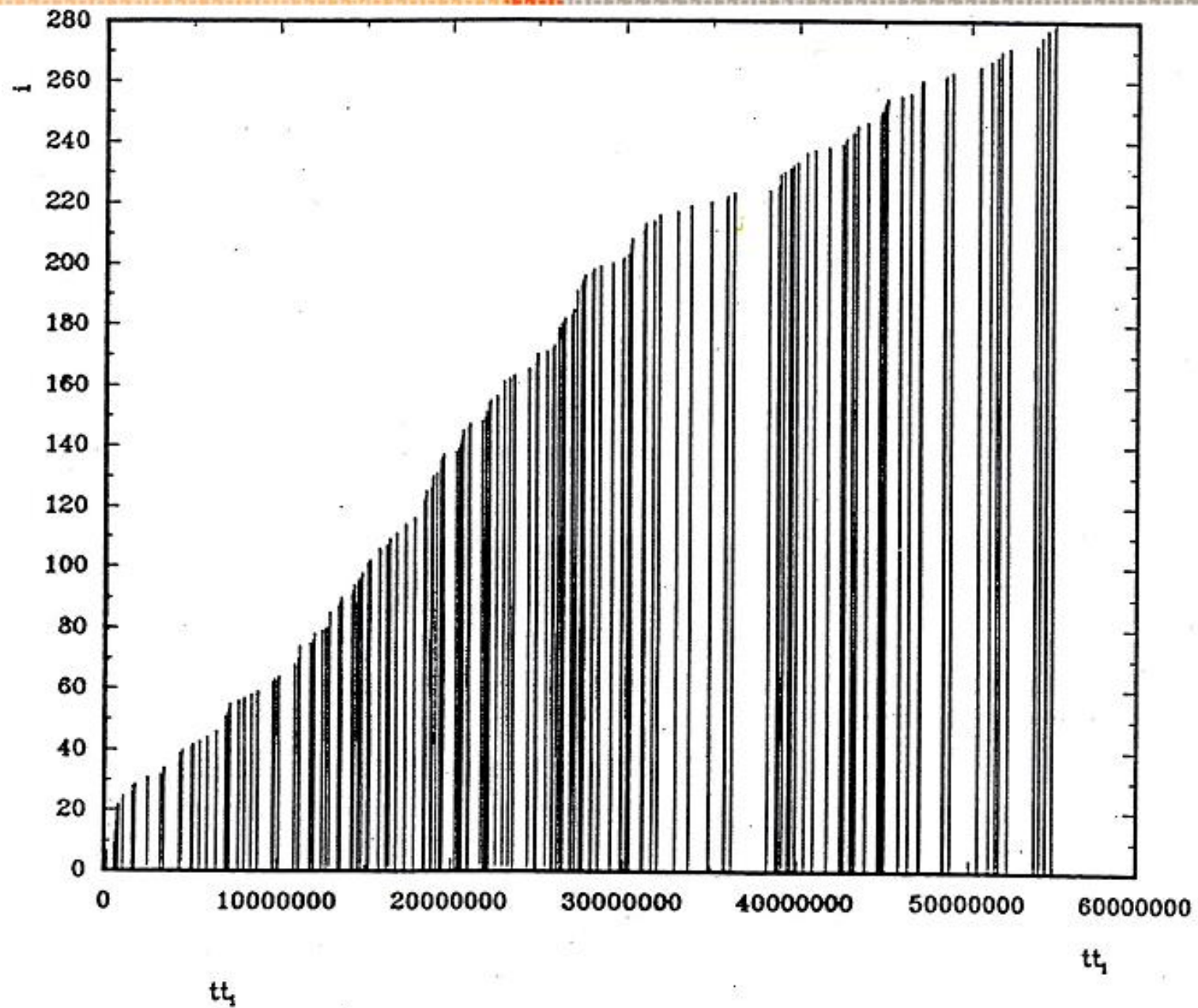


Laplace statistic vs failure no : ss3



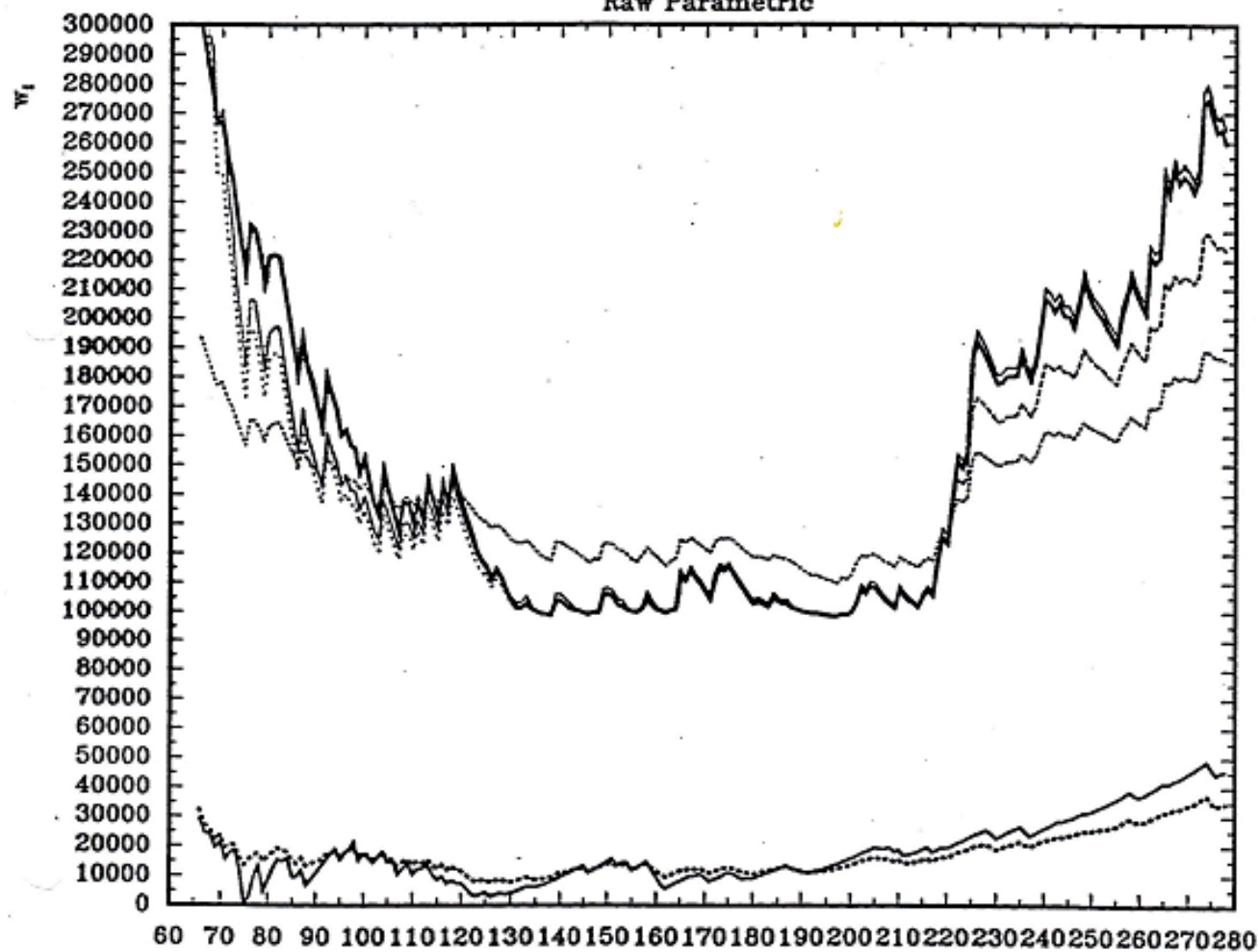


Cumulative failures vs total elapsed time : ss3



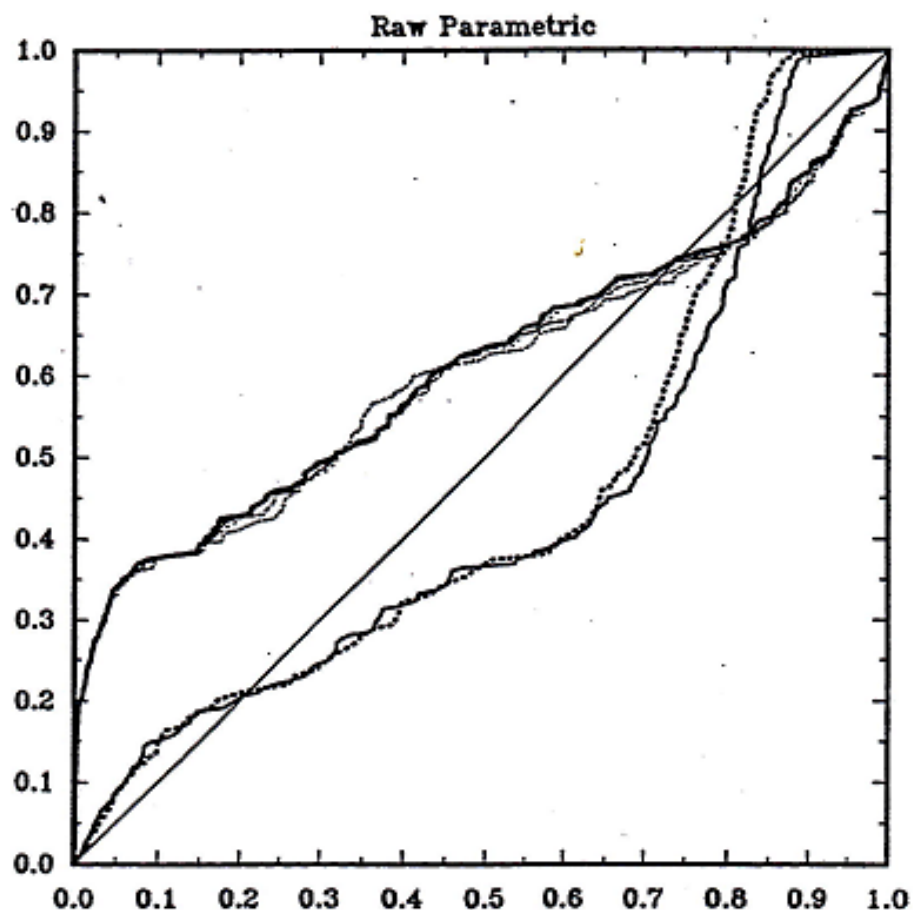
# Medians versus i for data set ss3

Raw Parametric



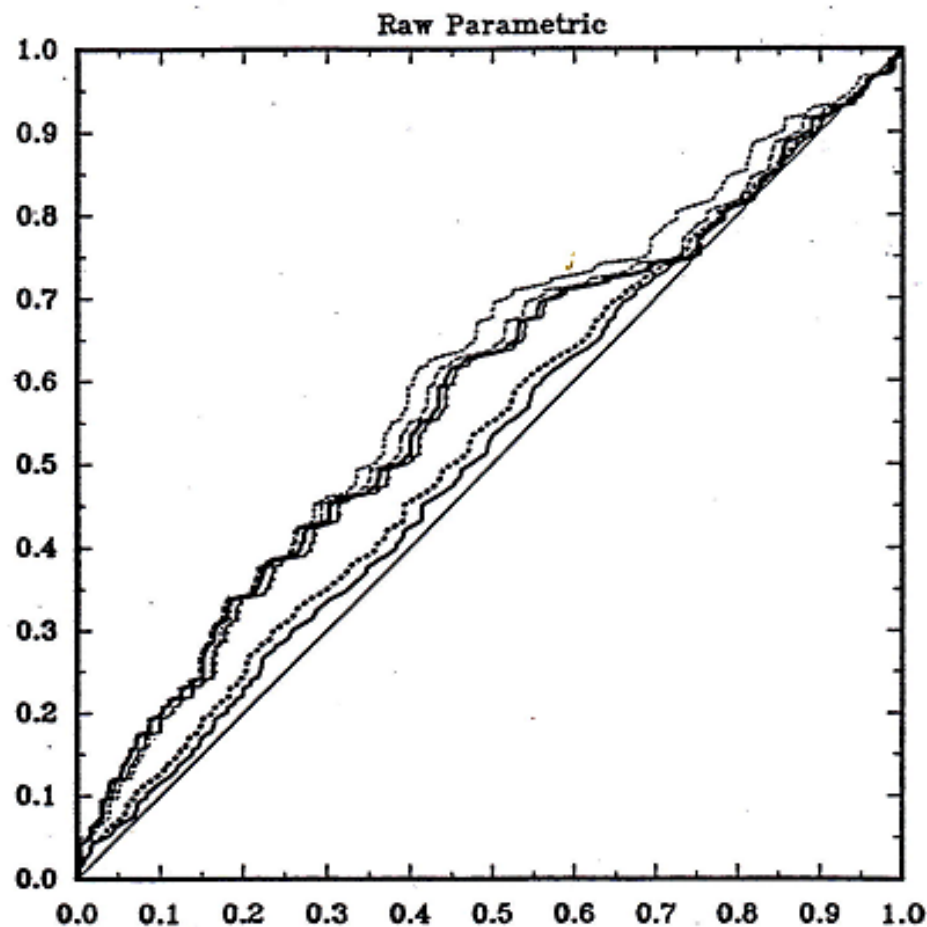
— JM  
 ..... GO  
 — MO  
 — DU  
 — LM  
 — LNHPP  
 — LV  
 ..... KL

# u-plot for data set ss3



|          | ks distance  | siglev |
|----------|--------------|--------|
| — JM     | 2.940505e-01 | >1%    |
| ..... GO | 2.928419e-01 | >1%    |
| —— MO    | 2.895953e-01 | >1%    |
| —— DU    | 2.870137e-01 | >1%    |
| —— LM    | 2.940440e-01 | >1%    |
| —— LNHPP | 2.928020e-01 | >1%    |
| —— LV    | 2.299730e-01 | >1%    |
| —— KL    | 2.153911e-01 | >1%    |

# y-plot for data set ss3

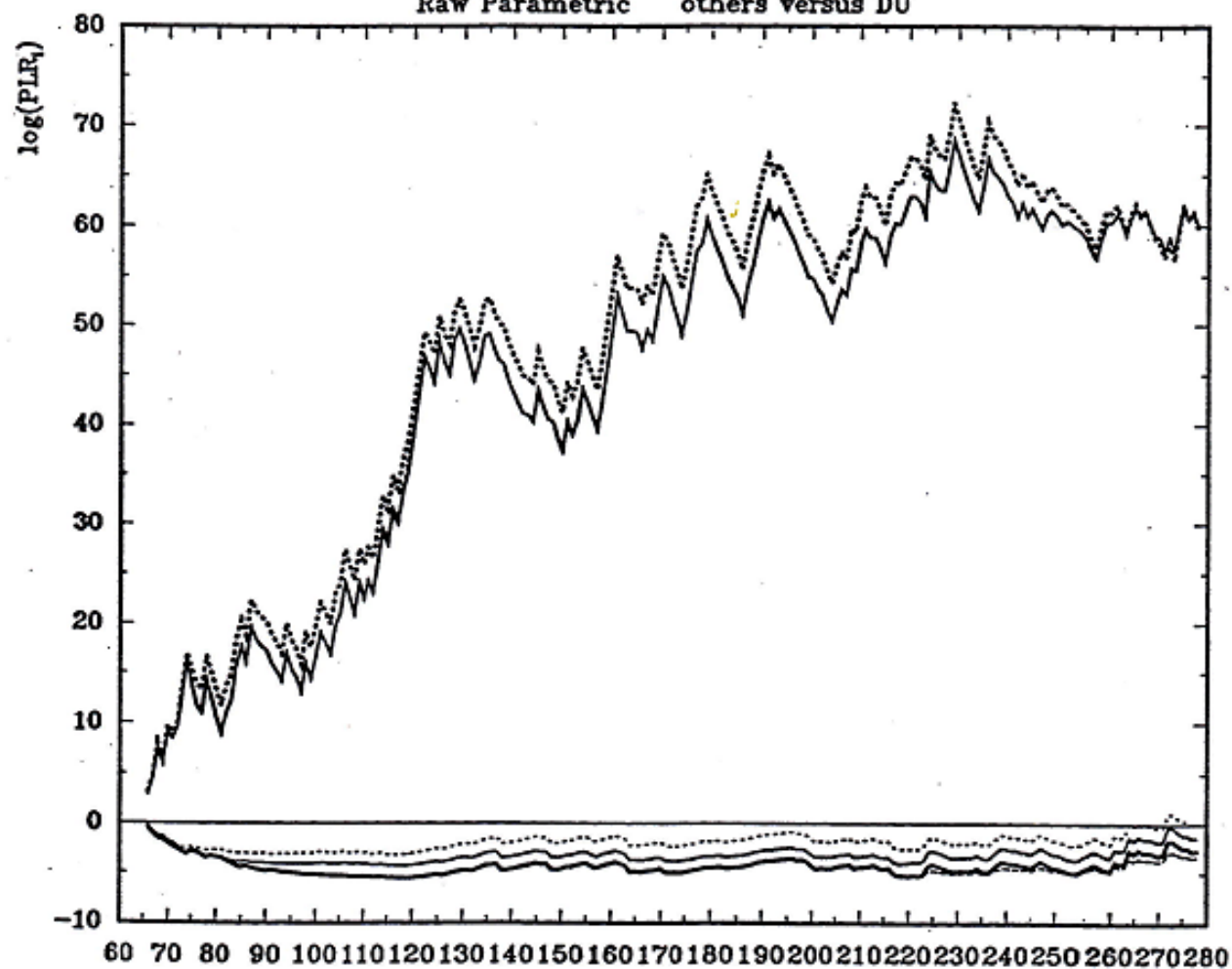


|          | ks distance  | siglev |
|----------|--------------|--------|
| — JM     | 1.571294e-01 | >1%    |
| ..... GO | 1.548109e-01 | >1%    |
| — MO     | 1.799047e-01 | >1%    |
| — DU     | 2.058513e-01 | >1%    |
| — LM     | 1.581629e-01 | >1%    |
| — LNHPP  | 1.660730e-01 | >1%    |
| — LV     | 4.411142e-02 | <20%   |
| ..... KL | 6.398147e-02 | <20%   |



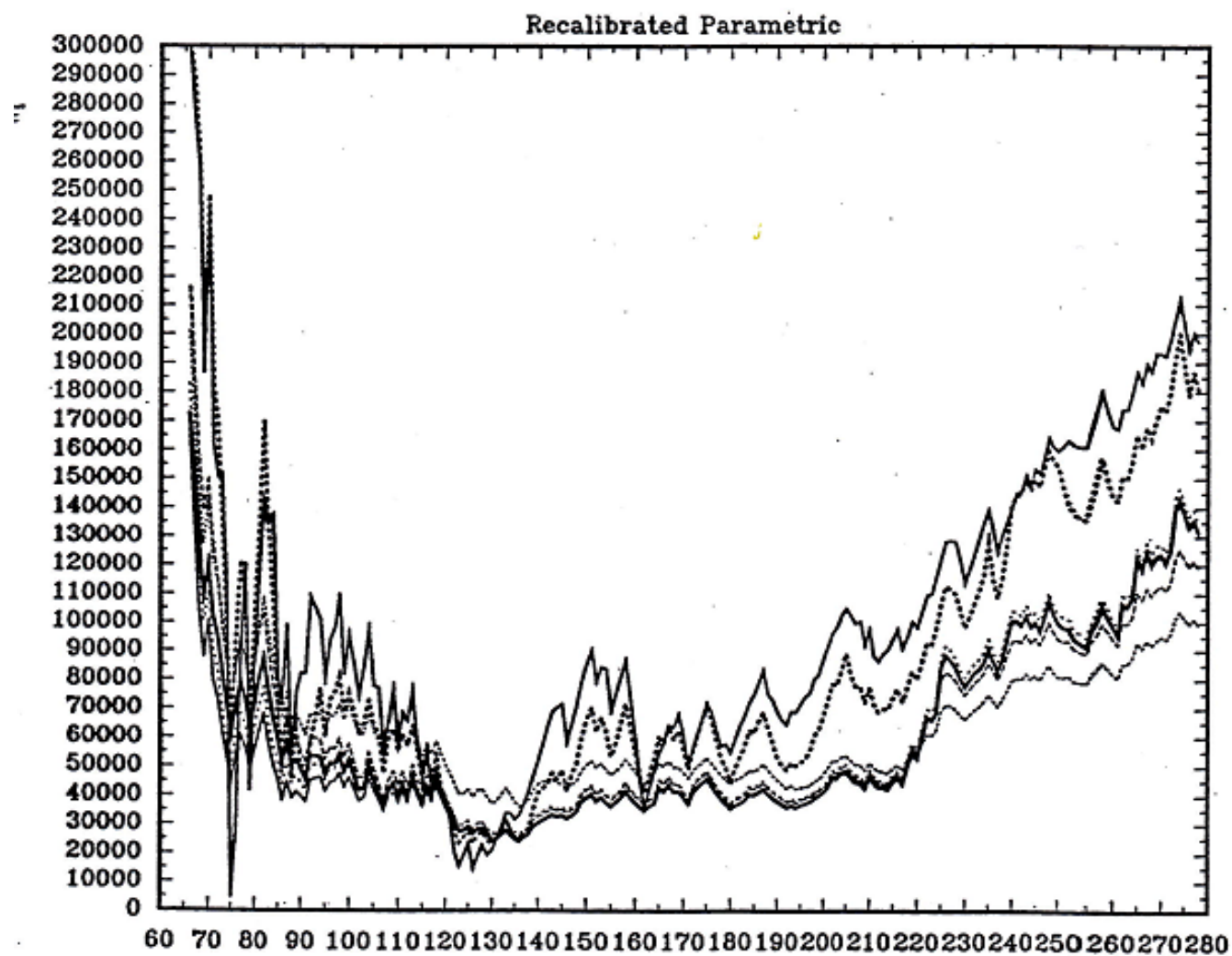
# Log(PLR) versus i for data set ss3

Raw Parametric others versus DU



— JM  
 ..... GO  
 — MO  
 — DU  
 — LM  
 — LNHPP  
 — LV  
 ..... KL

# Medians versus i for data set ss3

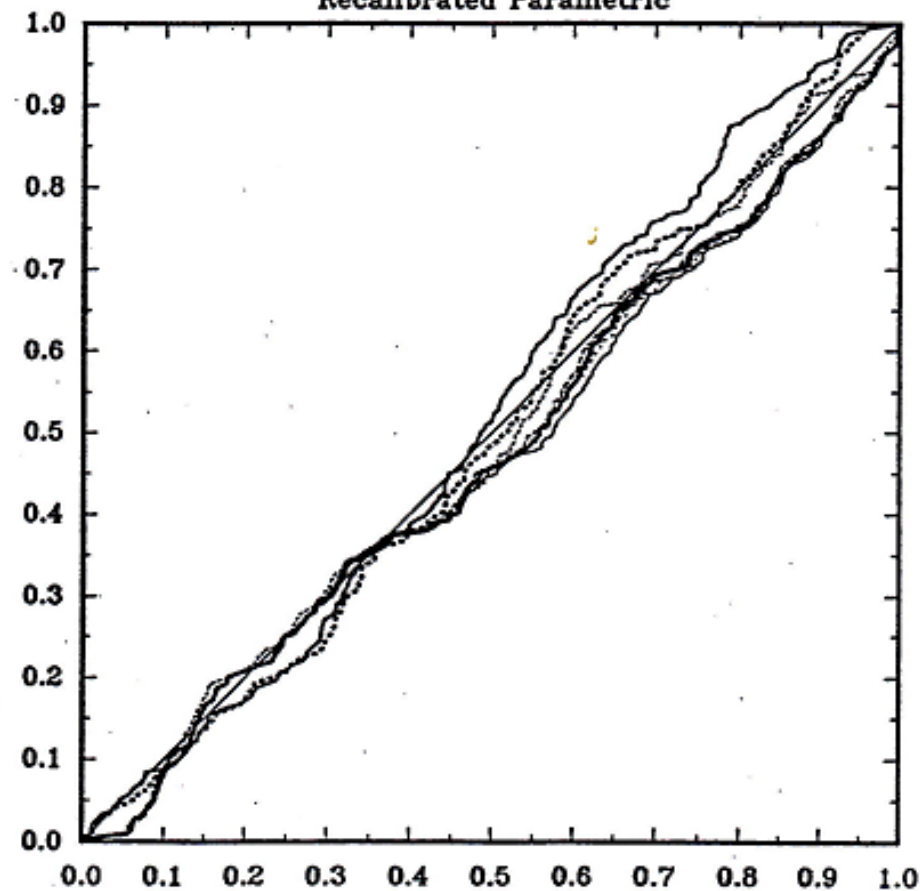


— JMS  
 ..... GOS  
 — MOS  
 — DUS  
 — LMS  
 — LNHPPS  
 — LVS  
 ..... KLS



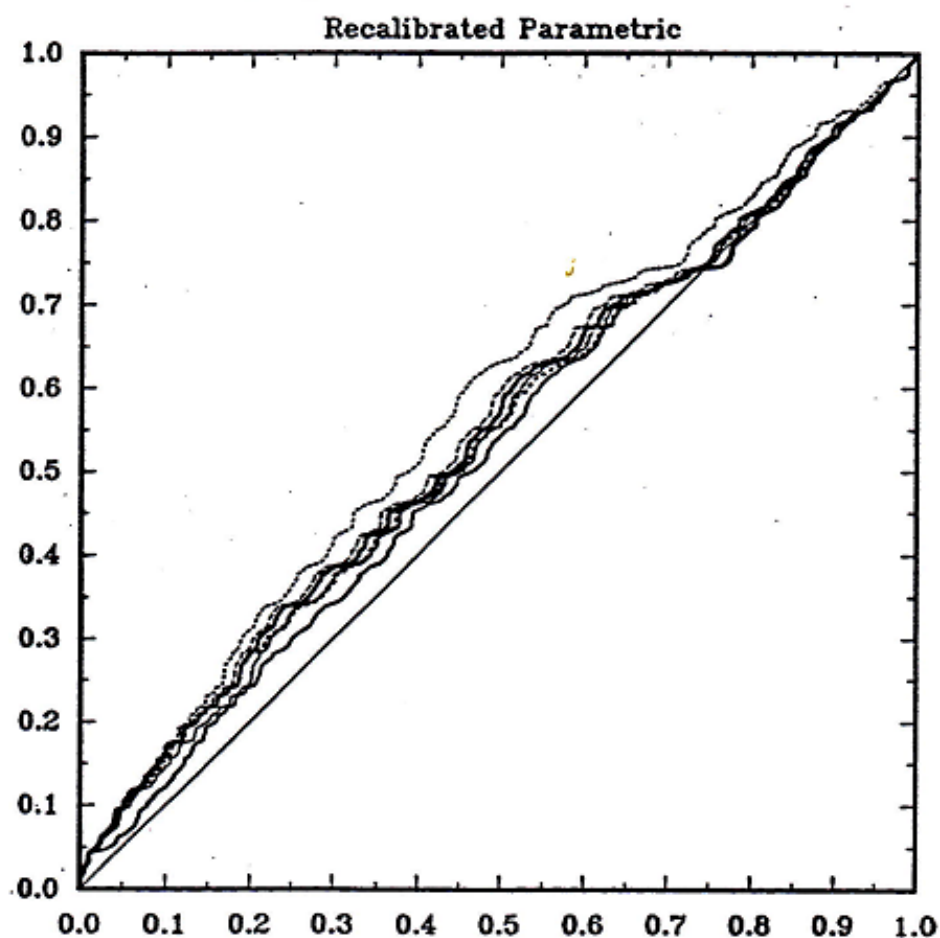
# u-plot for data set ss3

Recalibrated Parametric



|           | ks distance  | siglev |
|-----------|--------------|--------|
| — JMS     | 8.397251e-02 | 5-10%  |
| ..... GOS | 7.326858e-02 | 10-20% |
| — MOS     | 6.270546e-02 | <20%   |
| — DUS     | 5.632747e-02 | <20%   |
| — LMS     | 8.477152e-02 | 5-10%  |
| — LNEPPS  | 6.828946e-02 | <20%   |
| — LVS     | 8.694181e-02 | 5-10%  |
| ..... KLS | 6.401303e-02 | <20%   |

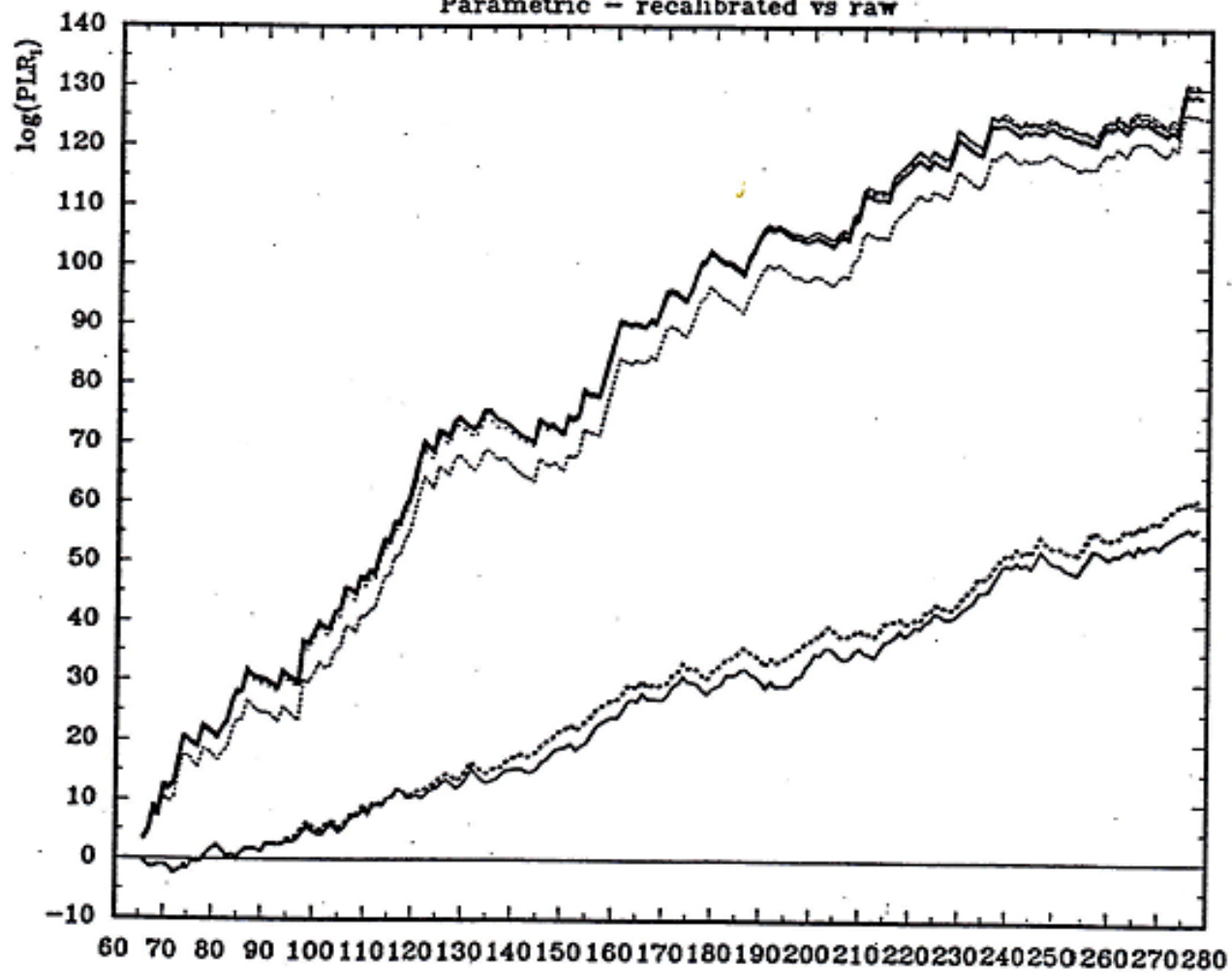
# y-plot for data set ss3



|           | ks distance  | siglev |
|-----------|--------------|--------|
| —— JMS    | 8.161902e-02 | 10-20% |
| ..... GOS | 8.279800e-02 | 10-20% |
| —— MOS    | 1.047152e-01 | 1-2%   |
| —— DUS    | 1.457976e-01 | >1%    |
| —— LMS    | 8.313545e-02 | 10-20% |
| —— LNHPPS | 9.673659e-02 | 2-5%   |
| —— LVS    | 5.745424e-02 | <20%   |
| ..... KLS | 8.173585e-02 | 10-20% |

# Log(PLR) versus i for data set ss3

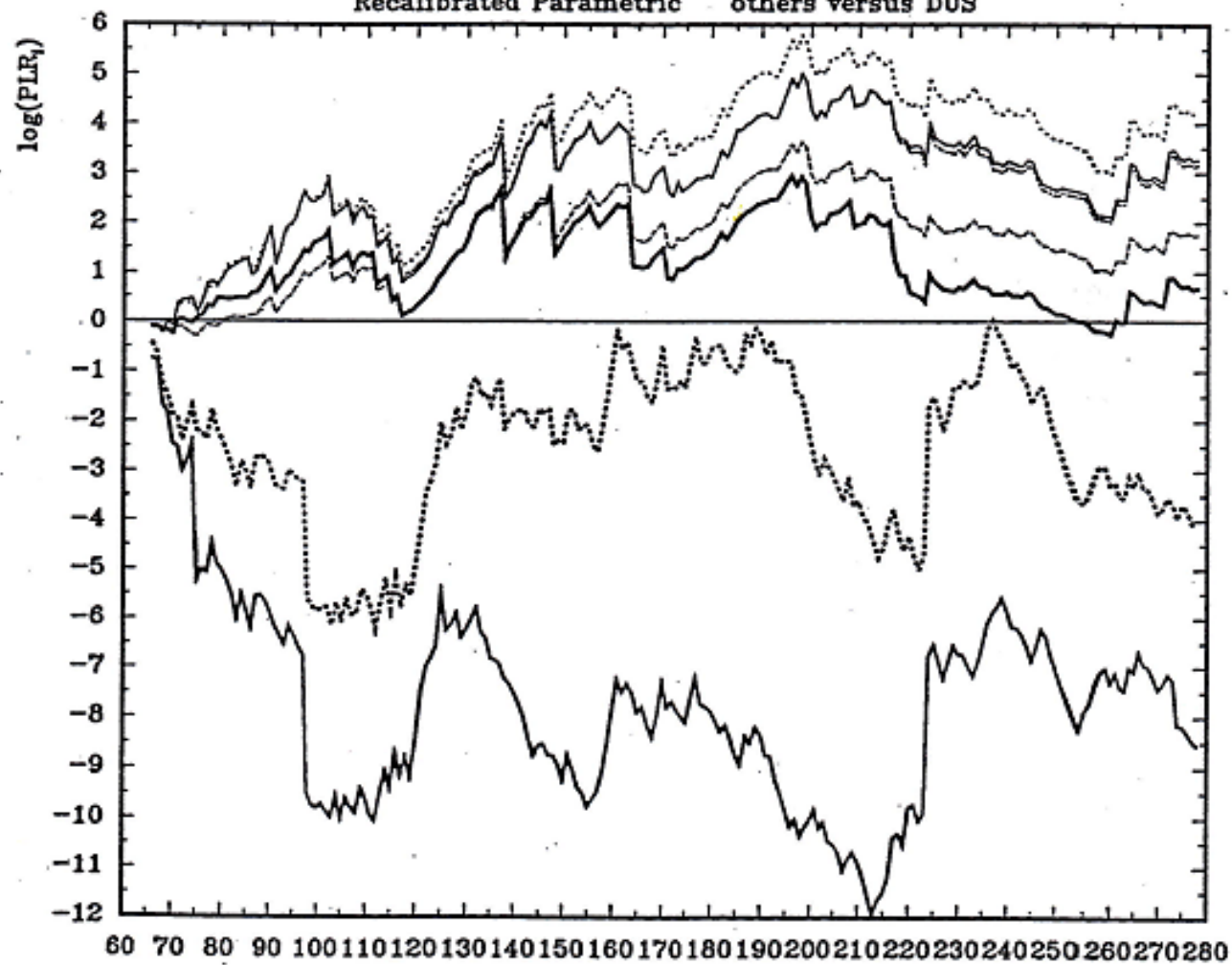
Parametric - recalibrated vs raw



— JM  
 ..... GO  
 — MO  
 — DU  
 — LM  
 — LNHPP  
 — LV  
 ..... KL

# Log(PLR) versus i for data set ss3

Recalibrated Parametric    others versus DUS



- JMS
- ... GOS
- MOS
- DUS
- LMS
- LNHPPS
- LVS
- ... KLS

# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
- 6. Software Reliability Engineering Practice**
7. Lessons Learned
8. Background Literature



## 6. Software Reliability Engineering Practice

- Software Reliability Tools
- SMERFS Main Features
- SREPT Main Features
- CASRE Main Features
- Frestimate Main Features
- CASRE in large
- Frestimate in large
- Examples
- Conclusions



# Software Reliability Tools

- Statistical Modelling and Estimation of Reliability Functions for Software (**SMERFS**) [William Farr of Naval Surface Warfare Center]
- **SREPT** (Software Reliability Estimation and Prediction Tool) [Center for Advanced Computing and Communication Department of Electrical and Computer Engineering Duke University]
- Computer-Aided Software Reliability Estimation Tool (**CASRE**) [Allen Nikora, JPL & Michael Lyu, Chinese University of Hong Kong]
- **Frestimate** [SoftRel, Ann Marie Neufelder, <http://www.softrel.com/prod01.htm>]
- etc.

# SMERFS Main Features

- Multiple Models (12)
- Model Application Scheme: Single Execution
- Data Format: Failure-Counts and Time-Between Failures
- On-line Model Description Manual
- Two parameter Estimation Methods
- Least Square Method
- Maximum Likelihood Method
- Goodness-of-fit Criteria: Chi-Square Test, KS Test
- Model Applicability - Prequential Likelihood, Bias, Bias Trend, Model Noise
- Simple Plots

# SREPT

SREPT : Software Reliability Estimation and Prediction Tool

Seprt Engines Plots Help

File containing number of lines data : C:\Users\UBAS\Desktop\SREPT\Sre

| Module name  | Number of lines | Number of faults |
|--------------|-----------------|------------------|
| analyze.c    | 946             | 37.84            |
| multpath.c   | 387             | 15.48            |
| share.c      | 1977            | 79.08            |
| pfqn.c       | 1155            | 46.2             |
| reachgraph.c | 1791            | 71.64            |
| mpfqn.c      | 1142            | 45.68            |
| util.c       | 1119            | 44.76            |
| inspade.c    | 880             | 35.2             |
| indist.c     | 680             | 27.2             |
| debug.c      | 259             | 10.36            |

Save table

Clear table

Click to read data into table from file

Estimate the number of faults

Experience from past projects:

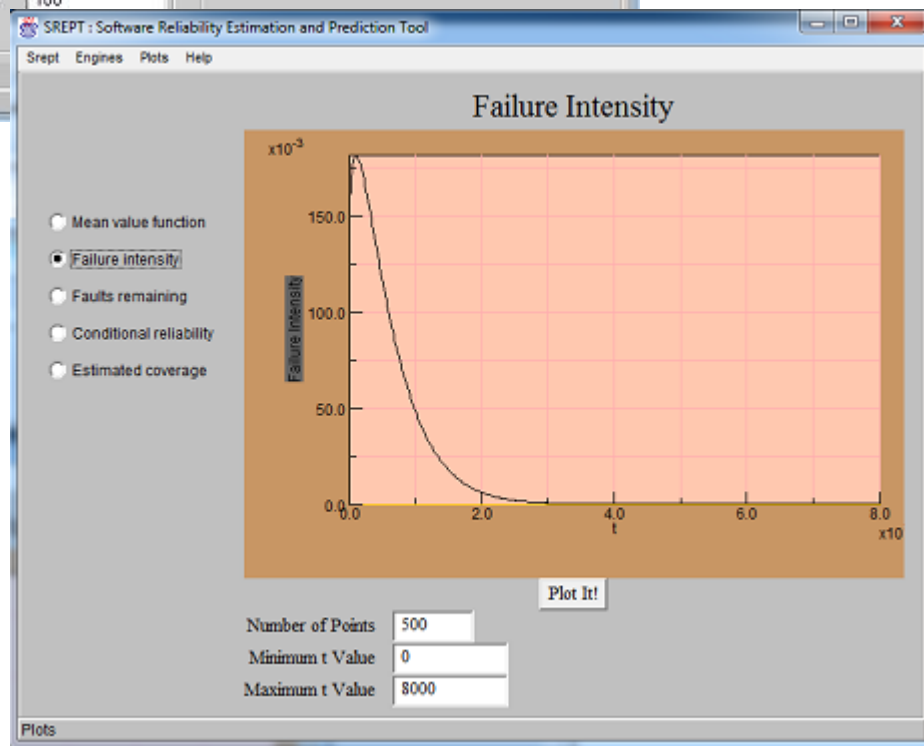
Total number of faults in a previous project : 4

Number of lines of code in the same project : 100

Estimates for the current project

Number of faults : 1415.84

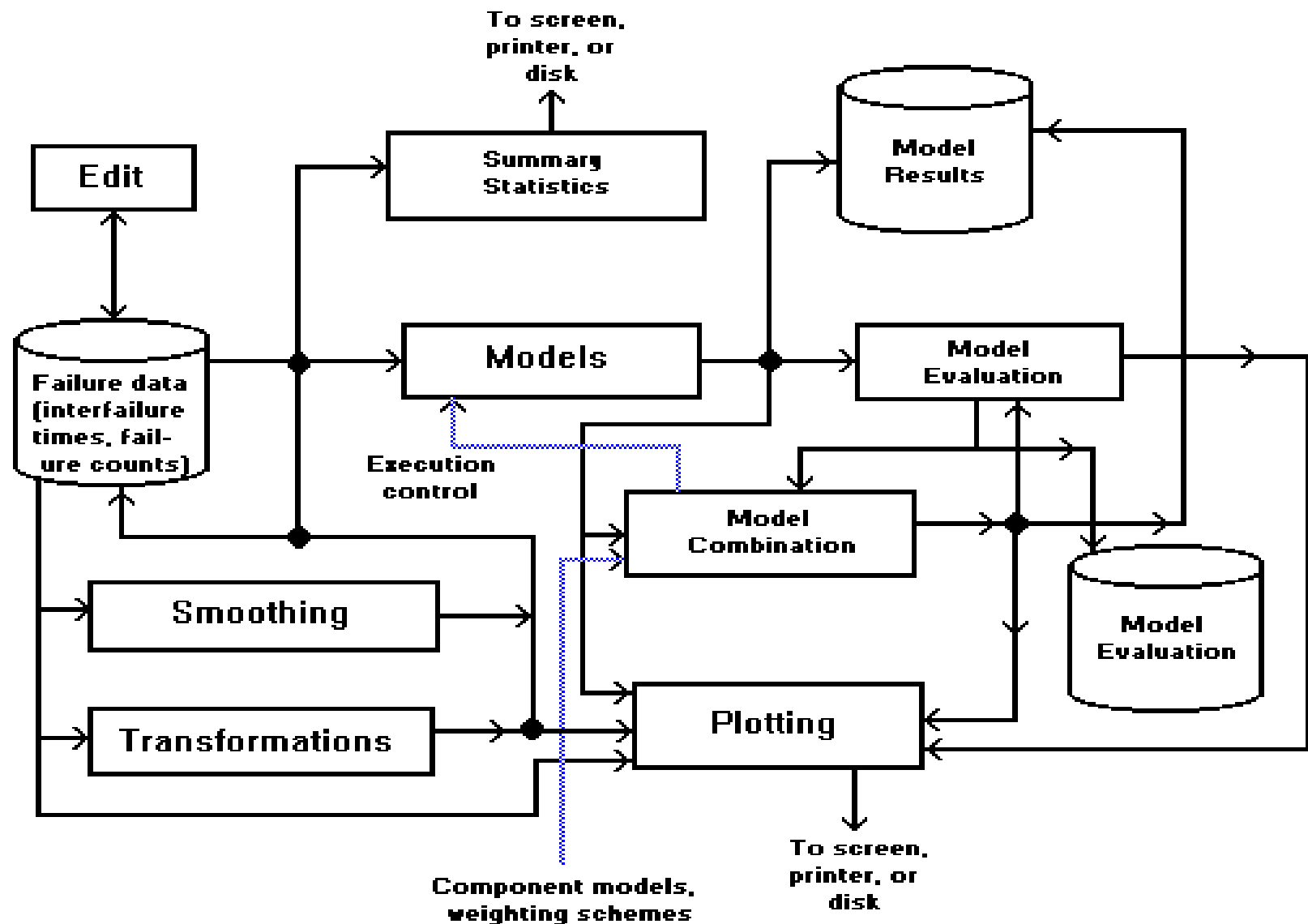
Fault density engine



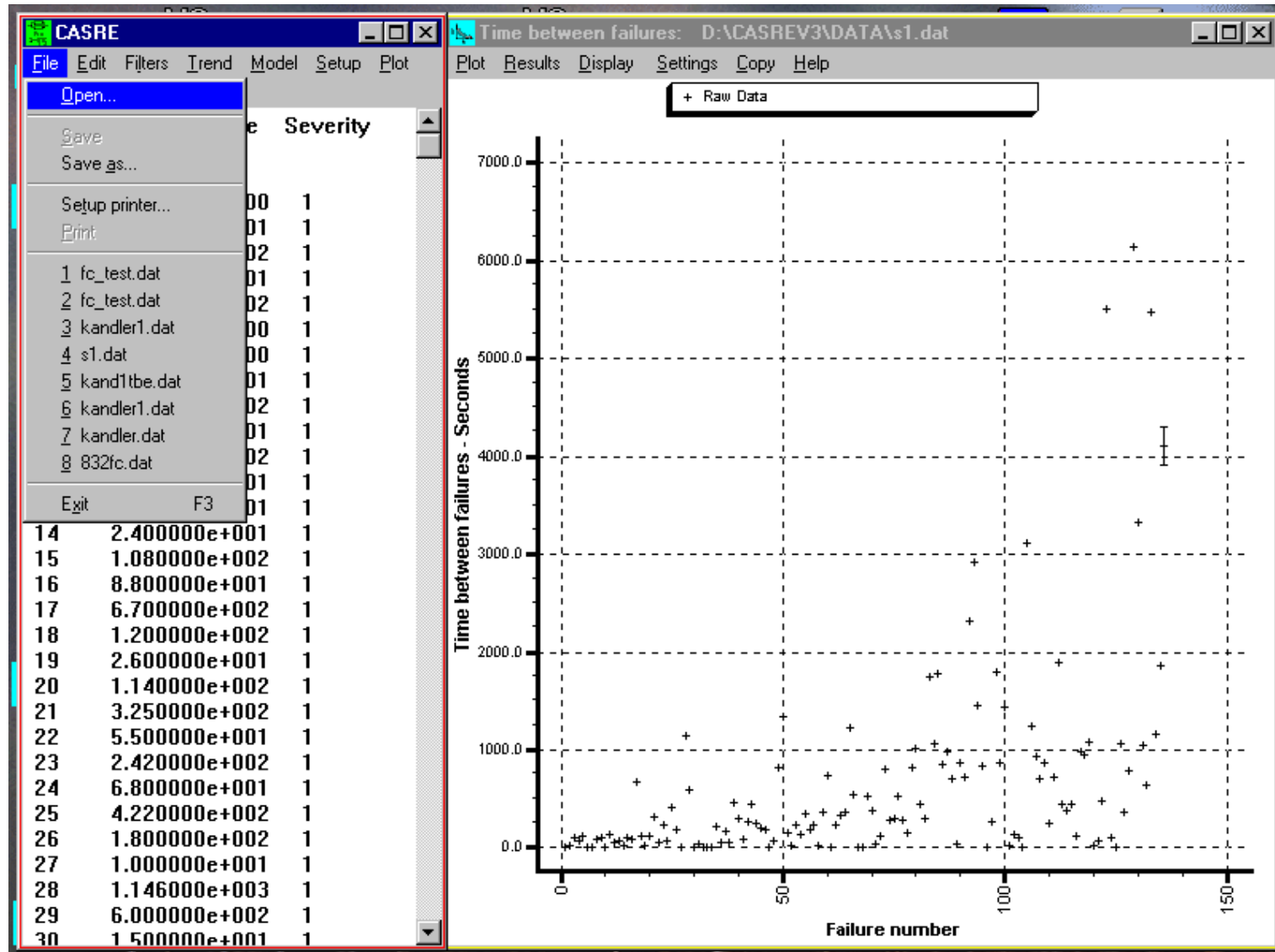
# CASRE Main Features

- Multiple Models (12)
- Model Application Scheme: Multiple Iterations
- Goodness-of-Fit Criteria - Chi-Square Test, KS Test
- Multiple Evaluation Criteria - Prequential Likelihood, Bias, Bias Trend, Model Noise
- Conversions between Failure-Counts Data and Time-Between-Failures Data
- Menu-Driven, High-Resolution Graphical User Interface
- Capability to Make Linear Combination Models

# CASRE High-Level Architecture

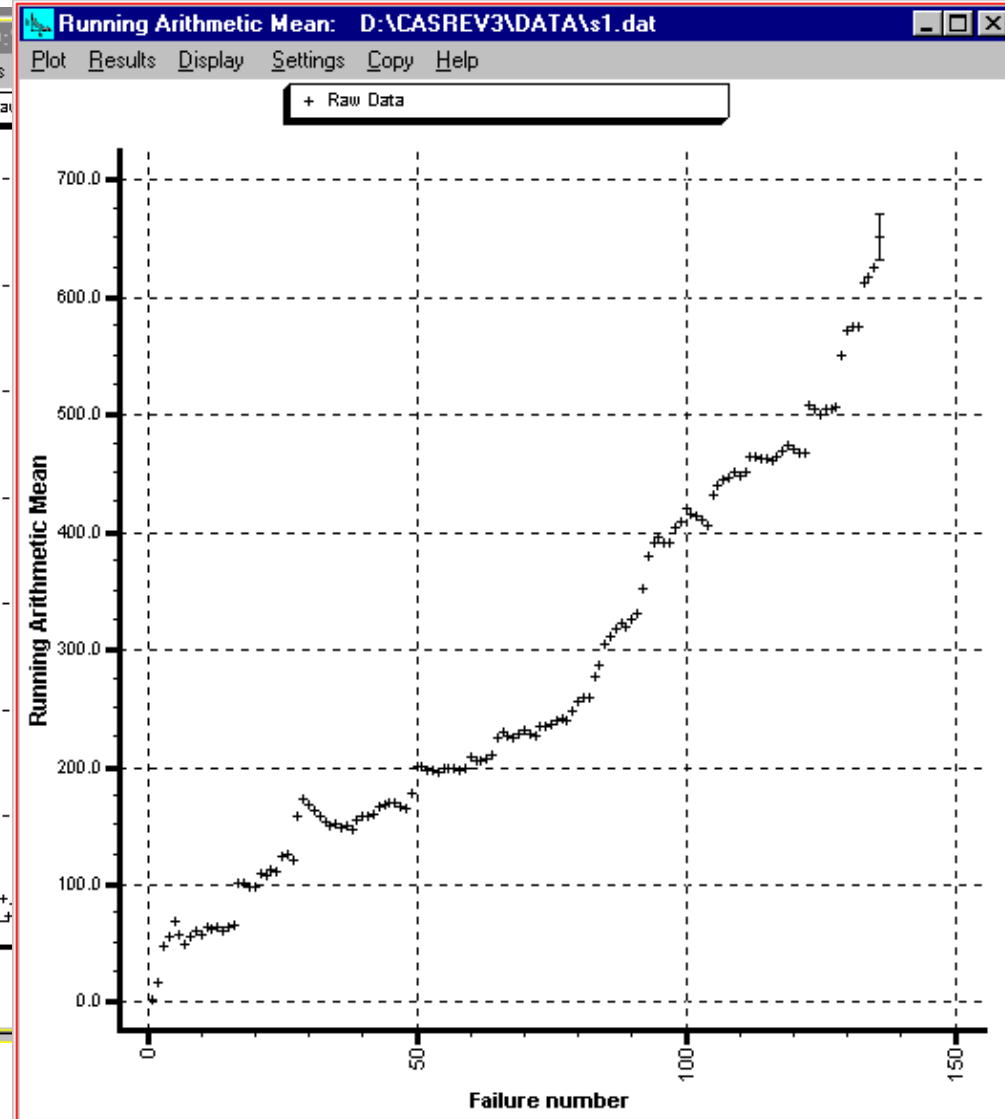
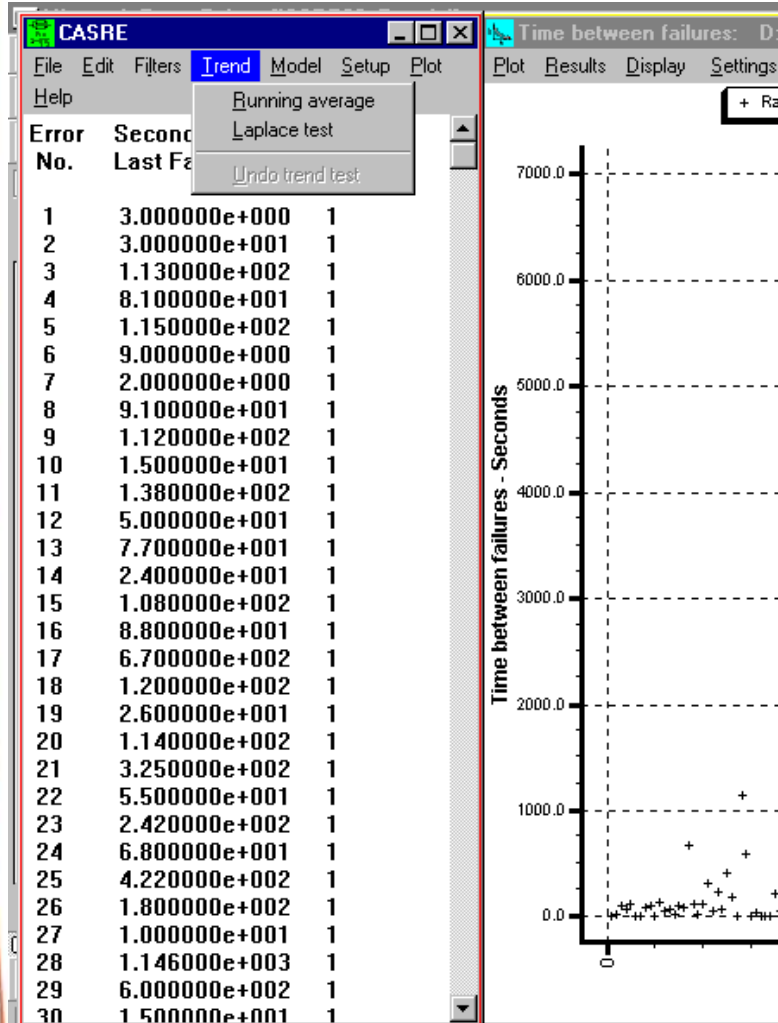


# CASRE Screen Shot

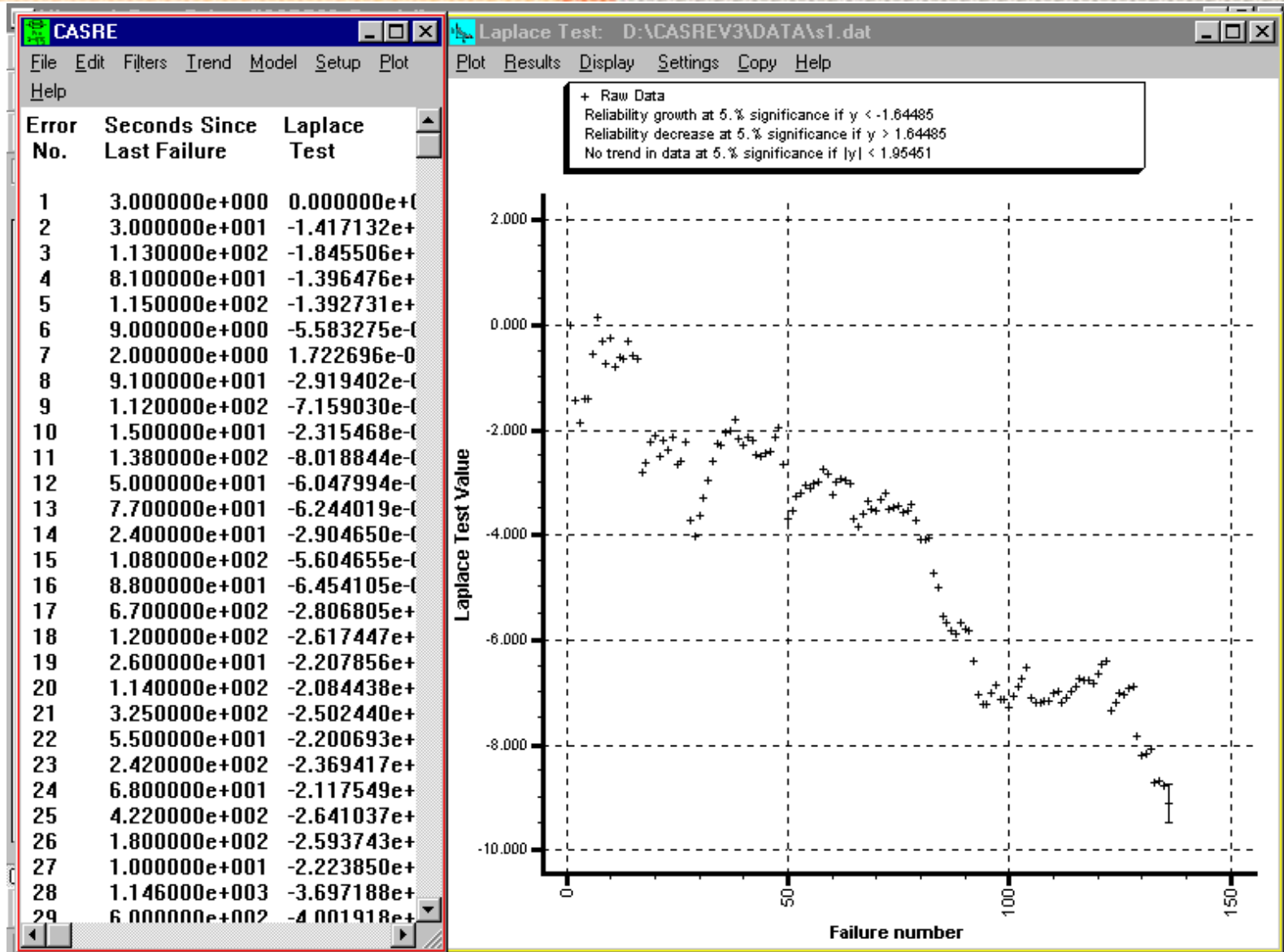




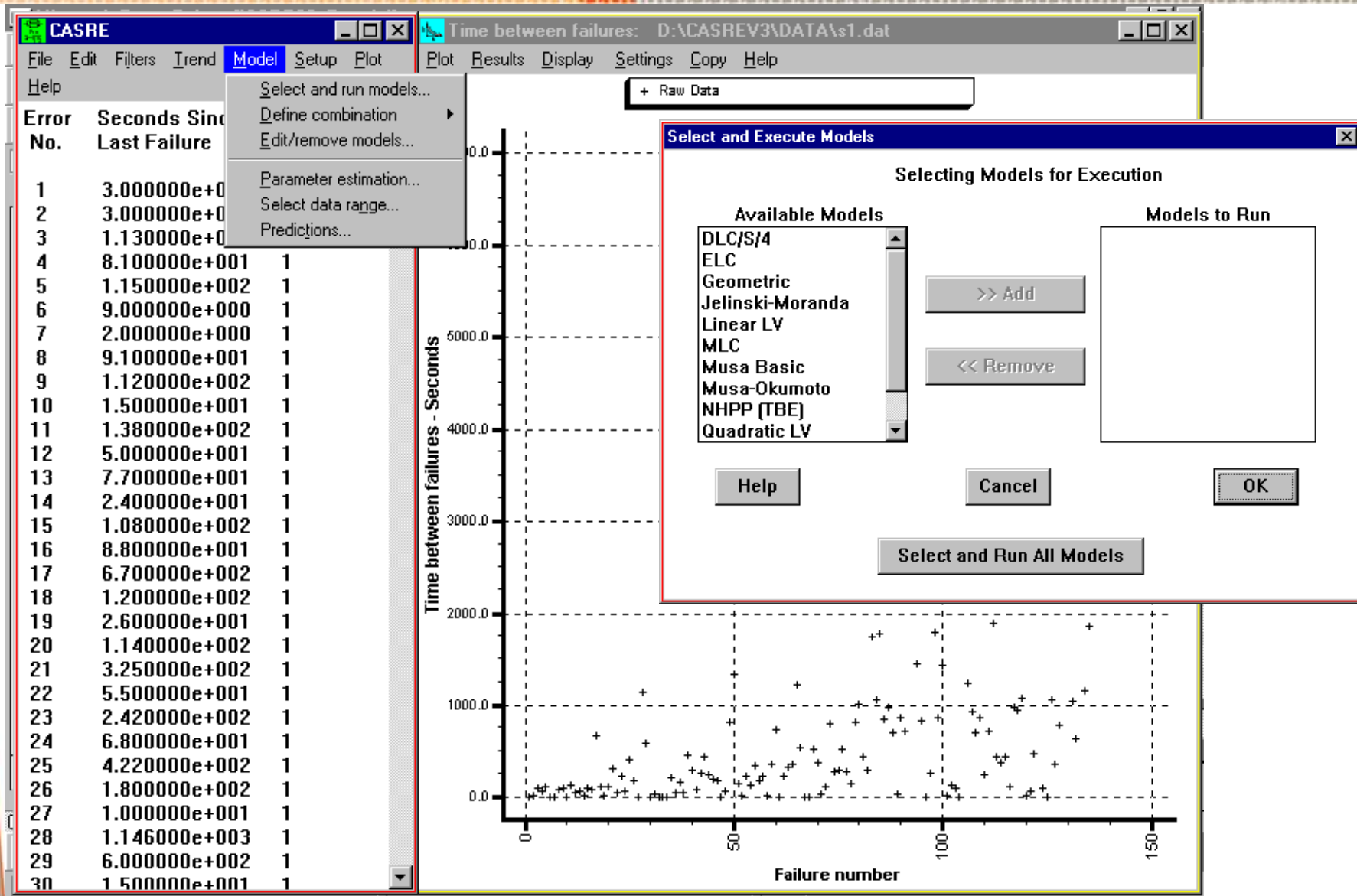
# CASRE – Running average Trend Test



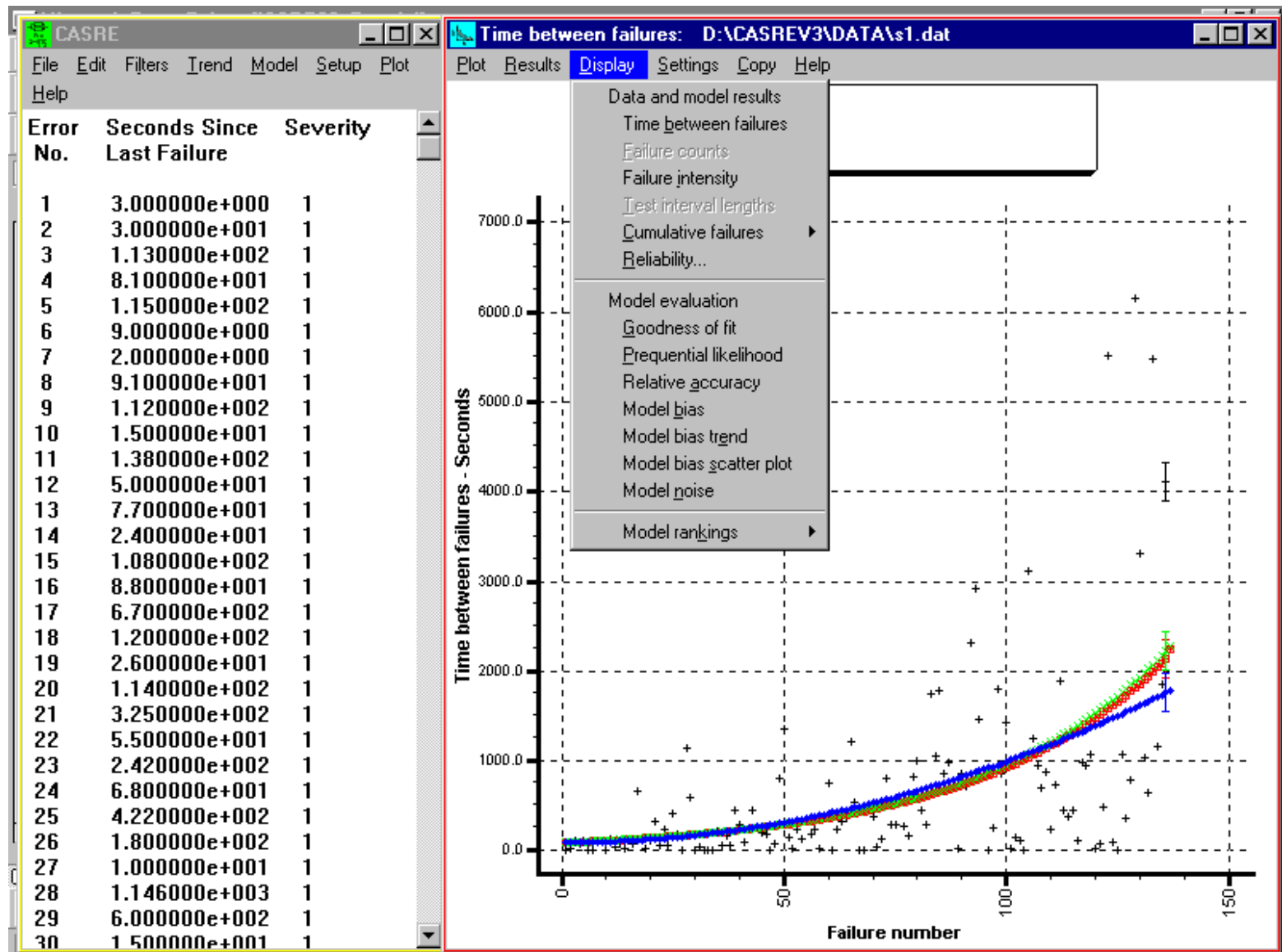
# CASRE – Laplace Test



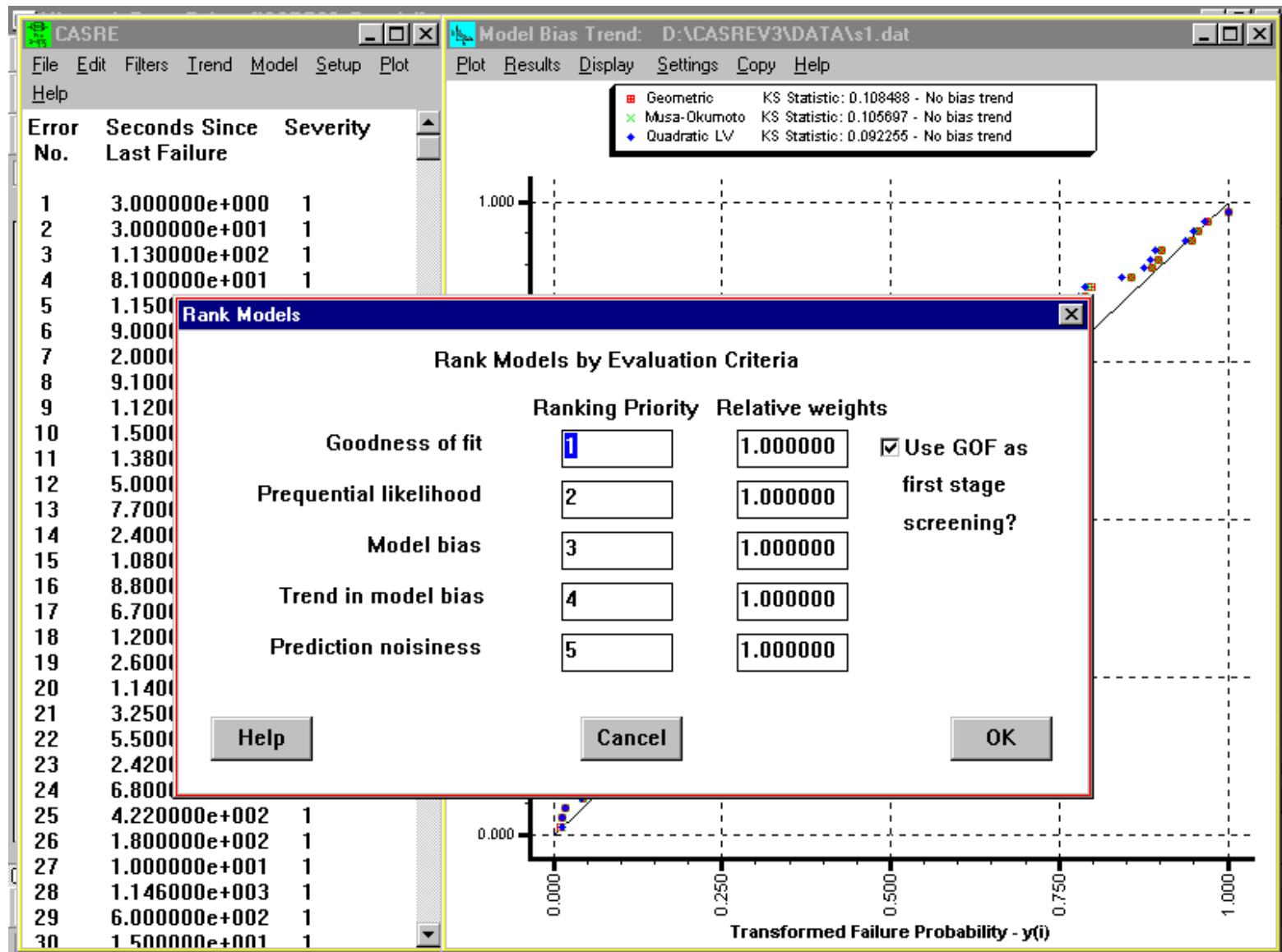
# CASRE – Select and Run Models



# CASRE – Display modelling results

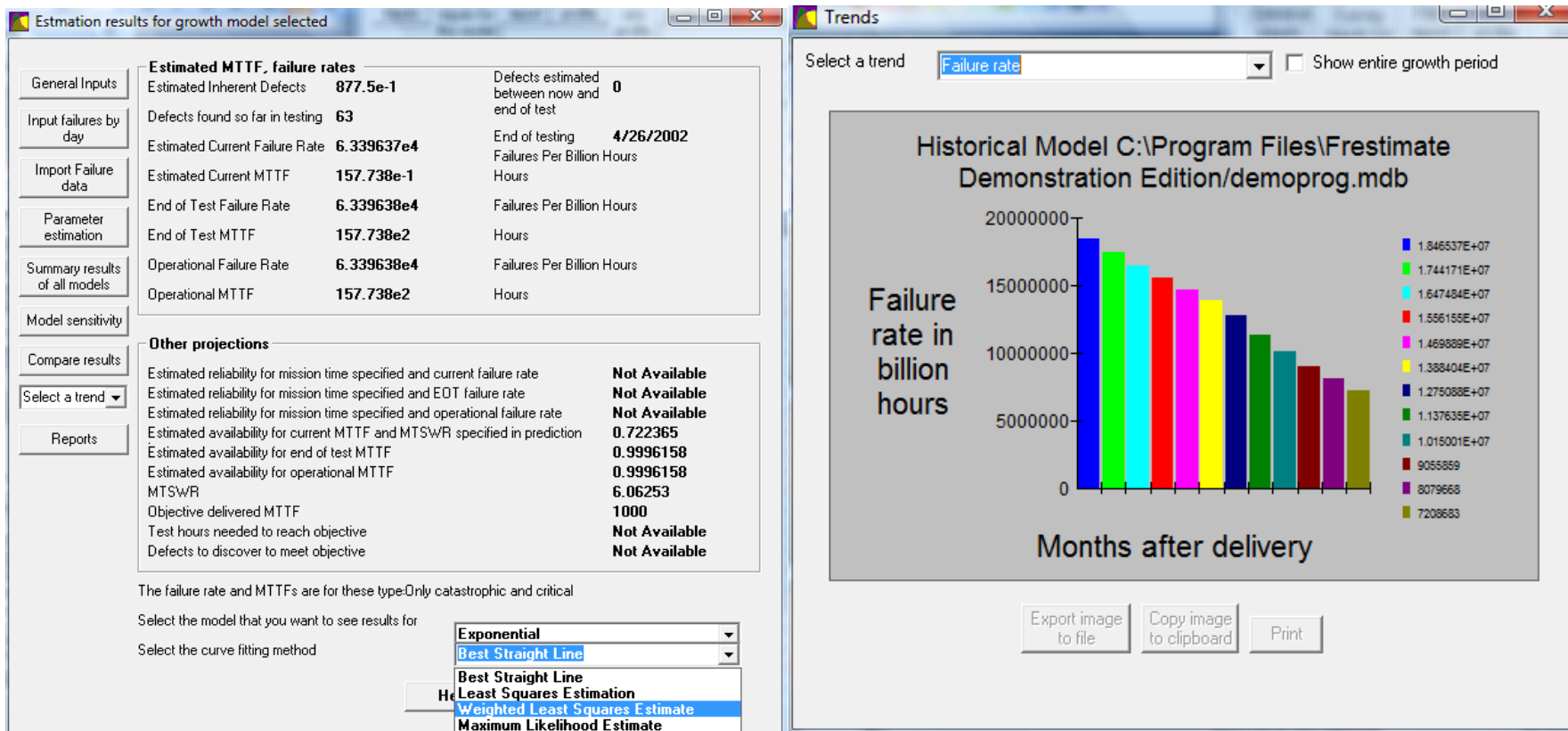


# CASRE – Ranking Models



# Frestimate Main Features

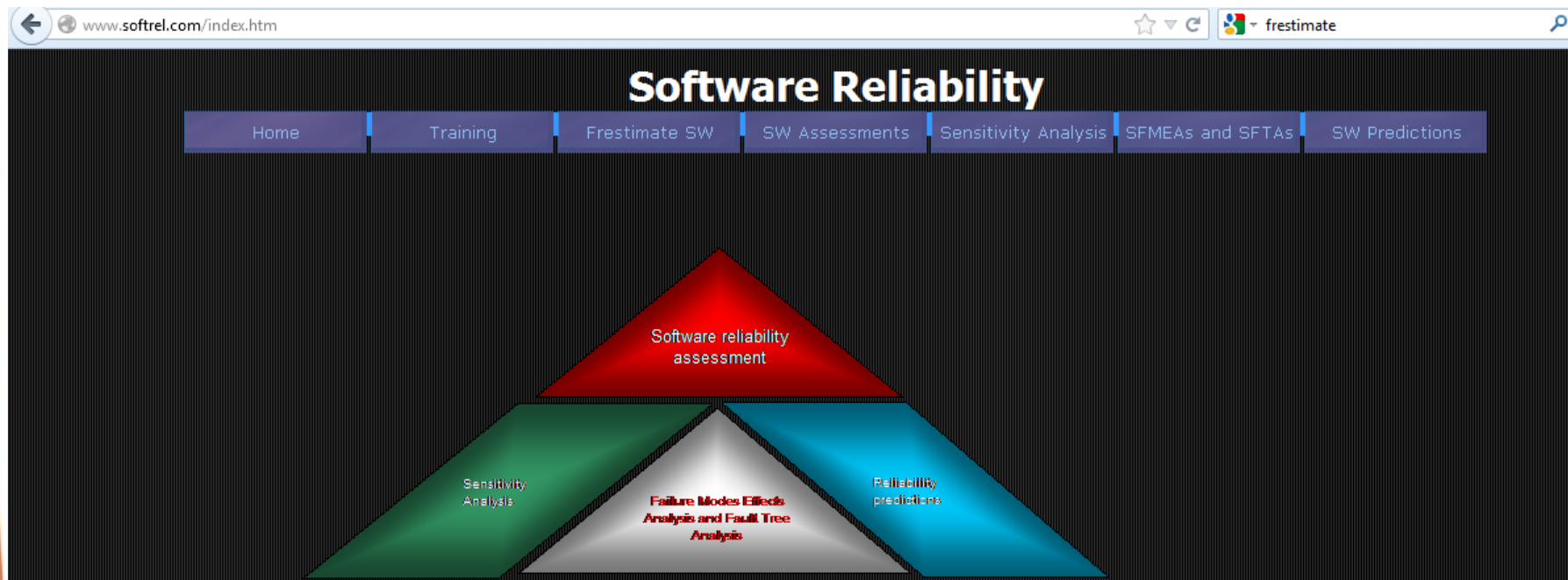
- **Frestimate** is a software reliability tool providing basic software prediction capabilities.





# FRESTIMATE - topics

- Frestimate – prediction/estimation models
- Frestimate versus CASRE/SMERFS



# Frestimate: prediction models (1a)

*Prediction models* - regardless of whether they are for software reliability or any other application - are developed by collecting trained data and observing relationships in that features and some outcome. In the case of software reliability the outcome is delivered defects normalized by code size.

*The features* vary from model to model and are generally related to development practices. Some models have only one feature. Some models have many features. The model is the mathematical expression that determines some outcome given some set of features.

# Frestimate: prediction models (1b)

- Predictors are used early in the development lifecycle to:
  - Determine whether the current capabilities/development practices are suitable for meeting a system reliability objective
  - Select the development practices that would allow the system reliability objective to be met
  - Determine whether vendor supplied software will meet a system objective

# Frestimate: prediction models (1c)

- Predictors are used early in the development lifecycle to:
  - Determine suitable quality and reliability objectives for the software
  - Determine staffing requirements for maintenance and testing
  - Predict the inherent number of defects in the software at the start and end of testing

# Frestimate: estimation (1)

- *Estimation models* - are models that project the future based on what has happened in the immediate past - on this project.
- *Estimators* do not use trained data like predictors, they use data collecting only from the project in which we are interested in measuring.

# Frestimate: estimation (2)

- *Estimators* have a variety of purposes including:
  - Projecting how many more hours of testing are needed to reach some reliability objective
  - Projecting how many more defects must be detected and then fixed to reach some reliability objective.
  - Validating a reliability prediction



# Frestimate / trained data

- Because the actual fielded defect density is known for the sample, it is called **trained data**.
- By exploring relationships between the development practices and observed defect density in trained data, we can develop mathematical models to predict defect density for organizations in which the development practices are known but the defect density is not known.

# Rome Laboratory Model (1)

- The Air Force's Rome Laboratory developed predictions of fault density that could be transformed into other reliability measures such as failure rates.

**Summary of Rome Labs Factors**

**Application factor**

Defect density baseline at start of testing 9

Defect density baseline at end of testing 0.5226

Conversion to KSLDC of assembler is applied only if you chose to use Rome Labs application factors 1 / 2.176 = .4596

**Factors applied to each of the above 2 Application factors**

|                     |          |           |                         |     |           |
|---------------------|----------|-----------|-------------------------|-----|-----------|
| Development Metric  | 1.035714 | D factor  | Anomaly Metric          | 1.1 | AM Factor |
| Complexity Metric   | 0.9      | SX Factor | Quality Reviews Metric  | 1   | QR Factor |
| Traceability Metric | 1.1      | ST Factor | Standards Review Metric | 1.5 | SR Factor |

**Predicted defect density using the Rome Labs factors**

Defect density at start of testing 41.98498

Defect density at end of testing 5.30493126648841

Predicted defect density is in terms of Object Oriented Language

Help Print

# Rome Laboratory Model (2)

A number of factors were selected:

**A - Application type** (e.g., real-time control systems, scientific, information management),

**D - Development environment** (methodology, tools, languages),

**Requirements and design representation metrics** (AM - anomaly management, ST - traceability, QR - incorporation of quality review results),

**Software implementation metrics** (SL - language type [assembly, high-order, object-oriented, etc.), SS - program size, SM - modularity, SU - extent of reuse, SX - complexity, SR - incorporation of standards review results into the software).

The initial fault density prediction is given by the product

$$\delta_0 = A * D * (AM * ST * QR) * (SL * SS * SM * SU * SX * SR).$$

# Rome Laboratory Model (3)

A prediction of *the initial failure rate* is made as  
[Musa]  $\lambda_0 = F * K * \delta_0 * \text{number of lines of source code}$   
 $= F * K * W_0$ , where:

- $W_0$  is called also ‘The number of inherent faults’
- $F$  is the linear execution frequency of the program
- $K$  is the fault expose ratio ( $1.4E-7 \leq K \leq 10.6E-7$ ).  
The *fault exposure ratio*,  $K$ , is an important factor that controls the per-fault hazard rate, and hence, the effectiveness of the testing of software.

# Rome Laboratory Model (4)

The initial failure rate can be expressed also by letting  $F=R/I$ , where :

- $R$  is the average instruction rate and
- $I$  is the number of object instructions in the program,

and then further rewriting  $I$  as  $I_s * Q_x$ , where

- $I_s$  is the number of source instructions and
- $Q_x$  is the code expansion ratio (the ratio of machine instructions to source instructions - an average value of 4 is indicated).

# Rome Laboratory Model @Frestimate

|  |  |
|--|--|
| <b>Select a model for predicting defects</b>     |  |
| Rome Labs model predicting end of test defects   |  |
| <b>Bounds on defect density</b>                  |  |
| SEI CMM model                                    |  |
| Industry model                                   |  |
| Shortcut model                                   |  |
| Fullscale model                                  |  |
| Rome Labs model predicting end of test defects   |  |
| Use components                                   |  |
| Rome Labs model predicting start of test defects |  |
| Closest database match                           |  |

|  |                          |   |                              |
|--|--------------------------|---|------------------------------|
| <b>Size inputs required to predict defects/failure rate/reliability/availability</b> |                          |   |                              |
| Select the units of measure for defect density                                       | KSLOC                    | Size wizard                                 |                              |
| Number of Components   | 13                       | Executable size                             | 23.44                        |
| Total KSLOC  | 224                      | Function Points                             | 23.44                        |
| Total effective KSLOC (EKSLOC)   | 224                      | Total effective function points             | 23.44                        |
| Critical EKSLOC  | 203                      | Critical Function Points                    | 23.44                        |
| Size error 5 %   |                          | Size error due to phase 50 %                |                              |
| Update Component Info  |                          | <input type="checkbox"/> Use component info |                              |
| Components List View   |                          | Components wizard                           |                              |
| Language   | Object Oriented Language | Code expansion ratio                        | 6                            |
| <input type="checkbox"/> Override default  |                          |   |                              |
| <b>Inputs required to predict reliability/availability</b>                           |                          |   |                              |
| Percent severe   | Object Oriented Language | Number months growth after testing ends     | 48                           |
| Average operating hours (duty cycle) per month                                       | 1600                     | Duty cycle wizard                           | Months to next major release |
| Fielded growth rate (Qdel) between defects and failures                              | 6                        | Growth period and Q Wizard                  |                              |
| Testing growth rate (Q0) between defects and failures                                | 9                        |   |                              |
| Application type   | Satellites               | Qdel confidence                             | 2.675                        |
| <b>Inputs required to predict reliability</b>  |                          |   |                              |
| MTSWR  | 6.0625                   | Duration                                    | 8                            |
| Inputs required to predict reliability   |                          |   |                              |
| Ratio of unique defects  | 5                        | Print                                       |                              |
| Ratio of interruptions that don't require a corrective action to those that do       |                          |   |                              |
| <b>Objectives - required for trends</b>  |                          |   |                              |
| MTTF at delivery   | 10                       | MTTF after growth period                    | 100                          |
| Wizard   |                          |   |                              |
| <b>Frestimate metrics</b>  |                          |   |                              |
| Import Size/Complexity   |                          |   |                              |



# FRESTIMATE 'against' CASRE and SMERFS

| Desired Feature  | CASRE, SMERFS  | Frestimate  |
|--|--|---|
| Supports both software reliability prediction models and estimation models               | No.<br><br>Supports only estimation models which are <b>used only after testing has started.</b> | Yes.<br><br>Has prediction models that <b>can be used before testing.</b><br><br>Also has estimation models in the WhenToStop module that can be used during testing. <b>You can use one software package from concept to delivery.</b> |
| Supports multi-parameter prediction models   | No.  | Yes. Multi-parameter models are generally more accurate then single parameter models. Frestimate also has 2 single parameter models in the event that you are short on available data.  |
| Compares your results with other projects that are similar in application, SEI CMM level | No.  | Yes.  |
| Has wizards to help you understand your inputs and outputs                               | No.  | Yes.  |
| Measures the three P's<br><br>Process, People and Product                                | No.  | Yes.  |

# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
- 7. Lessons Learned**
8. Background Literature

# 7. Lessons learned

- The *increase in software-based systems for safety functions* requires *systematic evaluation of software reliability*.
- Software reliability estimation is still *an unresolved issue* and existing *approaches* have *limitations* and assumptions that are not acceptable for safety applications.
- *Direct observation* of operational behaviour of the system (e.g. in test or simulation) is *not going to give assurance of ultra-high reliability*

# Tentative summary of the story so far:

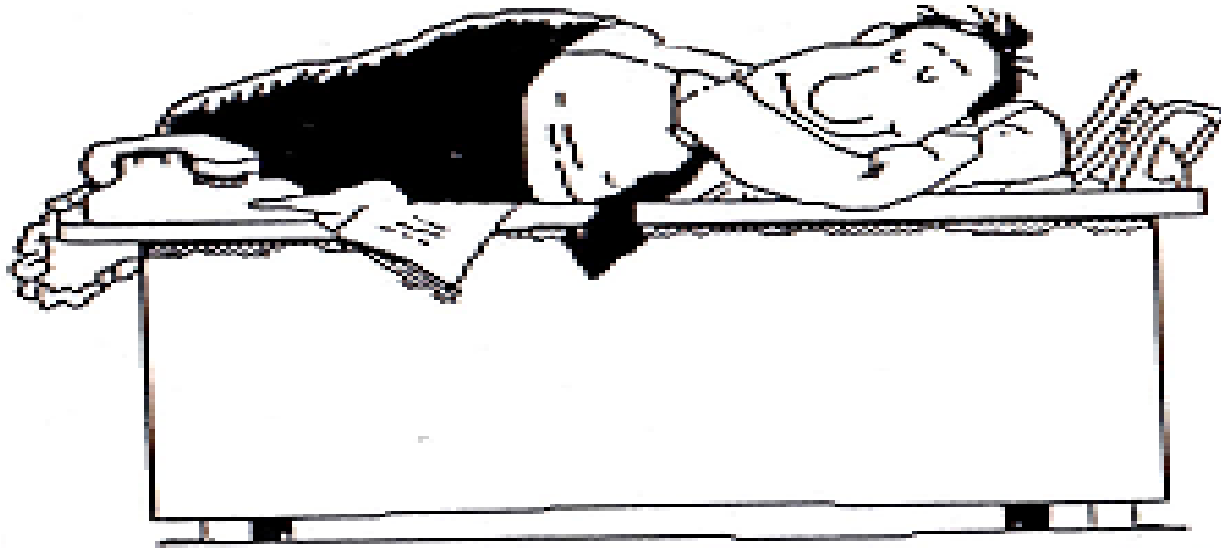
- *Lots of models* but no single “best buy”
- The *bad news*: some models almost universally bad, all models occasionally bad!
- The *good news*: some models OK sometimes
- Cannot select *a model a priori and trust it* to work (even if it worked well on a previous project, and you think the current project is ‘similar’)
- *Be eclectic*: try many models on your data and check for accuracy of predictions

It is **USUALLY** possible to predict software reliability with **REASONABLE** accuracy and have **SOME CONFIDENCE** you have done so.

# General conclusions

- *The bad news ...*
  - No perfect model
  - No way of selecting the best models *a priori*
  - All models sometimes inaccurate
- *... the good news ...*
  - Can analyse predictive accuracy dynamically
  - Recalibration often improve accuracy
  - Can usually obtain accurate reliability estimates and *know they are accurate*
- *... and the warning ...*
  - These techniques only work for modest reliability levels
  - They are essential no way assuring that ultrahigh reliability has been achieved

# Why Such Inactivity?



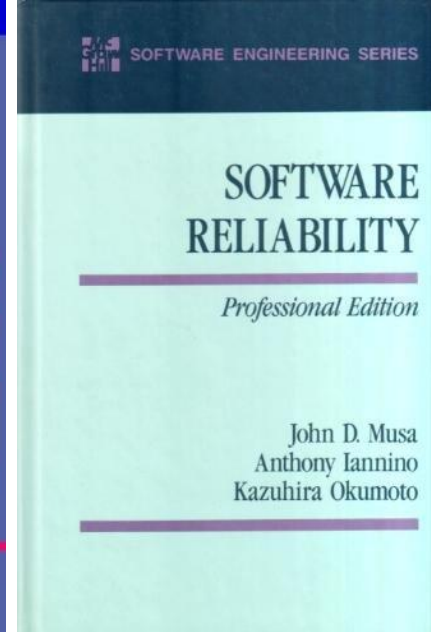
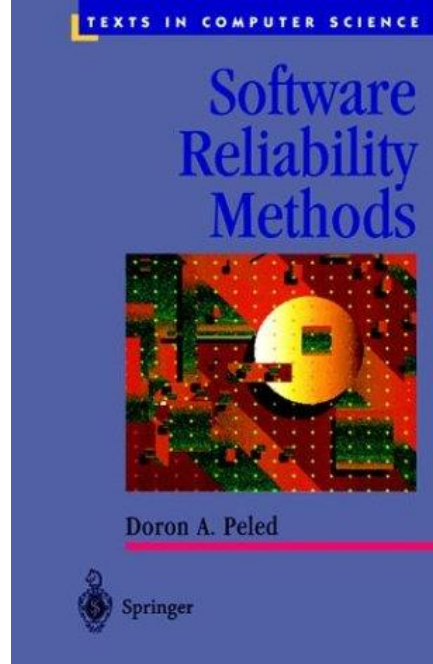
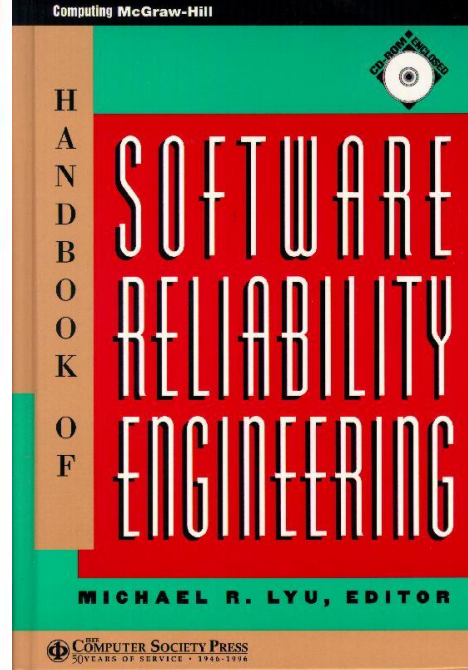
There are a number of reasons for this inactivity:

- Lack of awareness/training
- Disillusionment following “false starts” with immature measures and models. “All Models are Wrong - Some are Useful” (George E. P. Box)
- Too busy grappling with the current crisis to see long term



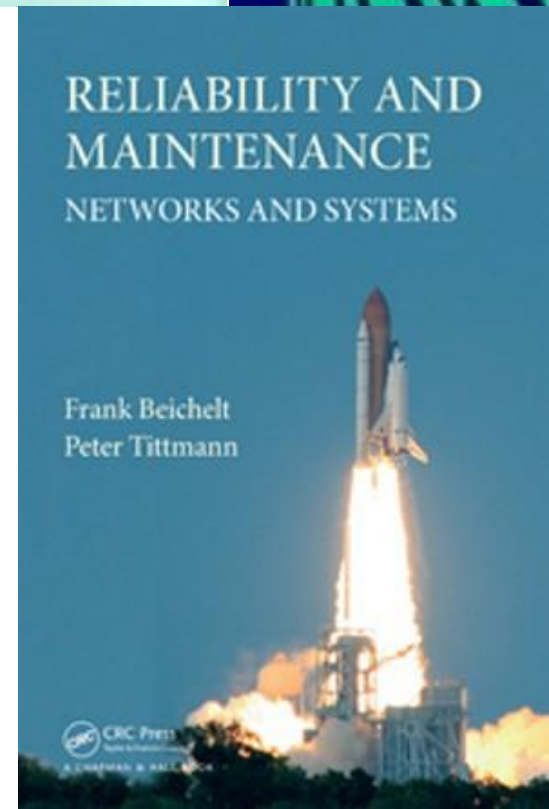
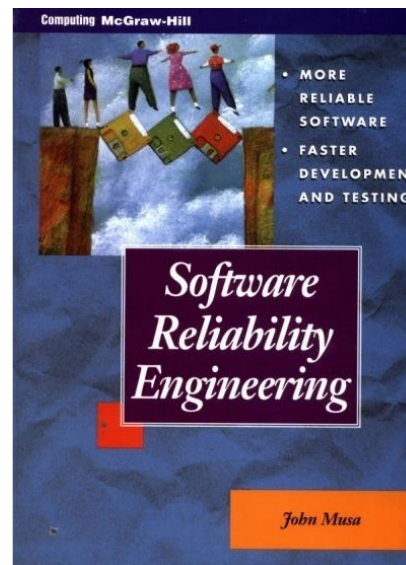
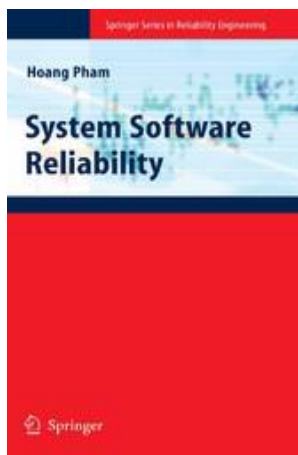
# Software Reliability Course - Agenda

1. Motivation
2. Introduction to Software Engineering
3. Measuring Software Reliability
4. Software Reliability Techniques and Tools
5. Experiences in Software Reliability
6. Software Reliability Engineering Practice
7. Lessons Learned
8. **Background Literature**



<http://www.cse.cuhk.edu.hk/~lyu/book/reliability/>

# References



# Appendix

Technical University of Denmark

Department of Informatics and Mathematical Modelling

# Software Reliability

## Appendix

Professor Florin POPENTIU VLADICESCU  
DTU, IMM, Building 305, Tel: + 45 4525 3353  
Fax: + 45 4588 2673

[popentiu@imm.dtu.dk](mailto:popentiu@imm.dtu.dk)  
<http://imm.dtu.dk/~popentiu>

**Cap. 1. Software Reliability Assessment**

**Cap. 2 NHPP Software Reliability Models**

**Cap. 3 Software Cost Models**

**Cap. 4 Some Statistical Approaches**



## Monitoring the development process

- Assuring that the plan is adhered to
- Assuring that the quality of the work is adequate



The time to think about monitoring is when the project is being planned. This enables review during the project to ensure that the plan is being adhered to, and that where divergences become necessary, they do not prejudice achievement of reliability targets.

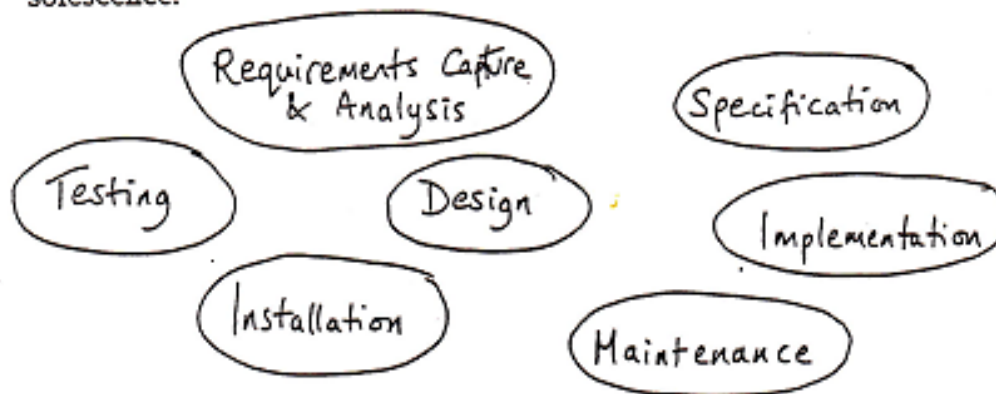
Similarly, the definition of adequacy should be agreed beforehand, as should ways by which the quality of the work will be assessed during development.

It is impossible to define precise relationships between quality achievements during development, and reliability at the end of development. Nevertheless, intermediate targets should be set which are acknowledged to be consistent with the targeted level of reliability.



# THE SOFTWARE LIFE CYCLE

The various phases that software undergoes from inception to obsolescence.



Each product passes through these phases. The duration, sequence, number of iterations, and exact effect of each stage may vary.

*Different software development process (or life-cycle) models:*

- Waterfall model
- Rapid prototyping
- Evolutionary development
- Component reuse
- Spiral model
- V model
- Formal transformations

# The need for Software Reliability

---

“If you don’t test your software,  
how do you know it works?”

## NHPP Software Reliability Models

---

“Every problem has in it the seeds of its solution.  
If you don't have any problems, you don't get any seeds.”

# **Software Reliability Models with Environmental Factors**

---

**“The whole of science is a refinement of everyday thinking”**

Albert Einstein (1879-1955)

Cap. 1. Software Reliability Assessment

**Cap. 2 NHPP Software Reliability Models**

Cap. 3 Software Cost Models

Cap. 4 Some Statistical Approaches

# NHPP Software Reliability Models

## *Notation*

- $m(t)$  expected number of errors detected by time  $t$   
("mean value function")
- $a(t)$  error content function, i.e., total number of error in the software including the initial and introduced errors at time  $t$
- $b(t)$  error detection rate per error at time  $t$
- $N(t)$  random variable representing the cumulative number of software errors detected by time  $t$
- $y(t)$  actual values of  $N(t)$  ( $y_i := y(t_i)$ )
- $S_j$  actual time at which the  $j^{\text{th}}$  error is detected
- $R(s/t)$  reliability during  $(t, t+s]$  given that the last error occurred at time  $t$



**EXAMPLE 5.1.** The data set in Table 5.1 was reported by Musa (1987) based on failure data from a real time command and control system, which represents the failures observed during system testing for 25 hours of CPU time. The delivered number of object instructions for this system was 21,700 and was developed by Bell Laboratories.

It should be noted that this data set belongs to the concave class, therefore, it seems reasonable to use the Goel-Okumoto NHPP model, to describe the failure process of the software system. From the failure data, the two unknown parameters,  $a$  and  $b$ , can be estimated as values for the two parameters:

$$\hat{a} = 142.3153$$

$$\hat{b} = 0.1246$$

where:

$\hat{a}$  is an estimate of the expected number of failures to be eventually detected

$\hat{b}$  is the number of faults detected per fault per unit time (hour). The estimated mean value function and the software reliability function are:

$$\hat{m}(t) = 142.3153(1 - e^{-0.1246t})$$

and

$$\hat{R}(x/t) = e^{-(142.3153)[e^{-(0.1246)t} - e^{-(0.1246)(t+x)}]}$$

respectively.

where:

$\hat{a}$  is an estimate of the expected number of failures to be eventually detected

$\hat{b}$  is the number of faults detected per fault per unit time (hour). The estimated mean value function and the software reliability function are:

$$\hat{m}(t) = 142.3153(1 - e^{-0.1246t})$$

and

$$\hat{R}(x/t) = e^{-(142.3153) \left[ e^{-(0.1246)t} - e^{-(0.1246)(t+x)} \right]}$$

respectively.

The above two functions can be used to determine when to release the software system or the additional testing effort required when the system is ready for release. Let us assume that failure data from only 16 hours of testing are available from the table 5.1 where a total of 122 failures have been observed:

The above two functions can be used to determine when to release the software system or the additional testing effort required when the system is ready for release. Let us assume that failure data from only 16 hours of testing are available from the table 5.1 where a total of 122 failures have been observed:

Table.5.1 Failure data in a one-hour interval and number of failures

| Hour | Number of failures | Cumulative failures |
|------|--------------------|---------------------|
| 1    | 27                 | 27                  |
| 2    | 16                 | 43                  |
| 3    | 11                 | 54                  |
| 4    | 10                 | 64                  |
| 5    | 11                 | 75                  |
| 6    | 7                  | 82                  |
| 7    | 2                  | 84                  |
| 8    | 5                  | 89                  |
| 9    | 3                  | 92                  |
| 10   | 1                  | 93                  |
| 11   | 4                  | 97                  |
| 12   | 7                  | 104                 |
| 13   | 2                  | 106                 |
| 14   | 5                  | 111                 |
| 15   | 5                  | 116                 |
| 16   | 6                  | 122                 |
| 17   | 0                  | 122                 |
| 18   | 5                  | 127                 |
| 19   | 1                  | 128                 |
| 20   | 1                  | 129                 |
| 21   | 2                  | 131                 |
| 22   | 1                  | 132                 |
| 23   | 2                  | 134                 |
| 24   | 1                  | 135                 |
| 25   | 1                  | 136                 |

Based on these data and using MLE method values for the two parameters are:

$$\hat{a} = 138.3779 \text{ and } \hat{b} = 0.1334$$

The estimated mean value function become:

$$\hat{m}(t) = 138.3779(1 - e^{-0.1334 \cdot t})$$

Thus the reliability of the system is :

$$\hat{R}(x/t) = e^{-(138.3779) \left[ e^{-(0.1334) \cdot t} - e^{-(0.1334) \cdot (t+x)} \right]}$$

**Table 5.2.** Software reliability performance measures.

| Test time<br>(T) | a        | b      | Remaining<br>errors | Reliability |          |
|------------------|----------|--------|---------------------|-------------|----------|
|                  |          |        |                     | $R(0.1/T)$  | $R(1/T)$ |
| 16               | 138.3779 | 0.1333 | 16.3780             | 0.8049      | 0.1294   |
| 17               | 133.7050 | 0.1432 | 11.7050             | 0.8466      | 0.2096   |
| 18               | 141.2543 | 0.1274 | 14.2544             | 0.8349      | 0.1817   |
| 19               | 139.7190 | 0.1304 | 11.7190             | 0.8591      | 0.2386   |
| 20               | 138.8495 | 0.1323 | 9.8495              | 0.8786      | 0.2951   |
| 21               | 140.3408 | 0.1290 | 9.3408              | 0.8871      | 0.3228   |
| 22               | 140.1002 | 0.1296 | 8.1002              | 0.9010      | 0.3737   |
| 23               | 141.9104 | 0.1255 | 7.9104              | 0.9060      | 0.3933   |
| 24               | 142.0264 | 0.1252 | 7.0265              | 0.9162      | 0.4372   |
| 25               | 142.3153 | 0.1246 | 6.3154              | 0.9248      | 0.4772   |



# NHPP S-Shaped Model

In the NHPP S-shaped model, the software reliability growth curve is an S-shaped curve which means that the curve crosses the exponential curve from below and the crossing occurs once and only once. The detection rate of faults, where the error detection rate changes with time, become the greatest at a certain time after testing begins, after which it decreases exponentially. In other words, some faults are covered by other faults at the beginning of the testing phase, and before these faults are actually removed, the covered faults remain undetected. Yamada (1984) also determined that the software testing process usually involves a learning process where testers become familiar with the software products, environments, and software specifications. Several S-shaped models (Yamada, 1984; Pham, 1997) such as delayed S-shaped, infection S-shaped, etc., will also be discussed in this section.

The NHPP S-shape model is based on the following assumptions:

1. The error detection rate differs among faults.
2. Each time a software failure occurs, the software error which caused it is immediately removed, and no new errors are introduced.

This can be shown as the following differential equations:



$$\frac{\partial m(t)}{\partial t} = b(t)[a - m(t)], \quad (1)$$

where

$a$  = expected total number of faults that exist in the software before testing

$b(t)$  = failure detection rate, also called the failure intensity of a fault

$m(t)$  = expected number of failures detected at time  $t$ .

The above differential equation can be easily solved and is given by

$$m(t) = a \left[ 1 - e^{-\int_0^t b(u) du} \right]. \quad (2)$$

# NHPP Inflection S-shape Model

The inflection S-shaped model (Ohba, 1984) is based on the dependency of faults by postulating the following assumptions:

1. Some of the faults are not detectable before some other faults are removed.
2. The probability of failure detection at any time is proportional to the current number of detectable faults in the software.
3. Failure rate at each detectable fault is constant and identical.
4. The isolated faults can be entirely removed.

Assume

$$b(t) = \frac{b}{1 + \beta e^{-bt}} . \quad (1)$$

where the parameters  $b$  and  $\beta$  represent the failure-detection rate and the inflection factor, respectively. From Eq. (5.15), the mean value function is given by:

$$m(t) = \frac{a}{1 + \beta e^{-bt}} (1 - e^{-bt}) . \quad (2)$$

This model is called the inflection S-shaped NHPP model (Ohba, 1984).

The failure intensity function is given by:

$$\lambda(t) = \frac{ab(1+\beta)e^{-bt}}{(1+\beta e^{-bt})^2} \quad (3)$$

We then obtain the expected number of remaining errors at time  $t$

$$m(\infty) - m(t) = \frac{a(1+\beta)e^{-bt}}{(1+\beta e^{-bt})} \quad (4)$$

For type 1 data, the estimate of parameters  $a$  and  $b$  for specified  $\beta$  using the MLE method can be obtained by solving the following equations simultaneously:

$$a = \frac{y_n(1+\beta e^{-bt_n})}{(1-e^{-bt_n})} \quad (5)$$

and

$$\begin{aligned} \sum_{i=1}^n (y_i - y_{i-1}) & \left( \frac{t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}}}{e^{-bt_{i-1}} - e^{-bt_i}} + \frac{\beta t_i e^{-bt_i}}{1 + \beta e^{-bt_i}} + \frac{\beta t_{i-1} e^{-bt_{i-1}}}{1 + \beta e^{-bt_{i-1}}} \right) \\ &= \frac{y_n t_n e^{-bt_n} (1 - \beta + 2\beta e^{-bt_n})}{(1 - e^{-bt_n})(1 + \beta e^{-bt_n})} \end{aligned} \quad (6)$$

Similarly, for type 2 data, the estimate of parameters  $a$  and  $b$  for specified  $\beta$  using the MLE method can be obtained by solving the following equations:

$$a = \frac{n(1 + \beta e^{-bs_n})}{1 - e^{-bs_n}} \quad (7)$$

and

$$\frac{ns_n e^{-bs_n} (1 + \beta)}{(1 - e^{-bs_n})(1 + \beta e^{-bs_n})} = \frac{n}{\beta} - \sum_{i=1}^n s_i + 2 \sum_{i=1}^n \frac{\beta s_i e^{-bs_i}}{1 + \beta e^{-bs_i}} \quad (8)$$

# NHPP Delayed S-Shape Model

We now discuss a stochastic model for a software error detection process based on NHPP in which the growth curve of the number of detected software errors for the observed failure data is S-shaped, called delayed S-shaped NHPP model (Yamada, 1984). The software error detection process described by an S-shaped curve can be characterized as a learning process in which test-team members become familiar with the test environment, testing tools, or project requirements, i.e. their test skills gradually improve. The delayed S-shape model is based on the following assumptions:

1. All faults in a program are mutually independent from the failure detection point of view.
2. The probability of failure detection at any time is proportional to the current number of faults in a software.
3. The proportionality of failure detection is constant.
4. The initial error content of the software is a random variable.
5. A software system is subject to failures at random times caused by errors present in the system.
6. The time between failures  $(i-1)^{\text{th}}$  and  $i^{\text{th}}$  depends on the time to the  $(i-1)^{\text{th}}$  failure.
7. Each time a failure occurs, the error which caused it is immediately removed and no other errors are introduced.



Assume

$$b(t) = \frac{b^2 t}{bt + 1} \quad (1)$$

where  $b$  is the error detection rate per error in the steady-state. The mean value function can be obtained as:

$$m(t) = a[1 - (1 + bt)e^{-bt}] \quad (2)$$

which shows an S-shaped curve. This model is called the delayed S-shape NHPP model for such an error detection process, in which the observed growth curve of the cumulative number of detected errors is S-shaped (Yamada, 1983). The corresponding failure intensity function is:

$$\lambda(t) = ab^2 te^{-bt} \quad (3)$$

The reliability growth of the software system is:

$$R(s|t) = e^{-[m(t+s) - m(t)]} = e^{-a[(1+bt)e^{-bt} - (1+b(t+s))e^{-b(t+s)}]} \quad (4)$$

The expected number of errors remaining in the system at time  $t$  is given by:

$$n(t) = m(\infty) - m(t) = a(1 + bt)e^{-bt} \quad (5)$$



For type 1 data, the estimate of parameters a and b using the MLE method can be obtained by solving the following equations simultaneously:

$$a = \frac{y_n}{[1 - (1 + bt_n e^{-bt_n})]} \quad (6)$$

and

$$\frac{y_n t_n^2 e^{-bt_n}}{[1 - (1 + bt_n e^{-bt_n})]} = \sum_{i=1}^n \frac{(y_i - y_{i-1})(t_i^2 e^{-bt_i} - t_{i-1}^2 e^{-bt_{i-1}})}{[(1 + bt_{i-1})e^{-bt_{i-1}} - (1 + bt_i)e^{-bt_i}]} \quad (7)$$

Similarly, for type 2 data, the estimate of parameters a and b for specified  $\beta$  using the MLE method can be obtained by solving the following equations:

$$a = \frac{n}{[1 - (1 + bs_n e^{-bs_n})]} \quad (8)$$

and

$$\frac{2n}{b} = \sum_{i=1}^n s_i + \frac{nbs_n^2 e^{-bs_n}}{[1 - (1 + bs_n e^{-bs_n})]} \quad (9)$$

**EXAMPLE 5.2.** The small on-line data entry software package test data, available since 1980 in Japan (Ohba, 1984), is shown in Table 5.3 (data set #5). The size of the software has approximately 40,000 LOC. The testing time was measured on the basis of the number of shifts spent running test cases and analyzing the results. The pairs of the observation time and the cumulative number of faults detected are presented in Table 5.3.

**Table 5.3.** On-line data entry software package test data (data set #5)

| Time of observation | Cumulative number of failures |
|---------------------|-------------------------------|
| 1                   | 2                             |
| 2                   | 3                             |
| 3                   | 4                             |
| 4                   | 5                             |
| 5                   | 7                             |
| 6                   | 9                             |
| 7                   | 11                            |
| 8                   | 12                            |
| 9                   | 19                            |
| 10                  | 21                            |
| 11                  | 22                            |
| 12                  | 24                            |
| 13                  | 26                            |
| 14                  | 30                            |
| 15                  | 31                            |
| 16                  | 37                            |
| 17                  | 38                            |
| 18                  | 41                            |
| 19                  | 42                            |
| 20                  | 45                            |
| 21                  | 46                            |

The MLEs of the unknown parameters  $a$  and  $b$  for the delayed S-shaped NHPP model are

$$\begin{aligned}a &= 71.725 \\ b &= 0.104.\end{aligned}$$

The estimated mean value function  $m(t)$  is

$$m(t) = (71.725)[1 - (1 + 0.104t)e^{-0.104t}].$$

Figure 5.1 shows the analysis results using the delayed S-shaped NHPP model. We can see that the model fits the observed failure data well in this set.

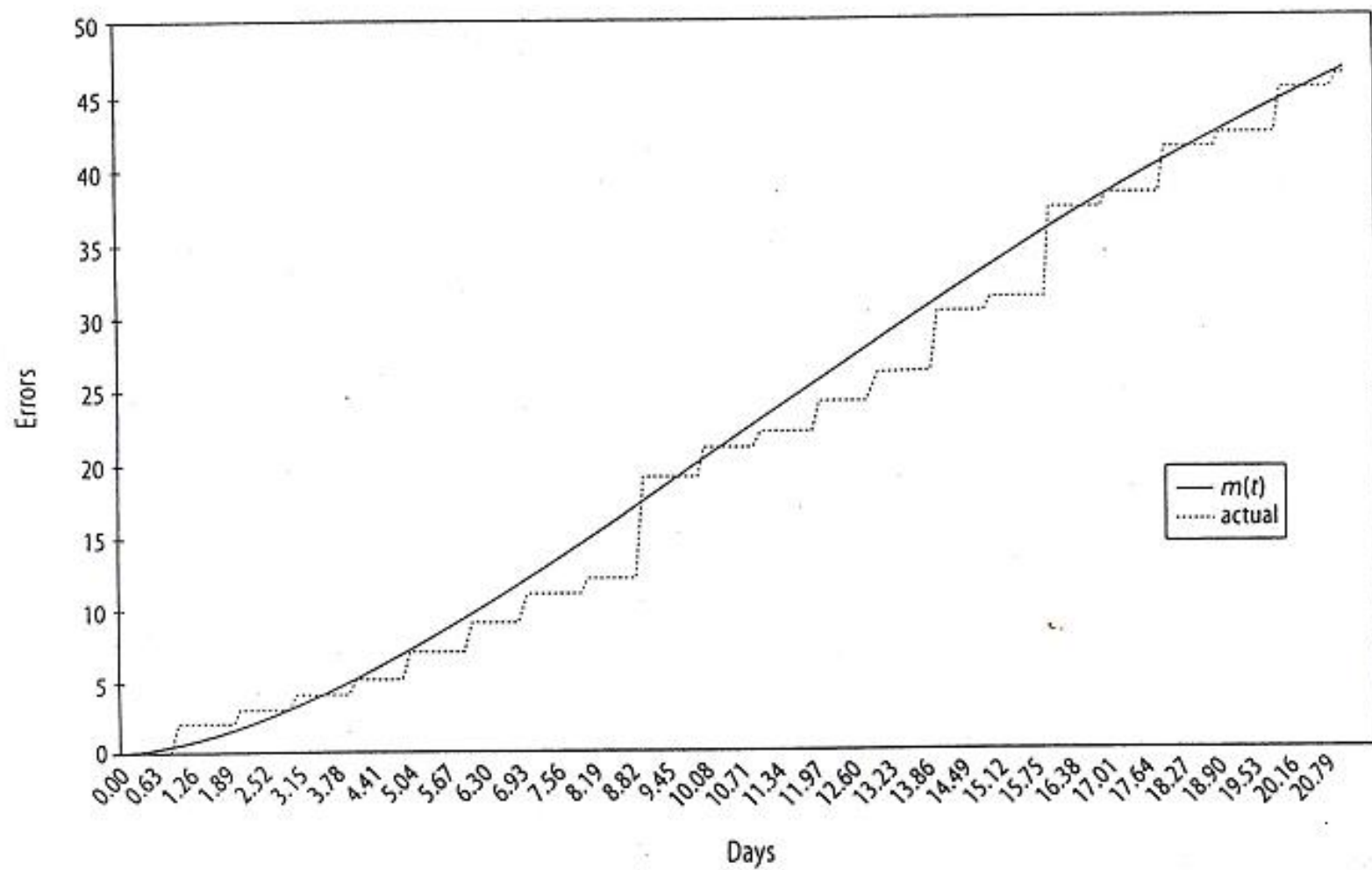


FIG. 5.1. Mean value function versus actual error data.

**Cap. 1. Software Reliability Assessment**

**Cap. 2 NHPP Software Reliability Models**

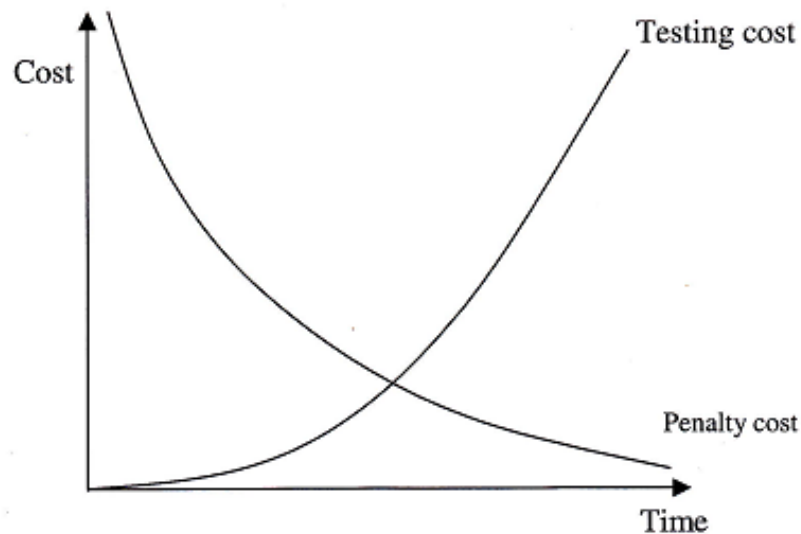
**Cap. 3 Software Cost Models**

**Cap. 4 Some Statistical Approaches**

# Software cost models

In defining important software cost factors, a cost model should help software and managers answer the following questions:

1. How should resources be scheduled to ensure the on-time and efficient delivery of a software product?
2. Is the software product sufficiently reliable for release (e.g. have we done enough testing?)
3. What information does a manager or software developer need to determine the release of software from current software testing activities?





Cost of fixing an error and the  
probability of fixing an error incorrectly

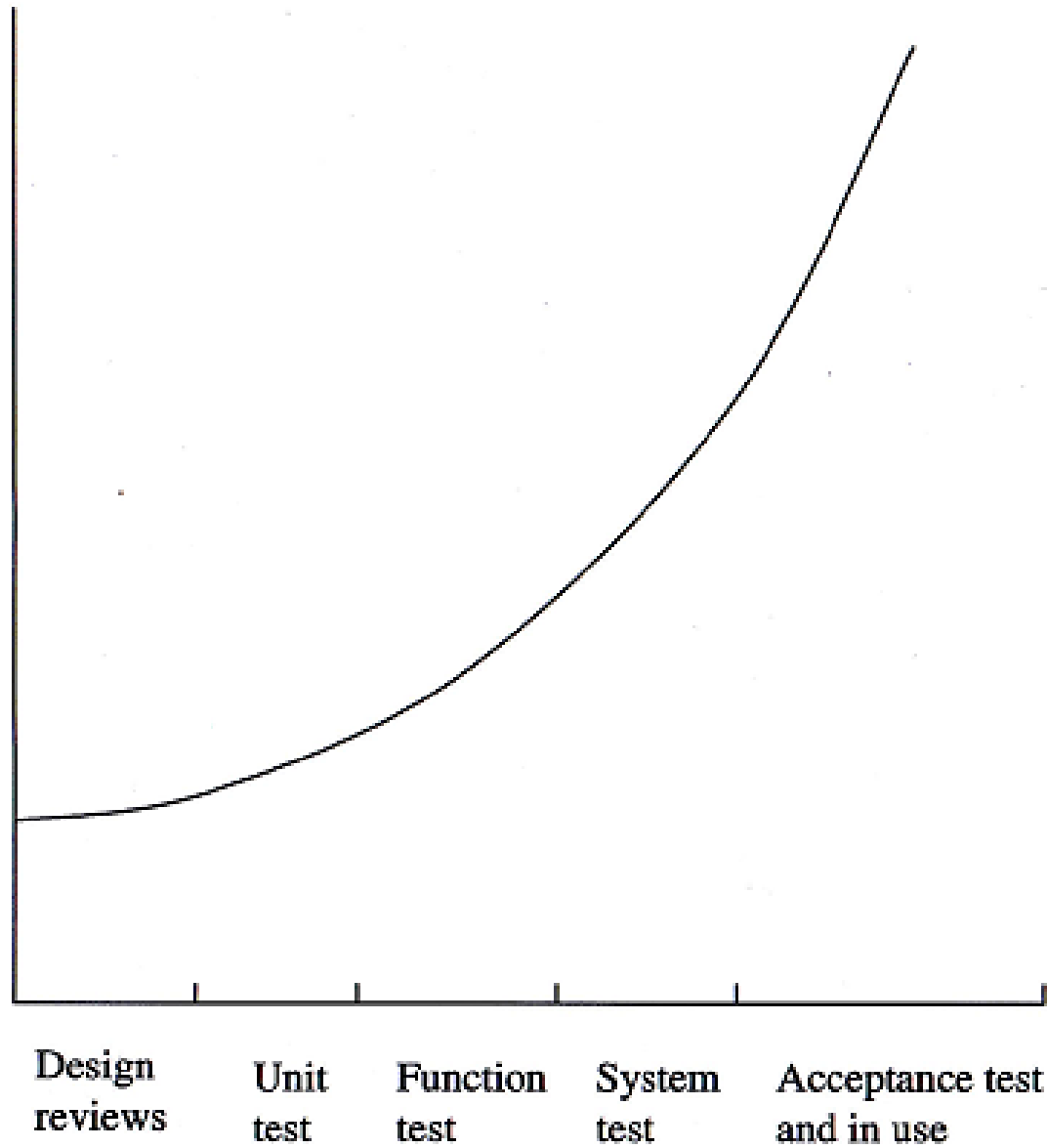


Fig. 3.10 Relationships between error correction and time.

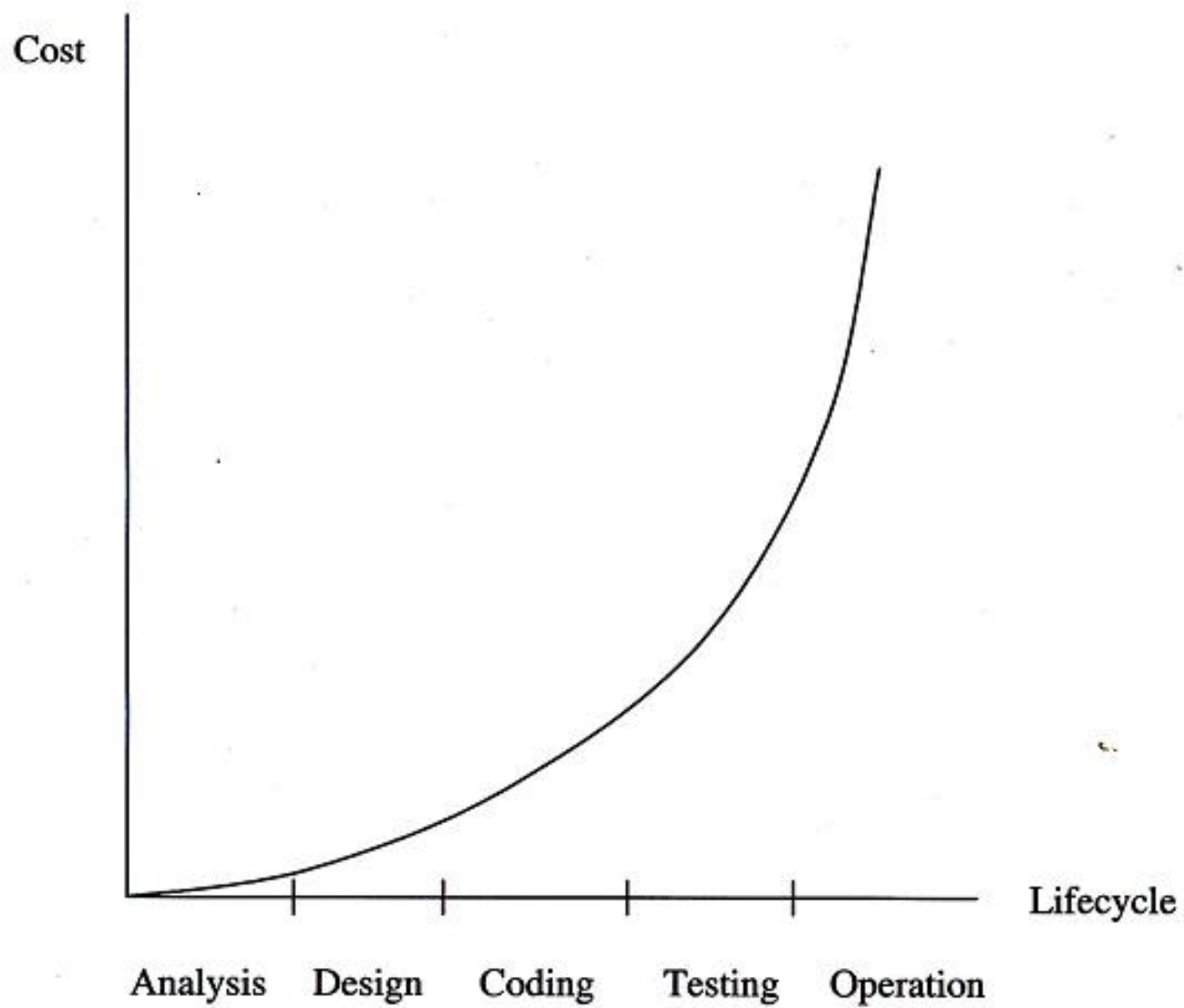


Fig. 3.15 Software cost versus the development phase.

TABLE 3.2 Software error introduction and discovery.

| Lifecycle phase    | Errors introduced<br>(%) | Errors detected<br>(%) |
|--------------------|--------------------------|------------------------|
| Analysis           | 55                       | 18                     |
| Design             | 30                       | 10                     |
| Coding and testing | 10                       | 50                     |
| Operations         | 5                        | 22                     |

# A Software Cost Model with Risk Factor

The expected software system cost,  $E(T)$ , is defined as: (1) the cost to perform testing; (2) the cost incurred in removing errors during the testing phase; and (3) a risk cost due to software failure.

- a. The cost to perform testing is given by

$$E_1(T) = C_1 T.$$

- b. The expected total time to remove all  $N(T)$  errors is

$$E\left[\sum_{i=1}^{N(T)} Y_i\right] = E[N(T)]E[Y_i] = m(T)\mu_y$$

Hence the expected cost to remove all errors detected by time  $T$  can be expressed as

$$E_2(T) = C_2 E\left[\sum_{i=1}^{N(T)} Y_i\right] = C_2 m(T)\mu_y$$

- c. The risk cost due to software failure after releasing the software is

$$E_3(T) = C_3 [1 - R(x|T)]$$

where  $C_3$  is the cost due to software failure.

The following notations and basic assumptions are applied throughout this chapter.

*Notation*

|              |   |
|--------------|---|
| $m(T)$       | expected number of errors to be detected by time $T$                  |
| $a$          | total number of software errors to be eventually detected             |
| $b$          | exponential index   |
| $\lambda(T)$ | fault detection rate per unit time or intensity function              |
| $x$          | mission time  |
| $R(x/T)$     | reliability function of software by time $T$ for a mission time $x$   |
| $T$          | software release time   |
| $C1$         | software test cost per unit time                                      |
| $C2$         | cost of removing each error per unit time during testing              |
| $E(T)$       | expected total cost of a software system by time $T$                  |
| $Y$          | time to remove an error during testing phase                          |
| $\mu_y$      | expected time to remove an error during testing phase which is $E(Y)$ |

*General assumptions:*

1. The cost to perform testing is proportional to the testing time.
2. The cost to remove errors during the testing phase is proportional to the total time of removing all errors detected by the end of the testing phase.
3. There is a risk cost related to the reliability at each release time point.

4. The time to remove each error during testing follows a truncated exponential distribution.
5. Without loss of generality, the Goel-Okumoto NHPP model will be used as a reliability function.

Let  $Y$  be a random variable of time to remove an error. Based on assumption (4), the probability density distribution of  $Y$  is given by

$$s(y) = \frac{\lambda e^{-\lambda y}}{\int_0^{T_0} \lambda e^{-\lambda z} dz}, \quad \text{for } 0 \leq y \leq T_0 \quad (1)$$

where  $T_0$  is the maximum time to remove an error. The expected time to remove each error is

$$\begin{aligned} \mu_y = E(Y) &= \int_0^{T_0} y s(y) dy \\ &= \int_0^{T_0} \frac{y \lambda e^{-\lambda y}}{\int_0^{T_0} \lambda e^{-\lambda z} dz} dy \end{aligned} \quad (2)$$

After simplifications we obtain

$$\mu_y = \frac{1 - (\lambda T_0 + 1)e^{-\lambda T_0}}{\lambda(1 - e^{-\lambda T_0})} \quad (3)$$



Therefore, the expected total software cost can be expressed (Zhang, 1998) as:

$$E(T) = C_1 T + C_1 \bar{F} m(T) \mu_y + C_3 [1 - R(x | T)] \quad (4)$$

The mean value function  $m(T)$  is

$$m(T) = a(1 - e^{-bT}) \quad (5)$$

The error detection rate function is

$$\lambda(T) = abe^{-bT} \quad (6)$$

and the reliability of the software is

$$\begin{aligned} R(x | T) &= e^{-[m(T+x) - m(T)]} \\ &= e^{-a[e^{-bT} - e^{-b(T+x)}]} \end{aligned} \quad (7)$$

**Example 6.1** Assuming a software failure data is given in Table 5.1, the parameters of the Goel-Okumoto model using MLE is given by:

$$\hat{a} = 142.32, \hat{b} = 0.1246$$

The mean value function becomes

$$m(T) = a(1 - e^{-bt}) = 142.32(1 - e^{-0.1246T}).$$

Given  $C_1 = \$25$ ,  $C_2 = \$200$ ,  $C_3 = \$7,000$ ,  $\mu_y = 0.1$ , and  $x = 0.05$ . The results are shown in Table 6.1.

The optimal release time in this case is  $T^* = 21.5$  hours and the corresponding expected total cost is \$3,600.49. If we increase the value of  $C_3$  from \$7,000 to \$10,000, we expect to have a longer testing time. In this case ( $C_1 = \$25$ ,  $C_2 = \$200$ ,  $C_3 = \$10,000$ ,  $\mu_y = 0.1$ , and  $x = 0.05$ ), the optimal release time is  $T^* = 27$  hours and the corresponding expected total cost is \$3,723.95.

Table 6.1. Optimal release time for  $C_1 = \$25$ ,  $C_2 = \$200$ ,  $C_3 = \$7,000$ ,  $\mu_y = 0.1$ , and  $x = 0.05$

| Release time $T^*$ (hours) | Expected total cost |
|----------------------------|---------------------|
|                            | $E(T)$ (\$)         |
| 19.5                       | 3,607.34            |
| 20.0                       | 3,604.47            |
| 20.5                       | 3,602.39            |
| 21.0                       | 3,601.07            |
| 21.5*                      | 3,600.49            |
| 22.0                       | 3,600.60            |
| 22.5                       | 3,601.39            |
| 23.0                       | 3,602.81            |
| 23.5                       | 3,604.83            |

# A Generalised Software Cost Model

## Notations:

|          |   |
|----------|---|
| $C_0$    | set-up cost for software testing  |
| $C_3$    | cost of removing an error per unit time during the operational phase                                |
| $C_4$    | loss due to software failure  |
| $W$      | variable of time to remove an error during the warranty period in the operation phase               |
| $\mu_w$  | expected time to remove an error during the warranty period in the operation phase, which is $E(W)$ |
| $T_w$    | period of warranty time   |
| $\alpha$ | the discount rate of the testing cost   |

## Additional Assumptions:

- (6) There is a set-up cost at the beginning of the software development process.
- (7) The cost of testing is a power function of the testing time. This means that at the beginning of the testing, the cost increases with a higher gradient, slowing down later.
- (8) The time to remove each error during the warranty period follows a truncated exponential distribution.
- (9) The cost to remove errors during the warranty period is proportional to the total time of removing all errors detected between the interval of  $(T, T_w)$ .

Similarly, from assumption 8, the truncated exponential density function of error removal time during the warranty period is

$$q(w) = \frac{\lambda_w e^{-\lambda_w w}}{\int_0^{T_0} \lambda_w e^{-\lambda_w w}} \text{ for } 0 \leq w \leq T_0'. \quad (8)$$

Therefore, the expected time to remove an error during the warranty period is

$$\mu_w = \frac{1 - (\lambda_w T_0' + 1)e^{-\lambda_w T_0'}}{\lambda_w (1 - e^{-\lambda_w T_0'})}. \quad (9)$$

The expected software system cost comprises of the set-up cost, the cost to do testing, the cost incurred in removing errors during the testing phase and during the warranty period, and the risk cost in releasing the software system by time T. Hence, the expected total software system cost E(T) can be expressed as follows (Pham, 1999):

$$E(T) = C_0 + C_1 T^\alpha + C_2 m(T) \mu_y + C_3 [m(T+T_w) - m(T)] \mu_w + C_4 [1 - R(x|T)],$$

where  $0 \leq \alpha \leq 1$ .

(10)

**Example 6.2** Considering a set of testing data given in Table 5.1 and Example 6.1, the mean value function is

$$m(T)=142.32(1-e^{-0.1246T}).$$

Given  $C_1=\$50$ ,  $C_2=\$25$ ,  $C_3=\$100$ ,  $C_4=\$1,000$ ,  $\mu_y=0.1$ ,  $\mu_w=0.5$ ,  $x=0.05$ ,  $\alpha=0.05$ , and  $T_w=20$ . We obtain the following results in Table 6.2.

The optimal release time in this case is  $T^*=24.5$  and the corresponding expected total cost is \$1,836.15.

Table 6.2. Optimal release time for  $C_0=\$100$ .

| $T^*(\text{hours})$ | $E(T)(\$)$ |
|---------------------|------------|
| 22.5                | 1,843.31   |
| 23.0                | 1,843.31   |
| 23.5                | 1,839.52   |
| 24.0                | 1,837.17   |
| 24.5*               | 1,836.15   |
| 25.0                | 1,836.39   |
| 25.5                | 1,837.82   |
| 26.5                | 1,840.35   |
| 26.5                | 1,843.93   |



**Example 6.3** Given  $C_1=\$50$ ,  $C_2=\$25$ ,  $C_3=\$100$ ,  $C_4=\$10,000$ ,  $\mu_y=0.1$ ,  $\mu_w=0.5$ ,  $x=0.05$ ,  $\alpha=0.05$ , and  $T_w=20$ . Using Theorem 6.2, we obtain the results in Table 6.3.

The optimal release time in this case is  $T^*=30.5$  and the corresponding expected total cost is \$3,017.13.

Table 6.3. Optimal release time for  $C_0=\$1.00$

| $T^*(\text{hours})$ | $E(T)(\$)$ |
|---------------------|------------|
| 28.5                | 3,029.72   |
| 29.0                | 3,024.41   |
| 29.5                | 3,020.60   |
| 30.0                | 3,018.20   |
| 30.5*               | 3,017.13   |
| 31.0                | 3,017.30   |
| 31.5                | 3,018.64   |
| 32.0                | 3,021.09   |
| 32.5                | 3,024.57   |



**Cap. 1. Software Reliability Assessment**

**Cap. 2 NHPP Software Reliability Models**

**Cap. 3 Software Cost Models**

**Cap. 4 Some Statistical Approaches**

## Null Hypothesis

The practice of science involves formulating and testing *hypotheses*, assertions that are capable of being proven false using a test of observed data. It is important to understand that the *null hypothesis can never be proven*. A set of data can only **reject** a null hypothesis or **fail to reject it**. For example, if comparison of two groups (e.g.: treatment, no treatment) reveals no statistically significant difference between the two, it does not mean that there is no difference in reality. It only means that there is not enough evidence to reject the null hypothesis (in other words, one *fails to reject the null hypothesis*)

## Statistical significance

In statistics, a result is called *statistically significant* if it is unlikely to have occurred by *chance* alone, according to a pre-determined threshold probability, the *significance level*.

One often *rejects the null hypothesis* when the p-value is less than the *significance level*  $\alpha$  (Greek alpha), which is often 0.05 or 0.01. When the null hypothesis is rejected, the result is said to be *statistically significant*.

## P values

### Definition of a P value

Consider an experiment where you've measured values in two samples, and the means are different. How sure are you that the population means are different as well? There are two possibilities:

- . The populations have different means.
- . The populations have the same mean, and the difference you observed is a coincidence of random sampling.

The P value is a probability, with a value ranging from zero to one. It is the answer to this question: If the populations really have the same mean overall, what is the probability that random sampling would lead to a difference between sample means as large (or larger) than you observed?



## P values

How are P values calculated? There are many methods, and you'll need to read a statistics text to learn about them. The choice of statistical tests depends on how you express the results of an experiment (measurement, survival time, proportion, etc.), on whether the treatment groups are paired, and on whether you are willing to assume that measured values follow a Gaussian bell-shaped distribution.

## Common misinterpretation of a P value

Many people misunderstand what question a P value answers.

If the P value is 0.03, that means that there is a 3% chance of observing a difference as large as you observed even if the two population means are identical. It is tempting to conclude, therefore, that there is a 97% chance that the difference you observed reflects a real difference between populations and a 3% chance that the difference is due to chance. Wrong. What you can say is that random sampling from identical populations would lead to a difference smaller than you observed in 97% of experiments and larger than you observed in 3% of experiments.

You have to choose. Would you rather believe in a 3% coincidence? Or that the population means are really different?



## Statistical hypothesis testing (I)

The P value is a fraction. In many situations, the best thing to do is report that number to summarize the results of a comparison. If you do this, you can totally avoid the term "statistically significant", which is often misinterpreted.

In other situations, you'll want to make a decision based on a single comparison. In these situations, follow the steps of statistical hypothesis testing.

1. Set a threshold P value before you do the experiment. Ideally, you should set this value based on the relative consequences of missing a true difference or falsely finding a difference. In fact, the threshold value (called alpha) is traditionally almost always set to 0.05.

## Statistical hypothesis testing (II)

2. Define the null hypothesis. If you are comparing two means, the null hypothesis is that the two populations have the same mean.
3. Do the appropriate statistical test to compute the P value.
4. Compare the P value to the preset threshold value. If the P value is less than the threshold, state that you "reject the null hypothesis" and that the difference is "statistically significant". If the P value is greater than the threshold, state that you "do not reject the null hypothesis" and that the difference is "not statistically significant".

# DOF – Degrees Of Freedom

In **statistics**, *the number of degrees of freedom is the number of values in the final calculation of a statistic that are free to vary.*

[http://en.wikipedia.org/wiki/Degrees\\_of\\_freedom\\_%28statistics%29](http://en.wikipedia.org/wiki/Degrees_of_freedom_%28statistics%29)

A data set contains a number of observations, say,  $n$ . They constitute  $n$  individual pieces of information. These pieces of information can be used either to *estimate parameters* or *variability*. In general, **each item being estimated costs one degree of freedom**. The remaining degrees of freedom are used to estimate variability.

**A single sample:** There are  $n$  observations. There's one parameter (the mean) that needs to be estimated. That leaves  $n-1$  degrees of freedom for estimating variability.

**Two samples:** There are  $n_1+n_2$  observations. There are two means to be estimated. That leaves  $n_1+n_2-2$  degrees of freedom for estimating variability.

# **EOC – End Of Course!**

<http://www2.imm.dtu.dk/~popen/pec/pec.html/>