NTNU
Norwegian University of
Science and Technology

Efficient Handling of Empirical Probability Distributions in RAMS Models

Florian Marcel Carl Müller

Reliability, Availability, Maintainability and Safety (RAMS)
Submission date: June 2017
Supervisor: Antoine Rauzy, MTP

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Preface

This master thesis was written during the spring semester 2017 at the Department of Mechanical and Industrial Engineering, Norwegian University of Science and Technology (Trondheim).

It is part of the two-year international master's program in RAMS (Reliability, Availability, Maintainability and Safety).

The topic is based on the specialization project done in the autumn semester 2016 titled "Comparison of Machine Learning Assisted Non-Parametrical Statistics with Parametrical Approaches". This was modified to focus on the efficient handling aspect of non-parametrical data.

The thesis is supervised by Professor Antoine Rauzy.

It is written for readers with basic understanding in the fields of reliability analysis and the application of failure models.

Trondheim, 2017-06-11

Müller

Florian Müller

Acknowledgement

Thanks to Anne Barros and Bo Henry Lindqvist for providing the academic knowledge of reliability and lifetime data analysis.

Thanks to Antoine Rauzy for guiding me through my specialization project and this master thesis.

Thanks to my friends in our shared student office – Renny, Sun, Mariska – who were always available to engage in lively discussions in the wide topics of RAMS.

Thanks to all the anonymous people on Stack Overflow, who asked and answered Python questions relevant to my problems. Without them, I would have struggled much more to write the roughly 3500 lines of code I have written as part of this thesis.

Abstract

Performing reliability assessments always relies on utilizing data. Most often, this data is provided in the form of historic failure dates. To understand this data, models are used to derive reliability characteristics from it.

These models can be parametric, trying to describe the system by means of mathematical equations. They can also be empirical, letting the raw data describe the system without assuming a certain outcome.

Handling parametric models is convenient, as they are described by often just one value. Empirical probability distributions are built on all available data and hence requires them to be fully defined. Handling this amount of data is cumbersome.

Part of this thesis is proposing different methods to represent the empirical reliability estimator. These representations try to combine convenient usage while keeping accuracy.

Representing the empirical reliability graph by a reduced amount of linear segments is proposed and discussed. This is an efficient way to compress huge datasets to a low number of descriptive points to interpolate in.

Furthermore, the feasibility to use polynomial regression on the empirical probability distribution is evaluated.

The computational efficiency of all methods is compared. For all practical purposes, the time to retrieve a reliability estimate is negligible.

Parametric and empirical approaches are applied to various datasets and the results discussed. The empirical methods outperform the exponential estimator in all cases.

The given experiment hypothesis is validated on each of the four experiments: The empirical probability distributions do match sufficiently well the reference reliability and the computational efficiency is negligible for all practical purposes.

Contents

List of Abbreviations

Abbreviation	Meaning
CSV	Character Separated Values
ISO	International Organization for Standardization
KM	Kaplan-Meier
MET	Maximum Event Time
MTTF	Mean Time To Failure
OREDA	Offshore and Onshore Reliability Data
PDS	Pålitelighet av Datamaskinbasert Sikkerhetssystemer (Norwegian acronym for "Reliability of Computer based Safety systems")
PFD	Probability of Failure on Demand
RAMS	Reliability, Availability, Maintainability & Safety
SIL	Safety Integrity Level
SIS	Safety Instrumented Systems

Chapter 1

Introduction

1.1 Background

Since the early usage of machinery, stakeholders want to know how long their equipment will last. Systems eventually break down or their performance do no longer meet the requirements.

The process of failure estimation is always a balancing act between being too conservative and too risky. This can lead to either wasting potentially good equipment and money, or endangering people and the environment in case of a hazard.

The emerging field of statistics led to the widespread use of failure distribution models for this task. Parameter based models like the exponential one are commonly known and used. As soon as their parameters are found (given, assumed or estimated), the simple subsequent use is appealing to most engineers.

The selection of those parameters however is the critical step in the process. Its validity is defining the accuracy of all following calculations (Risk assessment, Definition of safety barriers, SIL allocation, etc.). The extent of consequences is often underestimated – a manufacturer stating a value for λ in their data sheet does not necessarily know what the customer is using this value for.

When compressing all known information into a limited amount of parameters – mostly only one or two – information gets lost. There are however other methodologies to estimate the failure behavior of equipment. This loss can be avoided by using empirical methods. The information gained by applying this method is however not easy to represent.

With the emerging trend of "Big Data", Industry 4.0 and continuous condition monitoring, better failure prediction is highly sought after. This trend however is also making the matter of efficient data handling more important. The vast amount of generated and available data has to be processed in an intelligent way, often by means of truncation and compression.

1.2 Objectives

This thesis is highlighting the differences between parametric and empirical reliability estimation. It further proposes a method to represent the empirical distribution in efficient ways and applies them to numerous examples. Examining the required amount of failure data to give sufficient estimators is an additional objective. An efficiency and accuracy assessment is carried out to compare all approaches.

1.3 Limitations

Limitations are given by the lack of real raw data. Plenty of failure rates are available, but access to detailed listings of failure dates and fleet composition are rare. In order to increase the number of experiment cases, some data is self-generated based on mathematical expressions.

This thesis is not focusing on the statistical methodologies to evaluate the fit of various curves in a mathematical way. It uses simple, comprehensible tools to obtain a first evaluation of the proposed methods.

1.4 Approach

Different example data sets are presented with various origins and size. The two methods, parametric and empirical, are applied to each dataset and the results compared and discussed.

In order to implement the proposals and carry out the necessary calculations a Python-program is written as part of this thesis.

1.5 Structure of the Report

The thesis is opened with highlighting the current literature work around the topic of empirical reliability evaluation in chapter 2.

The following chapter 3 gives a motivating example including some background information on data sources and a case study illustrating the problem.

The conceptual developments are introduced in chapter 4. The proposed data representations are explained within this chapter as well a method to allow prediction beyond observed mission times. It also contains a section about the efficiency measures and results of the methods under evaluation.

The main chapter for the experiments carried out can be seen in chapter 5. Four datasets of varying origin are assessed for their reliability behavior.

A concluding statement and a proposal of future works can be seen at last in chapter 6.

The Python program developed is shown in the Appendix.

Chapter 2

Related Works

Parameter-based models and their benefits and disadvantages compared to empirical distribution models are widely discussed in various fields of sciences.

As soon as the unknown parameters are estimated, the parametric models are easy to use. If however the assumptions are false, then the resulting model can be misleading. Non-parametric methods perform better than poorly specified parametric models in almost all cases [21]. Perretti furthermore states "it is best to let the data tell us how the system works without imposing preconceived ideas on the outcome". This is also called the "true model myth" as stated in [8].

As reported by Bobrowski [3], empirical methods are recommended if there is no information available for a possible underlying distribution.

Similar conclusions are also drawn by Mokhtarian [16], stating that when both non-parametric and parametric methods are applicable to a problem, the parametric method is usually preferred because of its efficiency and simple use. However when the assumptions for those parameters are questionable, non-parametric methods are more suitable.

The oil- & gas industry, traditionally sensitive to safety and reliability matters, is following a dedicated standard for the acquisition of reliability data: ISO 14224 – "Petroleum, petrochemical and natural gas industries – Collection and exchange of reliability and maintenance data for equipment". [7] This standard however does not include methods for analysis and applying of reliability and maintenance data. It gives however principles on the calculation of basic reliability parameters in the appendix.

Chapter 3

Motivating Example

3.1 Why exponential distribution?

Many aspects of understanding the world we live in can be broken down to an essential question: When do events occur?

When will the remaining radiation be decreased to 50%?
When will next earthquake happen?
When will there be another fire in the forest?
⋮

And lastly the question for all RAMS related tasks:
When will the next failure happen?

Describing – and predicting – the failure occurrence behavior of certain events is a major task of statisticians working in probability theory. Probability models are commonly used to achieve those two tasks.

Numerous probability distributions exist. In the RAMS aspect the most commonly [24] used ones are:

- Exponential distribution
- Weibull distribution
- Gamma distribution
- Homogeneous Poisson Processes

Of all available and suitable probability distributions, the exponential distribution is the prevalent one. Various reasons can be found for that:

1. **Ease of use**
   The exponential distribution is described by a single scalar parameter ("Hazard rate", "Failure rate", commonly used symbol: λ, commonly used unit: $[\frac{1}{h}]$).
   Having only one variable makes handling of data plain and simple. It can be entered by hand, looked up in tables and books and takes up almost no memory when stored digitally.

2. **Ease of comparison**
   When comparing different systems and components, the failure behavior is an important factor in the decision process. Comparing different failure rates yields instant results: A higher value represents a higher likelihood of a failure in a given time frame and hence a less preferable option.

   Additionally a confidence interval can be given in a generic $[\text{Value}]^{+[\text{Upper Margin}]}_{-[\text{Lower Margin}]}$ style.

3. **Suitability**
   A major criterion for using the exponential distribution is that many systems can be described good enough with it. It gives sufficient accuracy within a given confidence interval, as many setups seem to "follow" the exponential law.

4. **Constant failure rate**
   The nature of the exponential law is its constant failure rate. The probability of having a failure in a time interval of a given length is the same regardless of its position along the time axis. It has no memory. A full explanation of exponential distribution's properties can be found in [24]. This characteristic is required for many reliability tools, algorithms and classifications. This includes methods like Markov Chains, PFD calculations and SIL definition.

3.2 Current situation

When designing safety relevant systems, the RAMS engineer is relying on given information. This information is accessible in data sheets directly from the manufacturer, or gathered by collected and processed lifetime data. Latter is provided by their own company, if enough historic data is available, or by agencies specializing in aggregating relevant data.

3.2.1 Data sheets

The quality of information found in data sheets is depending on the kind of equipment.

Components for generic industrial or domestic usage often lack information about the failure behavior. Getting this knowledge is a costly and lengthy procedure. This is often not considered required information by the customer, either due to the low cost of the equipment or the short estimated usage time.

More advanced products often give information in the form of MTTF (Mean Time To Failure) values. While this value can be useful to compare competitors, it gives no information about the underlying failure probability distribution – a MTTF value can be calculated for any given distribution [14]. It is merely the integral of the reliability distribution – see Equation 3.1.

$$MTTF = \int_0^\infty R(t) \tag{3.1}$$

Equipment intended for safety relevant applications give the most insightful information. Depending on the industry sector, some applicable standard might require the indication of certain information. This includes MTTF, failure rate values and naming the standard according to which this data is gathered.

3.2.2 Databases / Books

Various databases for failure data exist. Due to the sensitive nature of this data, the access to them is restricted. Access is usually granted by paying a subscription fee. The scope of this thesis is on data collections of the oil and gas industry.

3.2.2.1 OREDA

The OREDA project is sponsored by eight oil and gas companies [17]. The fundamental task of this organization is to aggregate, process and exchange reliability related information. The full set of data is accessible to their partners in an online database and the reduced information level can be found in their printed OREDA handbooks ([18], [19]).

The OREDA organization is also involved in issuing the standard ISO 14224: "Petroleum, petrochemical and natural gas industries – Collection and exchange of reliability and maintenance data for equipment" [7].

The introduction chapter of the handbook describes the origin and processing of the data presented.

An important basic assumption is given early in this chapter:

"*All the failure rate estimates presented in this handbook are therefore based on the assumption that the failure rate function is* constant *and independent of time, in which case* $z(t) = \lambda$, *i.e. the failure rates are assumed to be exponentially distributed with parameter* λ."

The method to calculate the maximum likelihood estimator λ based on the collected information is given in Equation 3.2. This is also the method described in [24].

$$\hat{\lambda} = \frac{\text{Number of failures}}{\text{Aggregated time in service}} = \frac{n}{\tau} \tag{3.2}$$

The handbook further introduces the "OREDA estimator", a modified averaging estimator that considers data origins from different installations and samples [26].

3.2.2.2 ExproSoft Wellmaster

ExproSoft is a company founded in 2001 as an out-spring from the Sintef research society in Trondheim [6]. The original aim of the company was to improve the performance of Down-Hole Safety-Valves by use of statistical analysis of history reliability data. It expanded the scope towards complete installations. The company provides the tool "Wellmaster", an interactive page to access, analyze and display the available reliability data.

According to direct information requested by the company, they are using the same method to calculate an estimator for a constant failure rate λ as seen before in Equation 3.2.

3.2.2.3 PDS Data Handbook

The PDS forum [20] is a collaboration of circa 25 participants. Those are representatives of operators, manufacturers and research societies with an interest in safety instrumented systems (SIS).

They regularly meet and release two matters: The "PDS Method Handbook" and the "PDS Data Handbook" [25].

The PDS Method Handbook gives an approach to implement and verify SIL requirements according to IEC 61508 / 61511 standards. The PDS Data Handbook is based on experience with operating SIS. The given data dossiers are based on multiple data sources. They range from using OREDA, direct vendor data, expert judgment and operational reviews.

3.3 Small case study

3.3.1 Purpose description

The reliability performance of a component or system is evaluated assuming an underlying exponential distribution. Doing so can give results, which are not as holistic compared to using empirical methods. This case study will demonstrate the differences on a small example system.

3.3.2 System description

Following scenario is used throughout the remaining section:

A reliability engineer wants to perform a reliability assessment of a system. The system is built up of six elements made out of three different types. A reliability block diagram illustrating the system can be seen in Figure 3.1.

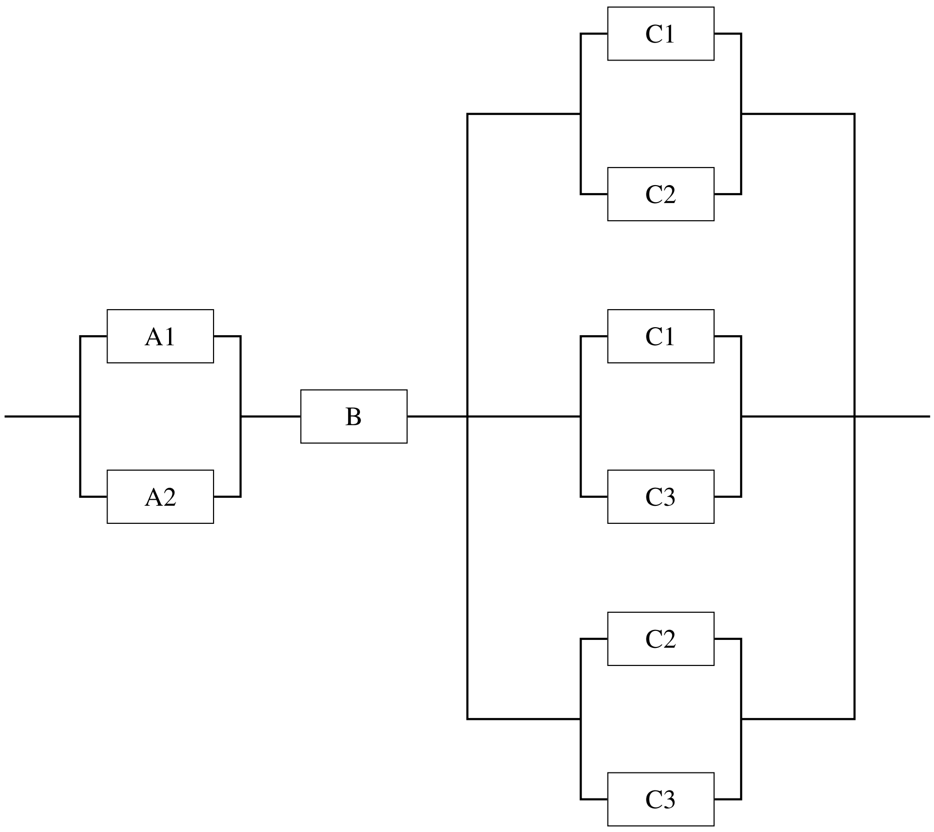


Figure 3.1: Reliability block diagram illustrating the motivational case study

3.3.3 Input Data

All three components in this scenario (A, B, C) were used in previous similar installations. The maintenance department kept track of all the times a unit was taken out of service – either due to failure, or due to end of mission. This results in a right censored dataset for each component containing the event time and the information weather it was a failure or censoring.

The full set of used input data can be seen in Figure A.1 in the appendix.

3.3.4 Output

When assessing the reliability of a system, the probability of having the Top-Event is a critical indicator. There are several methods to calculate this value – some are exact, some are just approximations.

The methodology being used within this case study is described in subsection A.1.2.

In addition to that, the individual component's reliability is calculated and shown.

3.3.5 Parametric approach

The input data (failure times) are processed to give a constant failure rate for each component type (A, B, C). The results can be seen in Table 3.1 and the calculation in subsection A.1.1.

Component Type	Failure rate $[\frac{1}{\mathrm{h}}]$
A	4.11×10^{-4}
B	3.70×10^{-4}
C	1.35×10^{-4}

Table 3.1: Resulting failure rates based on exponential distribution

An illustration of the three corresponding reliability diagrams can be seen in Figure 3.2. They cover the time range up to one year (8760 hours).

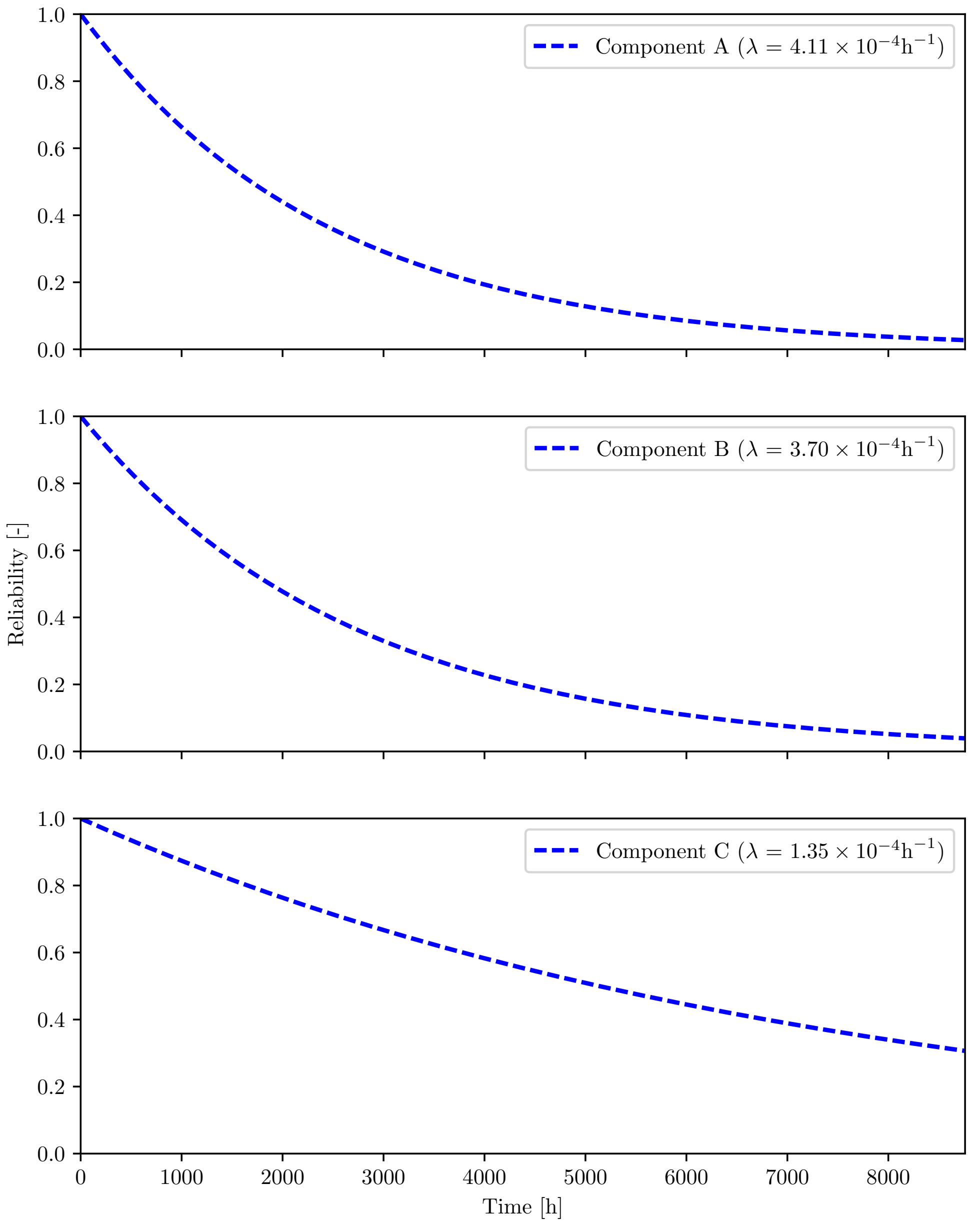


Figure 3.2: Reliability diagrams for one year (8760 hours) for all three components (A, B, C) based on exponential distribution

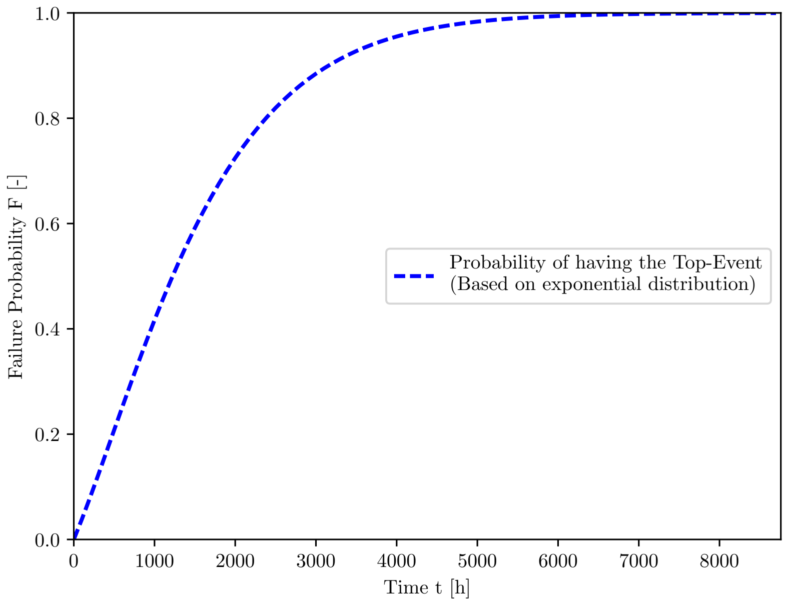


Figure 3.3: Probability of having the Top-Event with using exponential distribution

3.3.6 Empirical approach

The traditional Kaplan-Meier estimator is used to gain information about the reliability characteristics of each component.

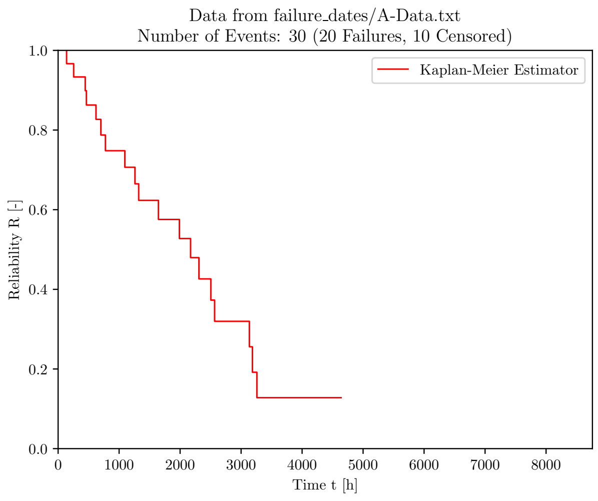
The Kaplan-Meier estimator for censored data used within this thesis is based on the original research paper [9].

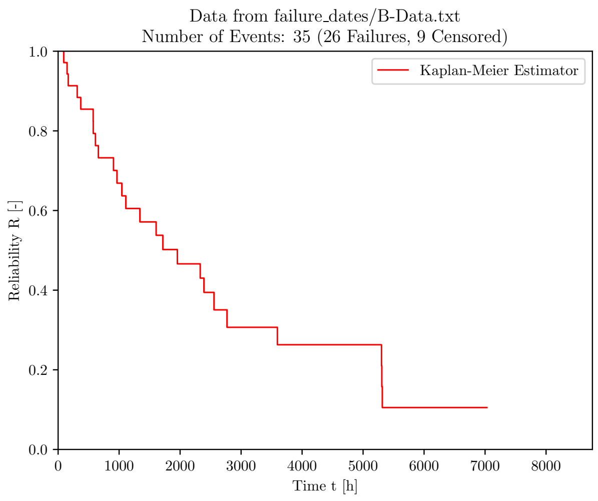
If $T_{(1)} < T_{(2)} < \ldots$ are the times with at least one failure, and n_i ,d_i are, respectively, the number at risk and the number of failures at T(i), then Equation 3.3 can be formulated.

$$\hat{R}(t) = \prod_{i:T_{(i)} \leq t} \frac{n_i - d_i}{n_i} \tag{3.3}$$

The results of those calculations are graphical depictions of failure probability over time, as seen in Figure 3.4.

It is worth mentioning that the Kaplan-Meier value is only available up to the highest observed event time. No information beyond that time is known, hence the abrupt ending of the graph. No assumptions are made for this time span. Furthermore, the maximum observed time is different for each component.





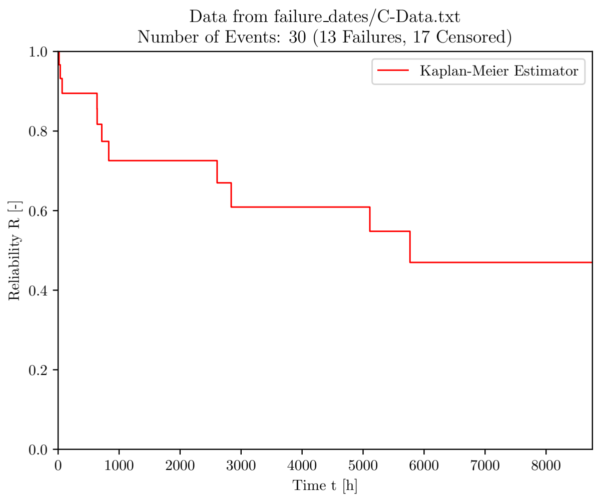


Figure 3.4: Reliability diagrams for all three components (A, B, C) based on empirical estimation

Similar to subsection 3.3.5, the Top-Event probability of the system is calculated using the empirical reliability information.

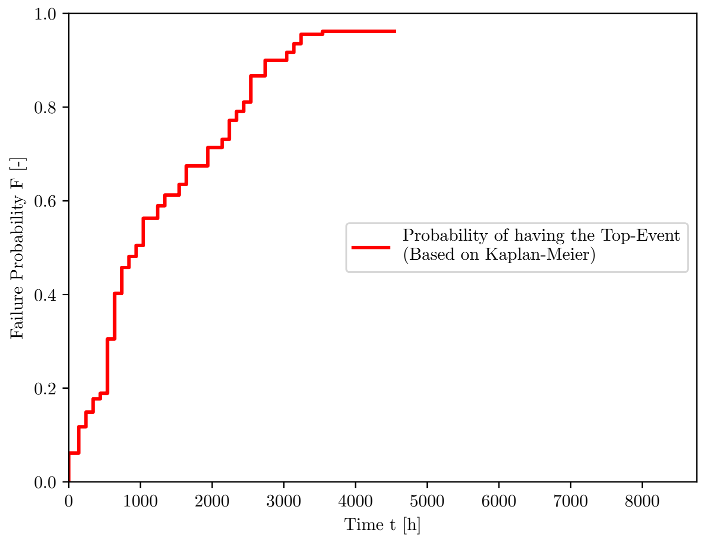


Figure 3.5: Probability of having the Top-Event with using Kaplan-Meier

3.3.7 Comparison

A visual comparison of the three individual reliability diagrams can be seen in Figure 3.6. Furthermore Figure 3.7 shows the probability of the Top-Event both based on empirical and exponential modelling.

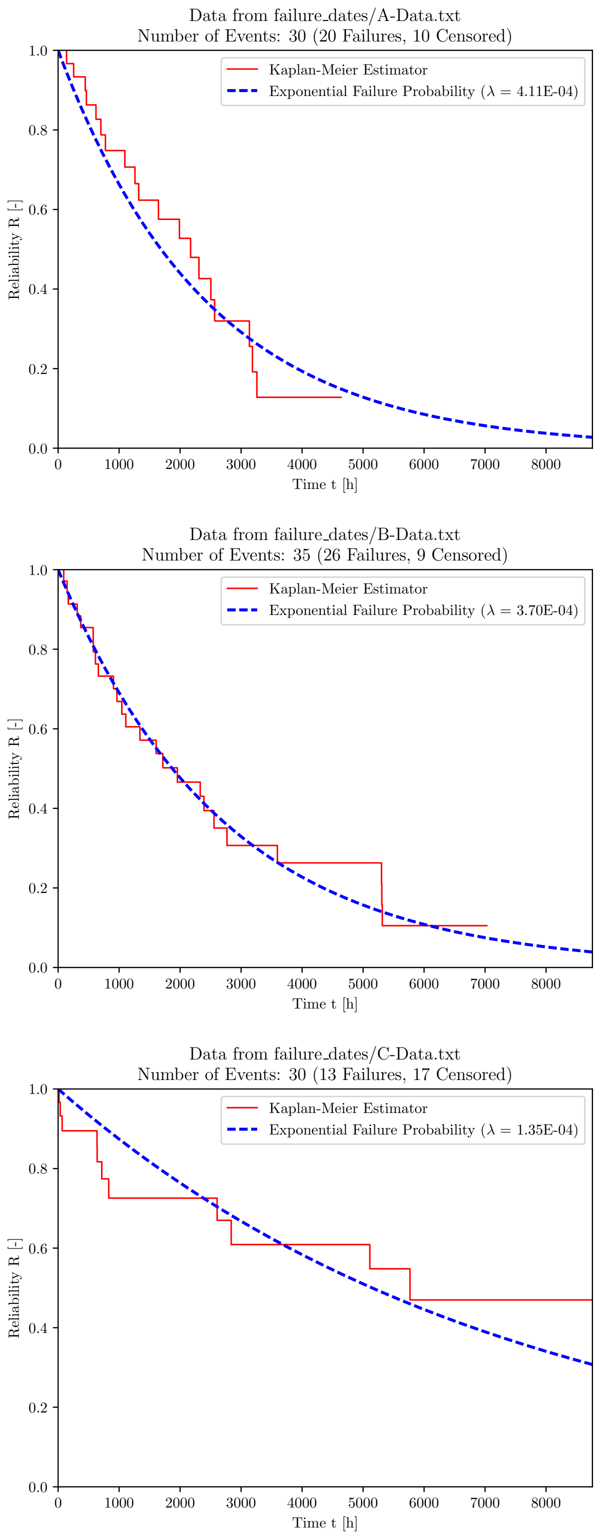


Figure 3.6: Comparison of reliability diagrams (exponential and empirical)

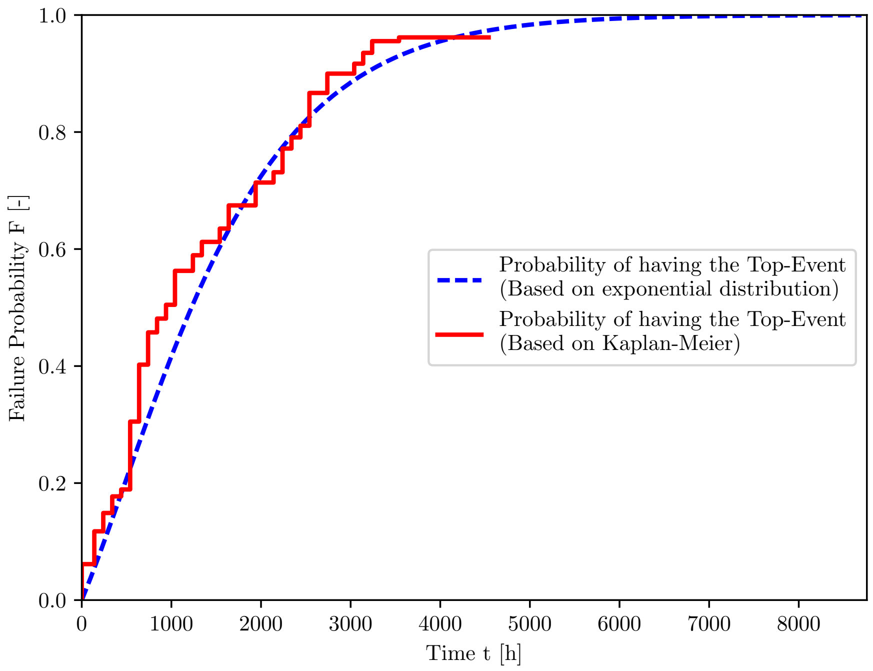


Figure 3.7: Probability comparison of having the Top-Event (exponential and empirical)

3.3.8 Case study conclusion

The scope of this case study is to show the generic difference between using parametric and empirical reliability data in a simple reliability model consisting of only six elements.

Depending on the input data, a parametric approach can give approximations of varying quality. Just by visual comparison of the empirical Kaplan-Meier and the exponential distribution, it is apparent that the goodness of fit is prone to be insufficient for some applications.

Evaluating the probability comparison of having the Top-Event for exponential and empirical methods however reveals an issue with the latter: After a certain time, the empirical method does not return any estimator of the reliability. This can be critical in a system involving numerous components.

Not all components have sufficient data available to give empirical reliability estimates for the complete anticipated mission time. Even if just one of the components used is lacking the data, the overall system-reliability estimator is at stake.

Chapter 4

Conceptual Developments

4.1 Introduction to the problem

The case study as seen in section 3.3 illustrated the potential for using empirical distributions instead of exponential assumed distributions. The input data for the case-study is of reasonably small size, as can be seen in Figure A.1. For each of the three different components, the number of recorded events are between 30 and 35.

The amount of data for real systems can however be larger. Advancements like "Big Data", "Industry 4.0" and "Internet of Things" are creating vast amounts of data.

4.2 Characteristics, Assumptions and Limitations

The reliability of a component is commonly represented as a 2-dimensional function R(t). The resulting reliability distribution curve has certain inherent properties. They are caused by a few basic assumptions and limitations:

- At time t = 0, the beginning of the mission time, the equipment is assumed to be fully functioning. So R(0) = 1.
- Equipment is not repaired, the item can only degrade. This results in a graph which is monotonously decreasing.
- The data considered is right-censored. As soon one specimen is taken out of the observation, the reliability will never decrease down to zero, as there is always a lack of information when that component actually failed.
- If failure times data is self-generated, the dataset only contains failure times and no censoring times.
- The empirical probability distribution only gives data up to the maximum observed event time. After that time, there is no information. If one wants to predict beyond that, assumptions have to be made (see section 4.5).

4.3 Intoduction of modified Kaplan-Meier

The original Kaplan-Meier estimator results in a stepwise curve as seen in Figure 3.4. Degradation however is a continuous process. The chance of survival at $t + \Delta t$ is slightly lower than at t. Discrete steps are not representing this behavior appropriately.

Additionally, when comparing a step-curve to a continuous curve – as it is the case when for example comparing the Kaplan-Meier curve to the exponential curve – the resulting difference becomes unsteady. This is illustrated in Figure 4.1. The green curve shows the relative difference from the exponential value using the Kaplan-Meier value as the reference.

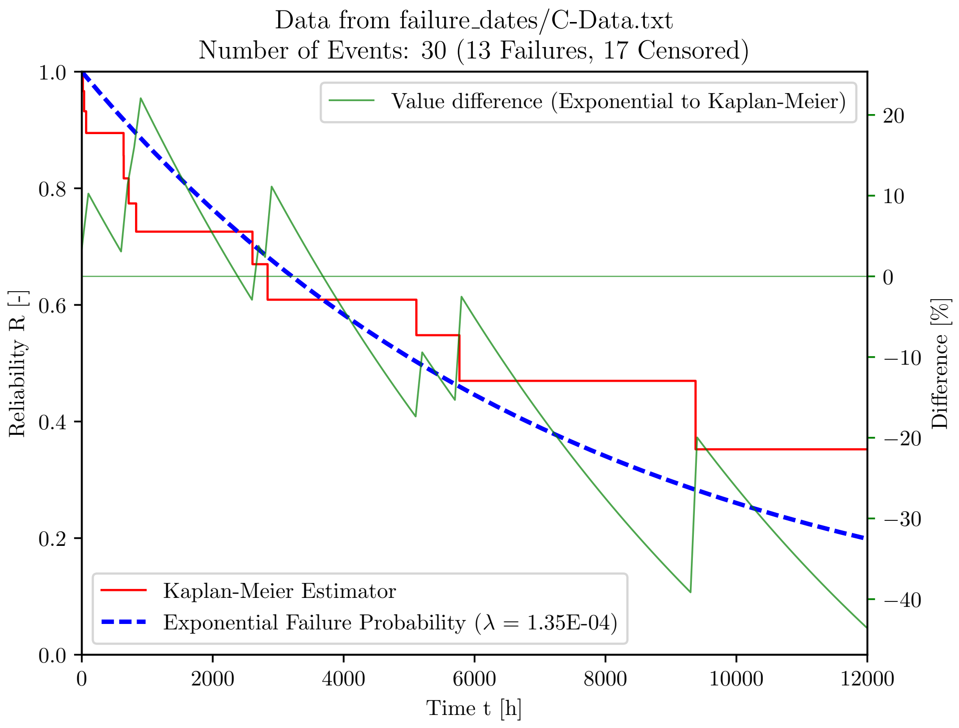


Figure 4.1: Illustration of unsteady differences between Kaplan-Meier curve and exponential curve

Different ways to make a smoother appearance are proposed and used by various authors ([30], [2], [10]).

These require varying mathematical implementation effort. The scope of this thesis is not to evaluate different smoothing algorithms, hence a simple, comprehensive method is proposed.

The method used within this thesis is carried out as follows:

1. The original Kaplan-Meier method is used to create a two-dimensional matrix containing the time, censoring indicator and reliability estimator. This can be seen in Table 4.1

Time	Censoring	Kaplan-Meier estimator
0	1	1
16	1	0.965
35	0	0.965
63	1	0.824
⋮	⋮	⋮

Table 4.1: Example of Kaplan-Meier array

2. The first value is always a 100% reliability at mission start, so:
   $R_{\text{Smooth}}(0) = 1$

3. The last value is always the Kaplan-Meier value at the last known event time (Maximum Event Time, MET), so:
   $R_{\text{Smooth}}(\textit{Maximum Event Time}) = R_{\text{Kaplan-Meier}}(\textit{Maximum Event Time})$

4. All censored values are taken out of the original Kaplan-Meier array. They do not contribute information to the definition of the curve.

5. At every step – the vertical path downwards – a new reliability estimator is generated for that time. The mean of the two Kaplan-Meier reliability values used for the step is the new value for this time.

6. The resulting points are connected by piecewise linear segments.

The resulting curve is demonstrated in Figure 4.2. It also shows the new coordinates used for the plot.

This smooth version of the Kaplan-Meier estimator is used instead of the original Kaplan-Meier values for all further comparisons.

When comparing the differences between the smooth Kaplan-Meier and the exponential curve, similar to Figure 4.1, the resulting graph shows no unsteady changes. There are still segments, but they are connected continuously and hence the differences are also continuous. The graphical representation of the differences over time can be seen in Figure 4.3.

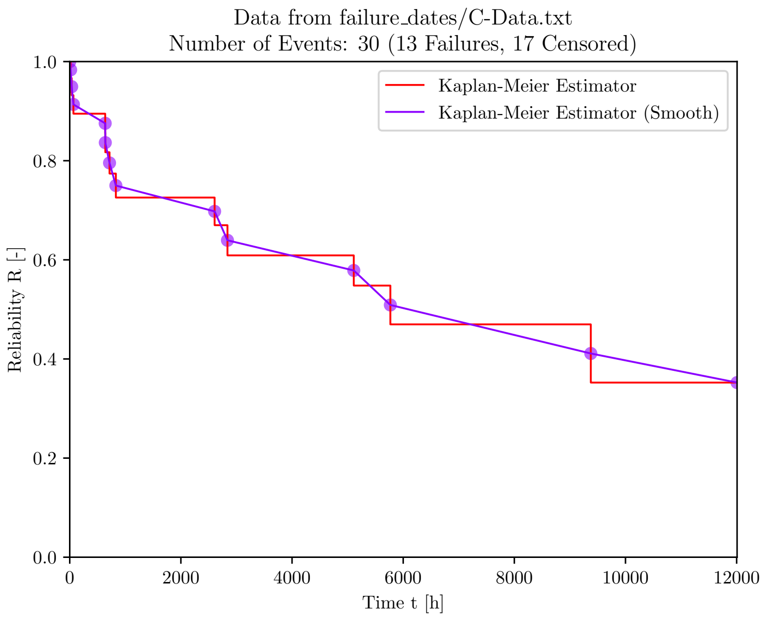


Figure 4.2: Modified Kaplan-Meier estimator for smoother curve

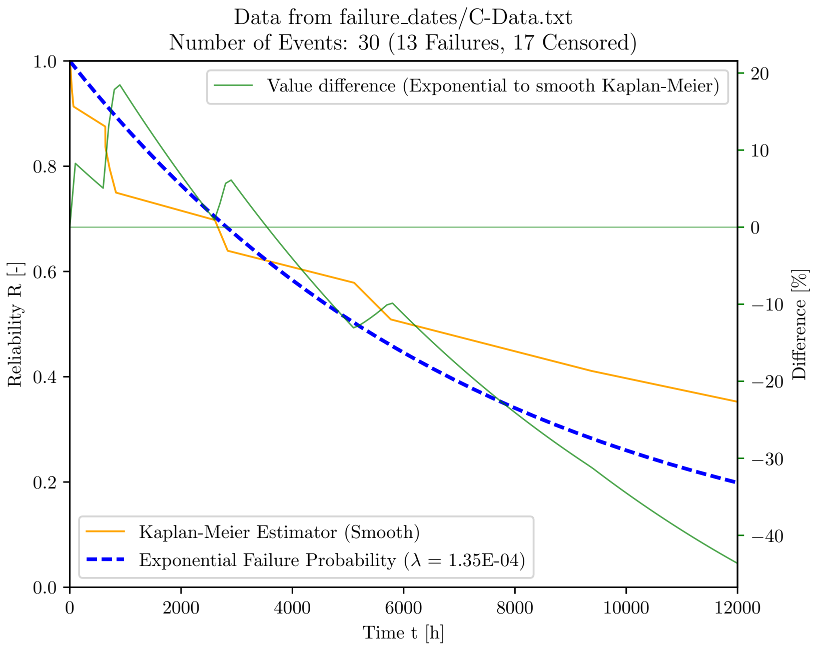


Figure 4.3: Differences between smooth Kaplan-Meier curve and exponential curve

4.4 Different representations

4.4.1 Reduced Kaplan-Meier

4.4.1.1 Description

Piecewise linear functions are extensively used to approximate functions [4]. The modified Kaplan-Meier, as introduced in section 4.3, is already a piecewise linear model.

The number of linear segments in the the smoothed Kaplan-Meier Estimator is equal to the number of failures in the input dataset. This gives the most accurate level of results, but also requires the most data points and hence increases the data handling effort and reduces the efficiency of calculation.

The generic scope for this representation is to find the smallest number of segments which still succeed in approximating the smoothed Kaplan-Meier curve within a given threshold level.

4.4.1.2 Mathematical background

Following algorithm is implemented to get the resulting reduced array

1. Create an array with two endpoints of a segment
   - The first value is always a 100% reliability at mission start, so:
     $R_{\text{Reduced}}(0) = 1$
   - The last value is always the Kaplan-Meier value at the last known event time, so:
     $R_{\text{Reduced}}(\textit{Maximum Event Time}) = R_{\text{Kaplan-Meier}}(\textit{Maximum Event Time})$
2. Separate this segment S_1 in 10 equidistant times $t_{1..10}$
3. At each of those times: Evaluate the reliability value based on the smoothend Kaplan-Meier ($R_{\text{KM}}(t_{1..10})$) and also the linear interpolated values from that first segment ($R_{\text{Reduced}}(t_{1..10})$)
4. Calculate the differences between each of the 10 pairs, calculate the square of said value, summarize those squared values, built the mean (see Equation 4.1)
5. Compare the mean with a given threshold. If the value is below that value, the segment is considered to be sufficiently accurate. If not, the segment number will be saved in a list
6. Retrieve the list of all non-sufficient segments
7. For each of those segments, divide their time duration by two. Add another point to the array with this new time and a reliability estimate based on the smoothed Kaplan-Meier.
8. Repeat Step 2 to 5 until there are no more segments in need of treatment.
9. Unnecessary entries are removed afterwards. This is done by removing the second index (first one is $R_{\text{Reduced}}(0) = 1$) and then checked if all remaining segments are still considered to be good. If yes, then this entry is deleted from the resulted array. If not, it is kept.

Manual adjustments to the threshold parameter are necessary for each application.

An illustration of the development of the algorithm is seen in Figure 4.4.

$$\frac{\sum_{t=1}^{10}(R_{\mathrm{KM}}(t_{\mathrm{i}}) - R_{\mathrm{Reduced}}(t_{\mathrm{i}}))^2}{10} \tag{4.1}$$

$$\begin{bmatrix} 0 & 1 \\ MET & R_{\mathrm{KM}}(MET) \end{bmatrix}$$

4.4.1.3 Data representation

Each segment of the smoothed Kaplan-Meier is defined by a data entry. They are essentially Cartesian coordinates for a two-dimensional curve.

An example can be seen in Listing 4.1

```

[[     0.             1.         ]
 [    93.             0.91170175]
 [   187.             0.90544117]
 [   375.             0.89292002]
 [   562.             0.88046547]
 [   750.             0.78191551]
 [  1501.             0.73017971]
 [  3002.             0.63500227]
 [  6004.             0.50253083]
 [ 12008.             0.3523202 ]]
```

Listing 4.1: Data representation for piecewise linear approximation used in reduced Kaplan-Meier

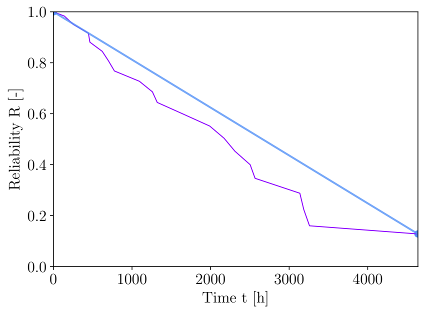
4.4.1.4 Data access

The data as seen in Listing 4.1 can be saved as a CSV file and later loaded back into the program.

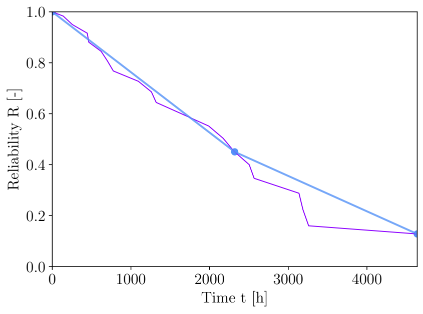
The Python package NumPy provides a dedicated routine to linearly interpolate values within a given array, named `numpy.interp` [27].

Running this methods requires 3 parameters:
The array giving the X-Coordinates (Time), the array giving the Y-Coordinates (Reliability) and the X-Coordinate (Time) to look up.

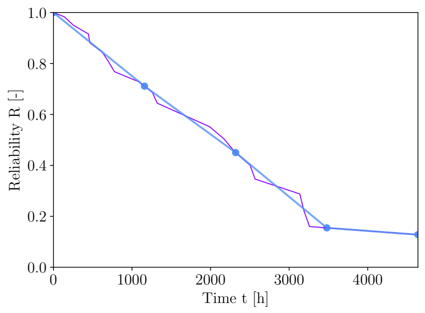
A discussion about the efficiency of this implementation is given in section 4.7.



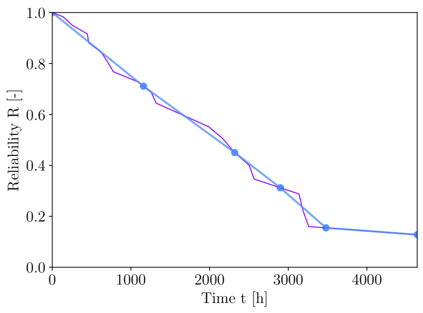
(a) First step, the initial single segment



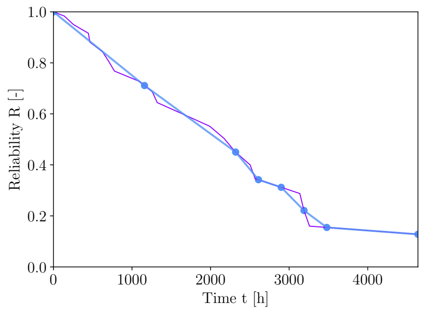
(b) Second step, 2 segments



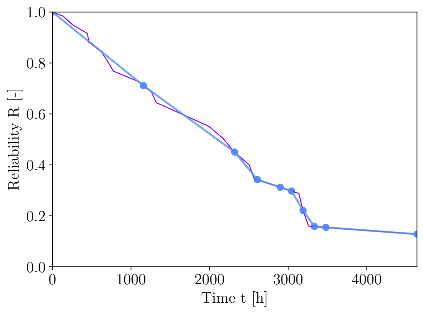
(c) Third step, 4 segments



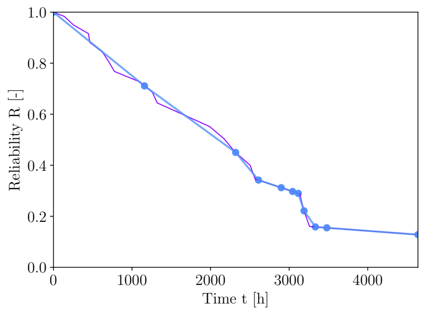
(d) Fourth step, 5 segments



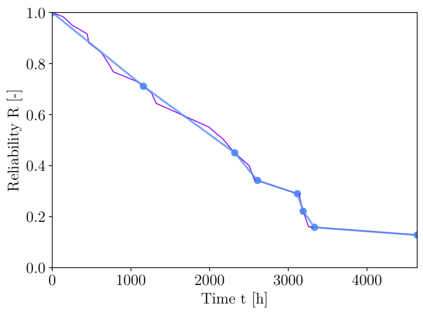
(e) Fifth step, 7 segments



(f) Sixth step, 9 segments



(g) Seventh step, 10 segments



(h) Eights step, 7 segments after removal of unnecessary segments

Figure 4.4: Illustration of steps for the reduction algorithm

4.4.2 Polynomial regression

4.4.2.1 Description

Polynomial regression is a widespread method to fit data to a mathematical equation.

This is done to by finding parameters of Equation 4.2

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{4.2}$$

The amount of degrees n is adjusted manually for every application.

The number of degrees and their corresponding values may inflict problems like overfitting or oscillation.

The polynomial regression does not incorporate known characteristics of the reliability curve like the monotonously decreasing shape and the starting condition at R(0) = 1.

4.4.2.2 Mathematical background

`numpy.polyfit` uses a least square polynomial fit, mathematical details can be found in [29].

The principle scope is to minimize the squared error, as seen in Equation 4.3, with $p(x_j)$ being the polynomial value and y_j the value to be fitted.

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2 \tag{4.3}$$

4.4.2.3 Data representation

Data is represented by the polynomial coefficients. The amount of coefficients is depending on the polynomial degree chosen.

These can be represented in a simple text file with one line per parameter, as seen in Listing 4.2.

```
1.234234234e1
4.123123123e2
5.2123123e-1
```

Listing 4.2: Data representation for a 3-degree polynomial equation

4.4.2.4 Data access

To access the data, the coefficients have to be loaded into the program. A simple routine to read in the lines of the textfile (see Listing 4.2) is used.

Python/Numpy provides a special class for polynomial data: `poly1d` [28]. It is "a one-dimensional polynomial class" and values can be conveniently retrieved by a corresponding function.

4.5 Implementation of prediction

When using a parametric distribution, like the exponential distribution, the whole system is fully described by a mathematical equation and it gives results for times from $t = 0$ to ∞.

The empirical Kaplan-Meier estimator – as well as the smoothed version of it – is only defined for times up to the last observed event (which can be either a failure or censoring).

While polynomial equations also give results for all every point of time, they are only fitted for the known times. For any times beyond that, polynomial graphs tend to drop rapidly. Hence they cannot be used for prediction.

A major reason to use models is not only to understand the observed characteristics of a system, but also to predict the performance beyond the time of available data.

In order to implement this requirement, an addition to empirical models is given: For all times beyond the largest observed time, use data resulting from using a constant failure rate.

This results in an unsteady jump of reliability values at the highest observed event time. It can be in either direction and thus create a scenario in which the reliability is not monotonously decreasing over time. Hence a proposal is made:

At the maximum observed event time calculate both reliability values – based on constant failure rate and by empirical estimation.

If the exponential value is lower than the empirical, it is kept being used for prediction. An unsteady step to a lower value is conservative and thus accepted. An example of this behavior can be seen in Figure 4.5.

If the value is higher than the last known empirical value, a new constant failure rate is calculated. See Equation 4.4 to see the formula used. It's the inverse of the reliability at the maximum known time.

$$\lambda_{\text{Prediction}} = -\frac{\ln \mathrm{R}_{\text{Smooth}}(\mathit{Maximum\ Event\ Time})}{\text{Maximum Event Time}} \tag{4.4}$$

This new adapted $\lambda_{\text{Prediction}}$ is then used for predictions. It will give a continuously decreasing reliability. This is illustrated in Figure 4.6.

When utilizing this method for a reliability assessment, the user should be notified if, and for which times, the estimated reliability is based on an assumed constant failure rate.

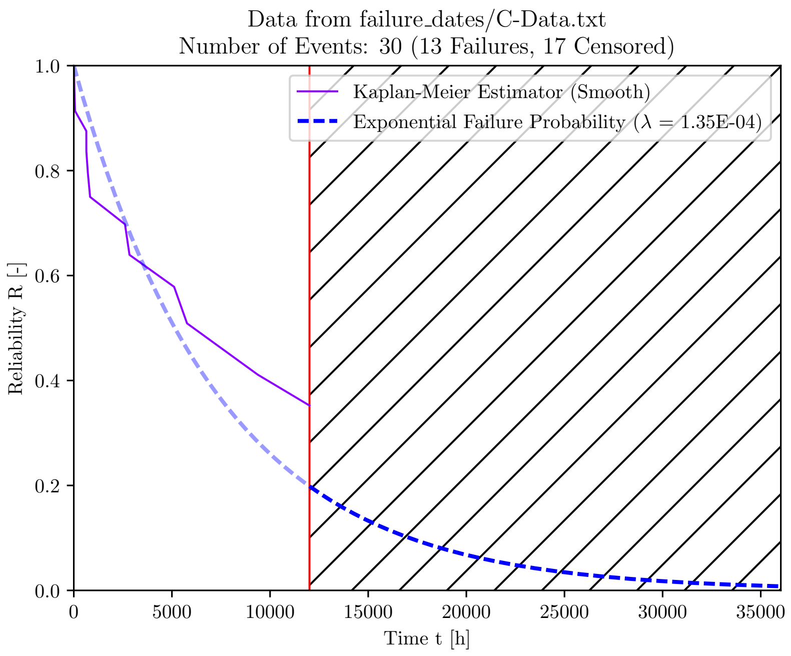


Figure 4.5: Example of prediction based on unaltered constant failure rate

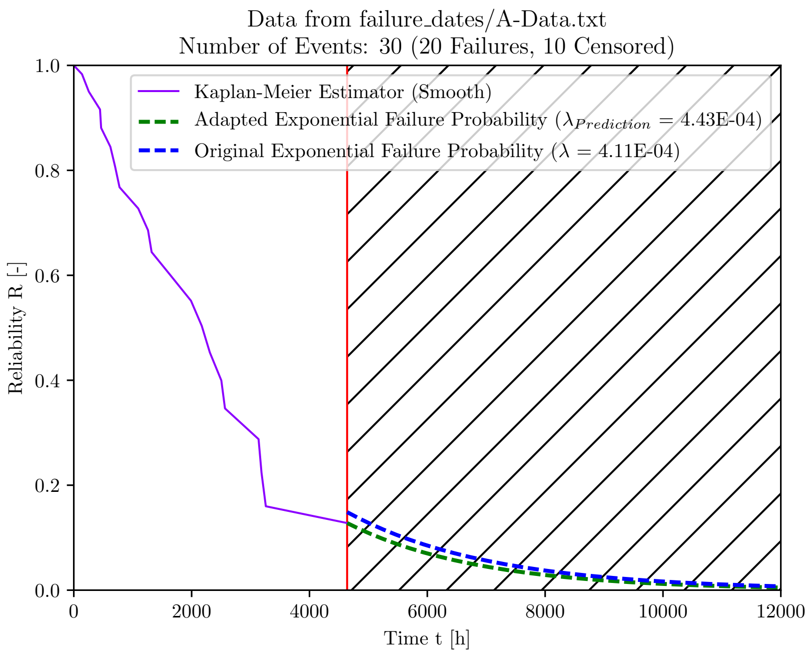


Figure 4.6: Example of prediction based on adapted constant failure rate

4.6 Accuracy of empirical probability distributions

The concept of reliability assessment itself builds on uncertainty and assumptions. There is no definite and known law behind the failure behavior of systems.

Evaluating the accuracy of any reliability model can be difficult. A "true" reference value is required to assess whether or not a result of a model is accurate or not.

If the data is self-generated by using a known generator with a defined mathematical model, this reference value is known and given. The estimated result from a model can be compared to the mathematically derived exact solution.

Real data however is not generated by an algorithm, but by a complex chain of events and consequences. There are no equations describing real failures. In this case, the smoothed Kaplan-Meier plot is used as a reference, as this estimator has no assumptions about any underlying failure distribution.

The essential task of evaluating the accuracy of the proposed empirical probability distribution is to compare two reliability plots with each other.

The Kolmogorov-Smirnov test is a non-parametric test to evaluate if two samples are drawn from the same distribution [22]. This test is used to give a numerical measure if the two curves under consideration are alike.

Following list describes how the Kolmogorov-Smirnov test value is implemented:

1. 30 uniformly random distributed times between 0 and the highest observed event time are selected
2. Reliability values of both models for each of those times (as illustrated with 10 time samples in Figure 4.7) are estimated
3. The values within each model for each discrete time step are accumulated
4. The value differences between those accumulation curves are calculated
5. The maximum difference found is the Kolmogorov-Smirnov test value (as indicated by red color in Figure 4.8)
6. Step 1. - 5. is repeated for 100 times and the mean of the resulting value calculated

This value is compared to the critical value for the two-sample Kolmogorov-Smirnov test for 30 compared samples. When using a significance value $\alpha = 0.05$ the value to compare against is 0.351 [5].

The simple example value (Figure 4.8) is 0.749. As this value is larger than the critical value for 10 samples (0.70), the two graphs shown are not based on the same distribution.

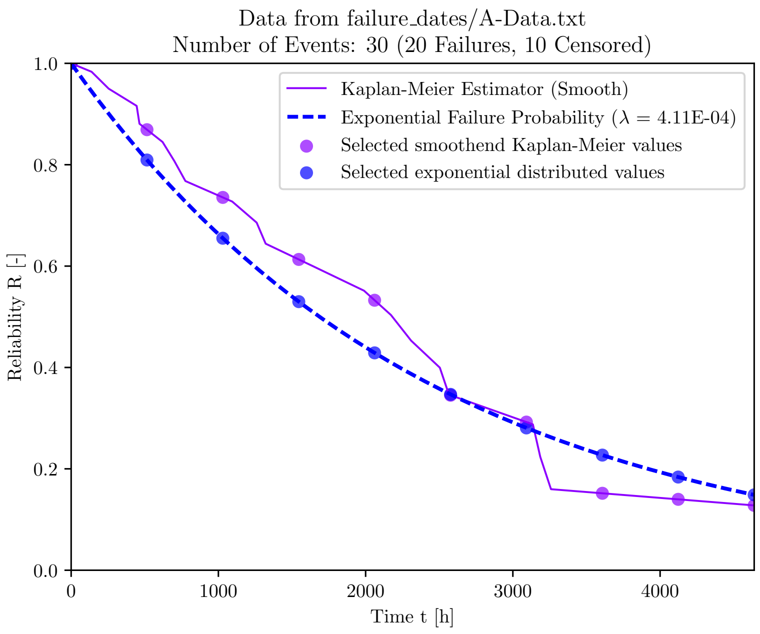


Figure 4.7: Illustration of the Kolmogorov-Smirnov test: First step

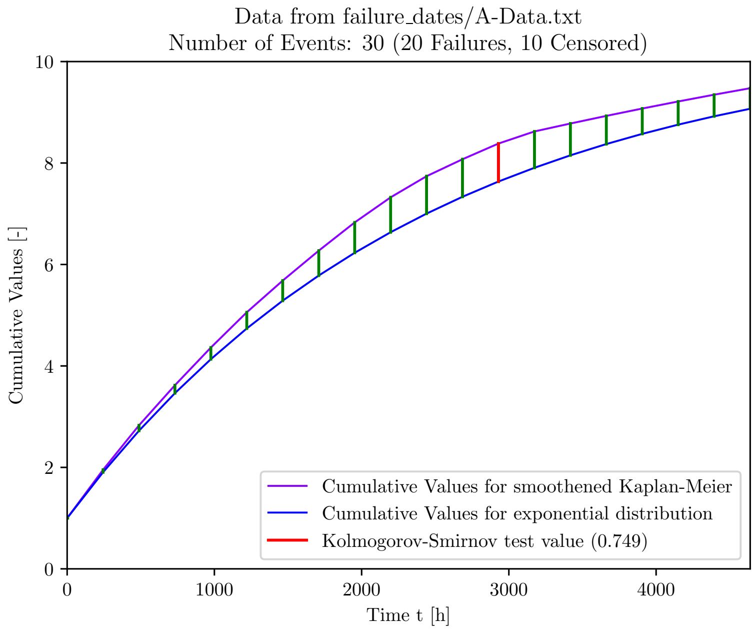


Figure 4.8: Illustration of the Kolmogorov-Smirnov test: Second step

4.7 Efficiency measures

Using a constant failure rate gives the benefit of calculating the estimated reliability at any given time by solving a simple equation (Equation 4.5, [24]). Only one value (λ) has to be known beforehand to fully describe the model.

$$R(t) = e^{-\lambda \times t} \tag{4.5}$$

Retrieving reliability values from the proposed empirical estimators needs more steps. These are briefly mentioned in subsubsection 4.4.1.4 and subsubsection 4.4.2.4.

The calculation of the Kaplan-Meier estimator and the subsequent routine to reduce the data is applied as preparation. Fitting the polynomial equation to the failure data is the required preparation for the polynomial estimator. The computational efficiency of those required steps is out of scope for the reliability estimation as described within this work. Its efficiency is relevant when failure data is continuously monitored and processed online.

Four different functions are evaluated in terms of their computational efficiency:

- Parametric method: Calculate R(t) based on Equation 4.5
- Empirical method:
  1. Load data array into memory
  2. Interpolate values within this array to retrieve an estimator for R(t)
  3. Solve polynomial equation to get an estimator for R(t)

All steps are executed 1000000 times and the mean value is calculated. Python's provided `timeit` function is utilized [23].

4.7.1 Efficiency of parametric estimation

In order to assess the time needed to calculate a reliability value based on Equation 4.5, it is implemented in a simple Python function as seen in Listing 4.3.

```python
def GetExponentialValue(Time, EstimatedLambda):
    return math.e ** (-(EstimatedLambda*Time))
```

Listing 4.3: GetExponentialValue function used for efficience measurement

A uniformly random distributed number between 0 and 10000 is used for each iteration as a time of interest, a uniformly distributed number between 0.001 and 0.00001 is chosen as the failure rate. The resulting time computational time per instance is 3,73 $\times$ 10^{-7} seconds.

4.7.2 Efficiency of empirical estimation

Loading data array into memory

Loading the data into memory is a necessary first step for further processing. The data is stored in a CSV text file and loaded into the program with the NumPy `loadtxt` function. The function implementing this routine used for measurement is seen in Listing 4.4.

This step is necessary for both the reduced Kaplan-Meier and the polynomial estimator. The polynomial estimator however has very little amount of data, as the number of reasonably used degrees is rather low.

```
def LoadArray():
    KM_Array = np.loadtxt("failure_dates/E_1000000_Efficiency_Array.txt")
```

Listing 4.4: LoadArray function used for efficiency measurement

Interpolating values

To evaluate the computational efficiency of the interpolation routine, a function as seen in Listing 4.5 is used.

```
def GetKaplanMeierValue(Time):
    return np.interp(Time, KM_Array[:,0], KM_Array[:,-1])
```

Listing 4.5: GetKaplanMeierValue function used for efficiency measurement

Polynomial values

Retrieving reliability values from the polynomial estimation is done by utilizing the polynomial functions from Numpy. The testing function used to evaluate the efficiency can be seen in Listing 4.6. A polynomial array with 6 degrees is used for assessment. Each parameter is a uniformly distributed random number between -1 and 1.

```
def GetPolyValue(Time,Polyobject):
    return Polyobject(Time)
```

Listing 4.6: GetPolyValue function used for efficiency measurement

Results

Unlike the parametric estimator, the empirical estimation steps require a dataset. The size of it has influence on the necessary time to run execute of both functions.

Four different datasets are used, with 6, 50, 1000 and 1000000 entries.

The results can be seen in Table 4.2.

	Size of Array			
	6	50	1000	1000000
Time to load array into memory [s]	$6,62 \times 10^{-4}$	$6,76 \times 10^{-4}$	$1,17 \times 10^{-2}$	$12,1$
Time to estimate reliability value [s]	$4,89 \times 10^{-6}$	$4,73 \times 10^{-6}$	$6,58 \times 10^{-6}$	$6,24 \times 10^{-3}$
Time to retrieve polynomial value [s]	$1,61 \times 10^{-5}$	-	-	-

Table 4.2: Empirical efficiency measurement results

4.7.3 Discussion and background information

The time needed to calculate a reliability value based on the exponential law is negligible.

For the empirical approach, the first step (loading of the data) has the biggest influence on overall execution time. This sub-step is also influenced the most by the size of the input data.

For this assessment, the data is saved and loaded to/from plaintext CSV files. NumPy offers a binary format optimized for storing and reading numerical data. Using an universal plaintext format has the benefit of being readable by any application and humans.

After applying the reduction algorithm for the Kaplan-Meier estimator, the amount of data is unlikely to exceed values above 50. In that range, the combined time of loading the array and estimating the reliability is in a magnitude of 10^{-4} seconds.

Even for the extreme case of 1 million entries, the loading time of the array is in the order of several seconds, and as soon as the array is loaded, the actual estimation time is still less than 0,1 seconds.

All calculations are performed on a machine with following brief specifications:

- Windows 7 Enterprise
- Intel Core i7-3770, 3.40 GHz
- 16 GB Memory
- Python 3.6.0
- NumPy 1.11.3

Chapter 5

Experiments

5.1 Experiment hypothesis

The hypothesis to be tested within this chapter is described as follows:

With enough data points provided, empirical probability distributions match sufficiently well the true probability distributions.

Additionally: It is possible to calculate reliability values, based on an empirical model, with sufficiently good computational efficiency.

Consequently, these empirical models can be used as a generic input of RAMS tools.

5.2 Experiment description

In the first step the data used is described. Its origin, any pre-existing knowledge and basic information like size and amount of failures.

If feasible (depending of the size) the whole data set is given, otherwise an excerpt of the data structure with a reference to the full data.

As a next step the data is processed. This is done in a parametric way and in both described empirical ways.

The parametric way is an estimation of a constant failure rate λ. The empirical methods as described in section 4.4 are applied to the data.

The accuracy of all results are compared. The reasoning and method behind the comparison is given in section 4.6. Both a qualitative, visual comparison and a quantitative evaluation are given.

5.3 First Data Set: Wellmaster

5.3.1 Data origin

The data for the first data set is extracted from the Wellmaster database by ExproSoft (See subsubsection 3.2.2.2). For the purpose of this thesis only limited access to the database is given – only the component "TRSCSSV" ("Tubing retrievable, surface controlled, subsurface safety valve", commonly known as "Downhole safety valve") is populated with failure data.

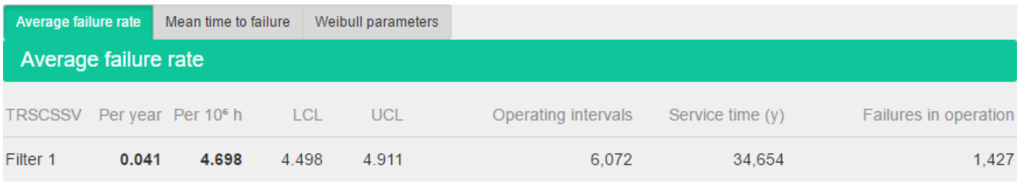
5.3.2 Data characteristics

The Wellmaster interface is showing basic characteristics about the failure data of the selected component. Figure 5.1 shows the information as they are presented. This includes the failure rate, confidence intervals and also service time and number of failures.

Additionally a graphical illustration of the survival probability can be accessed directly within Wellmaster – see Figure 5.2. It also shows the plot based on the constant failure rate as per Figure 5.1.

The Wellmaster software provides access to a CSV file with the recorded service times. Table 5.1 shows the content of it. Column `ServiceTimeInDays` gives the service time in days and column `Failures` is an indicator whether or not it is a failure or censoring (`1` corresponds to a failure, `0` to censoring). Only those two columns where considered for further processing. Column `LengthKm` had the value `0` for all entries, column `ServiceTimeLength` is completely empty and the `FailureId` is not taken into account.

In total the data consists of 6072 events – 1427 failures and 4645 censorings.



Average failure rate

TRSCSSV	Per year	Per 10⁶ h	LCL	UCL	Operating intervals	Service time (y)	Failures in operation
Filter 1	**0.041**	**4.698**	4.498	4.911	6,072	34,654	1,427

Figure 5.1: Average failure rate according to Wellmaster

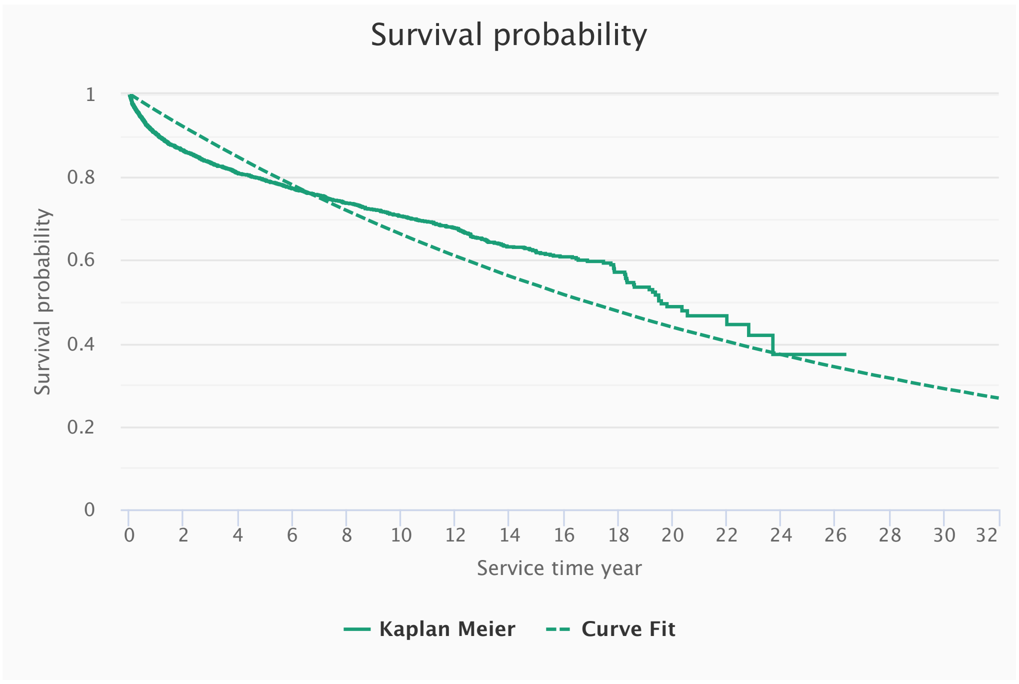


Figure 5.2: Survival probability plot according to Wellmaster

LengthKm	ServiceTimeInDays	ServiceTimeLength	Failures	FailureId
0	3336		0	
0	2265		0	
0	1726		0	
0	4150		0	
0	1628		0	
0	8603		0	
0	3036		0	
0	1228		0	
0	511		0	
0	2552		0	
0	950		0	
0	5073		0	
0	4390		0	
0	177		0	
0	252		0	
0	1368		1	1704
0	1605		0	
0	3112		0	
0	2199		0	
0	5526		1	2410
0	2390		0	
0	123		0	
0	1628		0	
0	1552		0	
0	1370		1	335
0	405		1	336
⋮	⋮	⋮	⋮	⋮

Table 5.1: Raw service times data provided by Wellmaster

5.3.3 Data processing

5.3.3.1 Parametric

The estimation of a constant failure rate is carried out according to Equation 5.1.

$$\lambda_1 = \frac{\text{Number of failures}}{\text{Aggregated time in service}} = \frac{1427}{303565392} = 4.70 \times 10^{-6} \frac{1}{\text{h}} \tag{5.1}$$

This value corresponds with the estimator given by Wellmaster as seen in Figure 5.1. The small deviation is most likely due to rounding differences for the service time. A figure showing the survival probability over time in comparison with the smoothed Kaplan-Meier is seen in Figure 5.3.

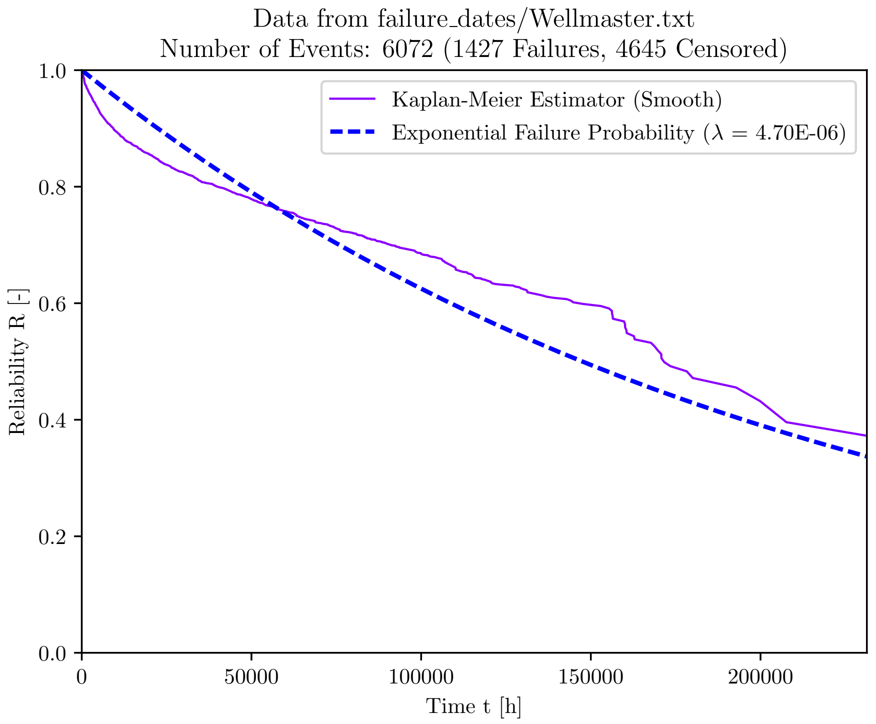


Figure 5.3: Survival probability plot for the first dataset based on exponential failure distribution

5.3.3.2 Empirical (Reduced Kaplan-Meier)

When applying the smoothed Kaplan-Meier, as introduced in section 4.3, a reliability plot is generated. The shape is similar to the survival probability plot given by Wellmaster (Figure 5.2).

The algorithm to reduce the amount of segments, as introduced in subsection 4.4.1, is applied to this dataset. The threshold parameter is set to 0.0003. An illustration of the resulting segments together with the smoothed Kaplan-Meier plot can be seen in Figure 5.4.

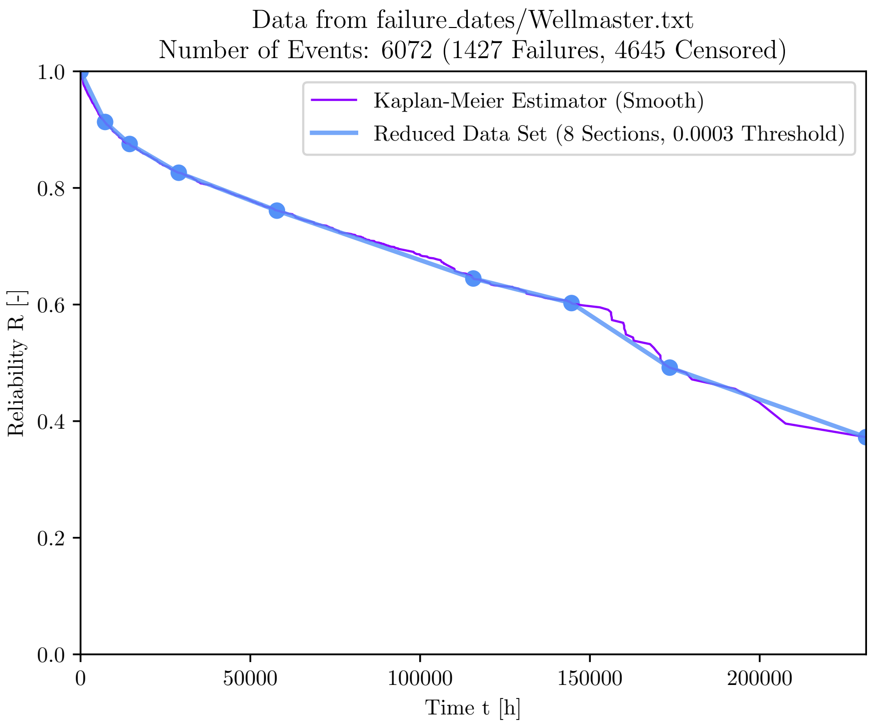


Figure 5.4: Survival probability plot for the first dataset based on reduced Kaplan-Meier

5.3.3.3 Empirical (Polynomial)

Applying the polynomial fit to the given data set gives Equation 5.2. The resulting graph is seen in Figure 5.5 in comparison with the smoothed Kaplan-Meier.

$$R(t) = 0.9679 - 6.792 \times 10^{-6} t + 7.937 \times 10^{-11} t^2 - 4.933 \times 10^{-16} t^3 + 9.964 \times 10^{-22} t^4 \quad (5.2)$$

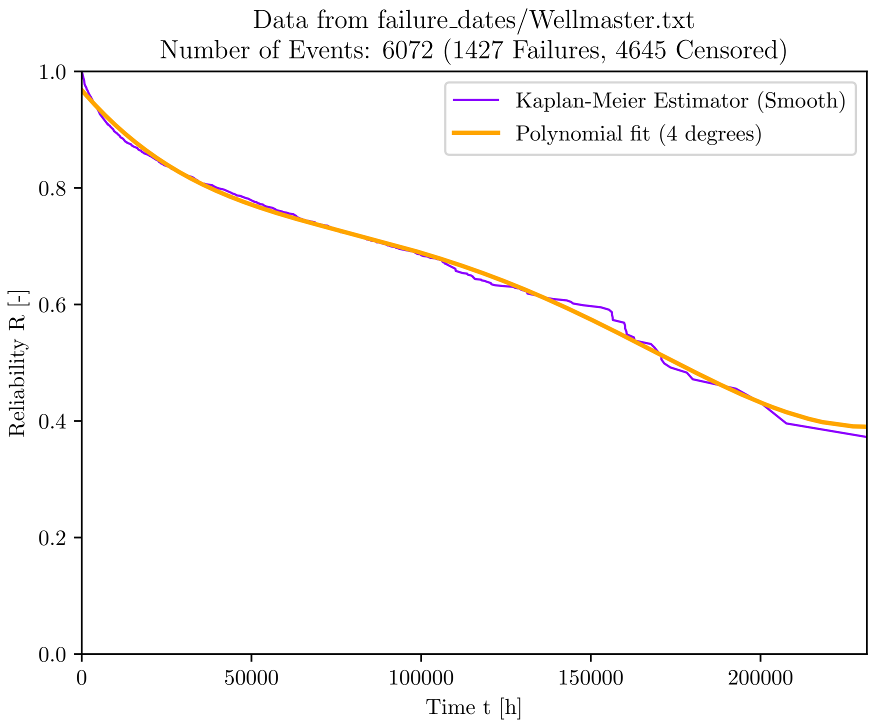


Figure 5.5: First Dataset: Comparison of smoothed Kaplan-Meier and polynomial fit

5.3.4 Accuracy comparison

As this dataset is not generated by mathematical equations, the smoothed Kaplan-Meier is used as the reference to compare accuracy.

A visual comparison of all estimators can be seen in Figure 5.6. Based on the shape of the curves it can be seen that the exponential model is too optimistic in the first 50.000 hours of operation. In times beyond that, the estimation is too pessimistic and potentially sufficient equipment might not be utilized completely.

A quantitative accuracy comparison is given in Table 5.2. Based on those Kolmogorov-Smirnov test results, the exponential distribution is not matching the reference reliability values (as 0,932 > 0,351). The accuracy of the reduced Kaplan-Meier is sufficient (as 0,071 < 0,351). Using a polynomial regression is giving comparable good results (with 0,098 < 0,351).

	Value
Reduced Kaplan-Meier vs. Smoothed Kaplan-Meier	0,071
Polynomial regression vs. Smoothed Kaplan-Meier	0,098
Exponential distribution vs. Smoothed Kaplan-Meier	0,932
Critical Kolmogorov-Smirnov test value	0,351

Table 5.2: Comparison of Kolmogorov-Smirnov test results for the first data set

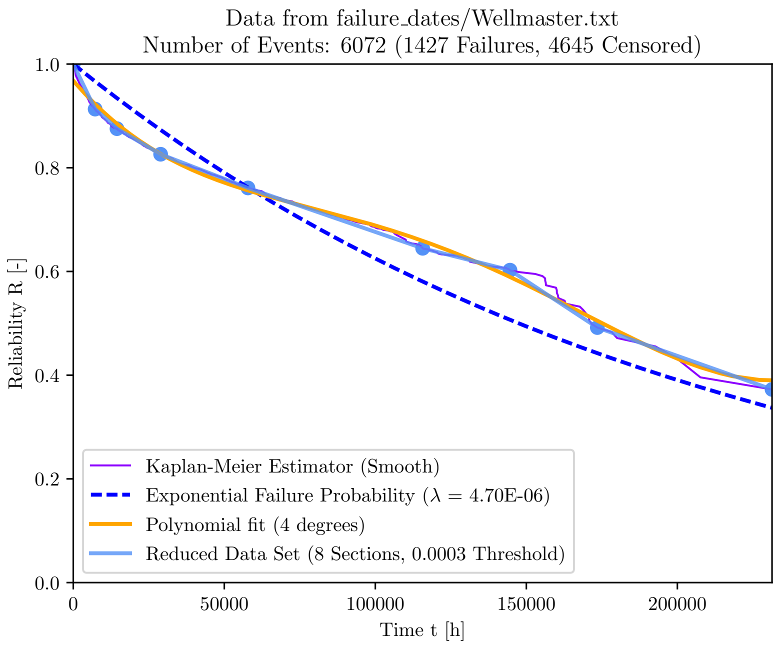


Figure 5.6: First Dataset: Comparison of smoothed Kaplan-Meier, reduced Kaplan-Meier, polynomial regression and exponential distribution

5.4 Second Data Set: Hard drives

5.4.1 Data origin

The company "Backblaze", based in California / USA, is operating a data hosting center for cloud based storage solutions. Within their server farm they operate more than 60.000 hard drives. Since 2013 the company is releasing raw statistical data on all their hard drives in use [1].

This data is freely available for everyone interested.

5.4.2 Data characteristics

The data is presented in CSV files. For each day of operation, one file is provided. An excerpt of one file can be seen in Table 5.3. There are 90 additional columns provided which give information retrieved by the on-board diagnostics. Those are omitted.

date	serial_number	model	capacity_bytes	failure
2016-10-01	MJ0351YNG9Z0XA	Hitachi HDS5C3030ALA630	3000592982016	0
2016-10-01	MJ0351YNG9WJSA	Hitachi HDS5C3030ALA630	3000592982016	0
2016-10-01	PL1321LAG34XWH	Hitachi HDS5C4040ALE630	4000787030016	0
2016-10-01	MJ0351YNGABYAA	Hitachi HDS5C3030ALA630	3000592982016	0
2016-10-01	Z305B2QN	ST4000DM000	4000787030016	0
2016-10-01	PL2331LAGN2YTJ	HGST HMS5C4040BLE640	4000787030016	0
2016-10-01	WD-WMC4N2899475	WDC WD30EFRX	3000592982016	0
2016-10-01	Z302A0YH	ST4000DM000	4000787030016	0
2016-10-01	Z305BT0W	ST4000DM000	4000787030016	0
2016-10-01	MJ0351YNG9Z7LA	Hitachi HDS5C3030ALA630	3000592982016	0
2016-10-01	Z302A0YE	ST4000DM000	4000787030016	0
2016-10-01	Z302PGH8	ST4000DM000	4000787030016	0
2016-10-01	Z3023VGH	ST4000DM000	4000787030016	0
2016-10-01	PL1311LAG2205A	Hitachi HDS5C4040ALE630	4000787030016	0

Table 5.3: Raw data as provided by Backblaze

This data representation is complete, but inconvenient in its usage for the used tool. Ross Lazarus ([12], [13]) has written a script to process this data into a different format. It keeps track of the appearance and disappearance of each individual drive (based on the unique `serial_number` field). The output is a summary file with one line per drive stating its observation time and reason of removal (failure or censoring). This approach neglects the operating hours of the drives before the first available daily report.

Data from the 2013 - 2016 period is used.

In total, the data consists of 92328 events – 5762 failures and 86566 censorings.

5.4.3 Data processing

5.4.3.1 Parametric

The estimator for a constant failure rate for the data can be seen in Equation 5.3. A plot showing the reliability based on this value is given in Figure 5.7. The scaling of the reliability ordinate is changed to show the range from 0,8 to 1,0 for better visibility.

$$\lambda_1 = \frac{\text{Number of failures}}{\text{Aggregated time in service}} = \frac{5762}{1486091064} = 3.88 \times 10^{-6}\,\frac{1}{\text{h}} \tag{5.3}$$

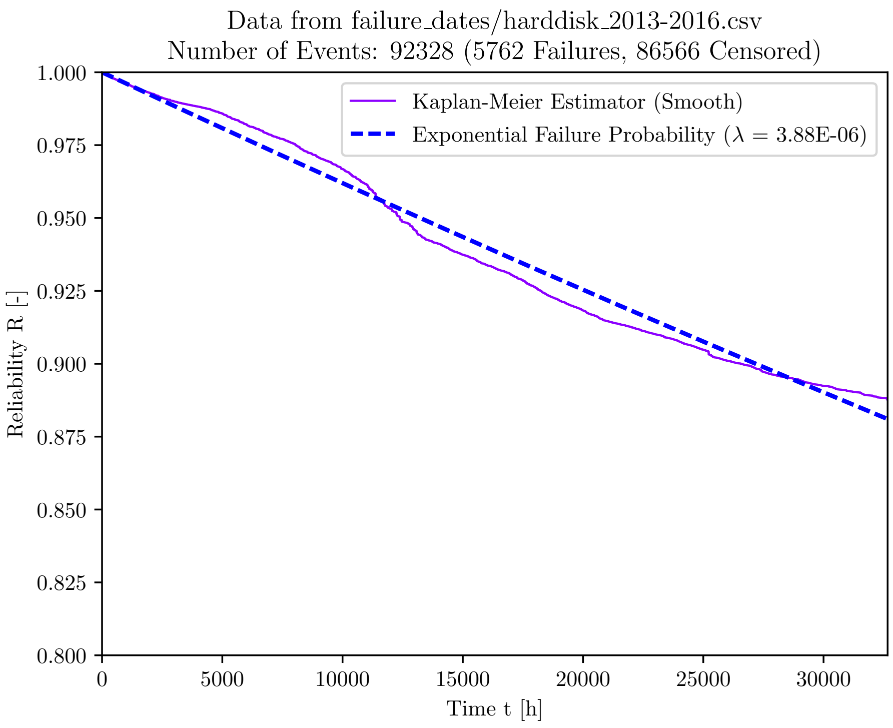


Figure 5.7: Second Dataset: Exponentially assumed failure distribution

5.4.3.2 Empirical (Reduced Kaplan-Meier)

Applying the reduced Kaplan-Meier algorithm to the data with a chosen threshold value of 0.00001 gives a representation with 5 segments. This can be seen in in comparison with the smoothed Kaplan-Meier in Figure 5.8.

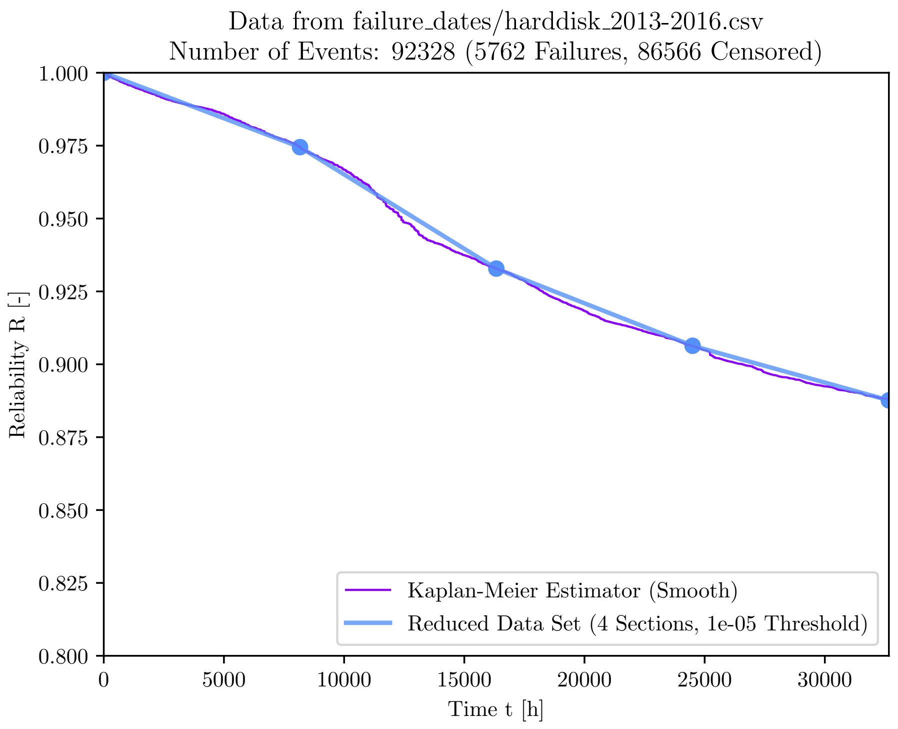


Figure 5.8: Second Dataset: Reduced Kaplan-Meier

5.4.3.3 Empirical (Polynomial)

Applying the polynomial fit to the second data set gives Equation 5.4. The resulting graph is seen in Figure 5.9 in comparison with the smoothed Kaplan-Meier.

$$R(t) = 0.9957 + 3.825\times 10^{-7}t - 5.5\times 10^{-10}t^2 + 2.24\times 10^{-14}t^3 - 2.762\times 10^{-19}t^4 \tag{5.4}$$

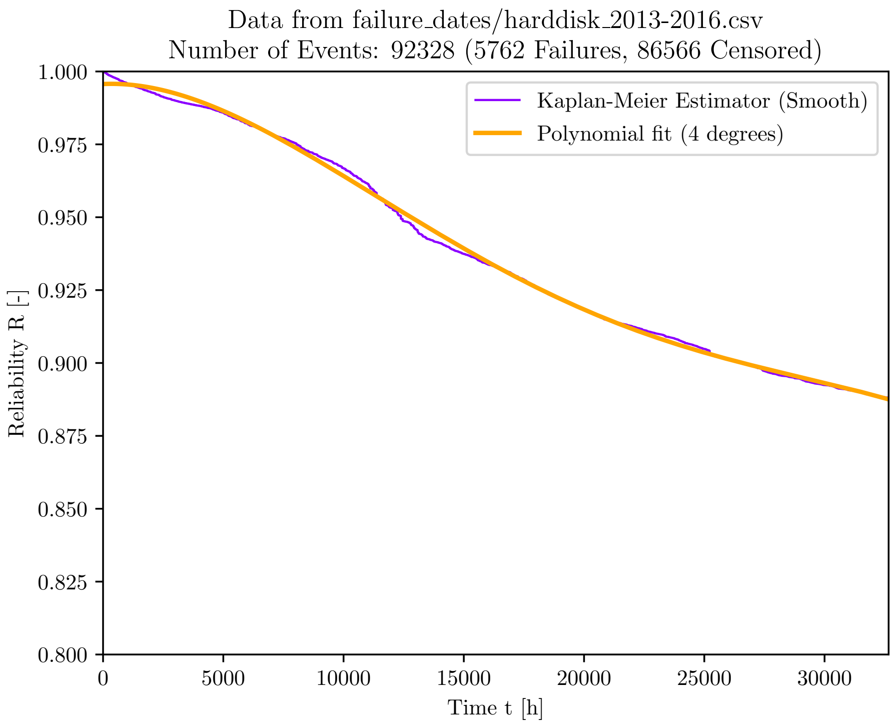


Figure 5.9: Second Dataset: Comparison of smoothed Kaplan-Meier and polynomial fit

5.4.4 Accuracy comparison

This data set is based on real failure data.

A visual comparison of all estimators can be seen in Figure 5.10.

All estimators are visually matching the trend of the data well, especially considering the adjusted scaling of the ordinate.

A quantitative comparison of accuracy is given in Table 5.4. Both the parametric and empirical distributions are meeting the given requirement and represent the reference sufficiently.

Therefore this dataset is used to compare the data requirements in section 5.7.

	Value
Reduced Kaplan-Meier vs. Smoothed Kaplan-Meier	0,027
Polynomial regression vs. Smoothed Kaplan-Meier	0,011
Exponential distribution vs. Smoothed Kaplan-Meier	0,051
Critical Kolmogorov-Smirnov test value	0,351

Table 5.4: Comparison of Kolmogorov-Smirnov test results for the second data set

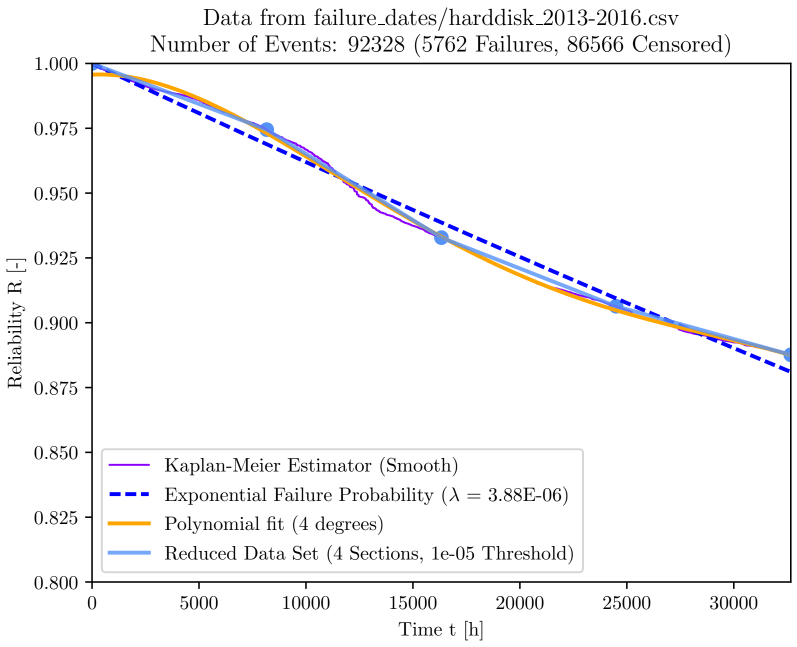


Figure 5.10: Second Dataset: Comparison of smoothed Kaplan-Meier, reduced Kaplan-Meier, polynomial regression and exponential distribution

5.5 Third Data Set: Modified Bathtub

5.5.1 Data origin

The traditional bath-tub curve, as illustrated in Figure 5.11, is described in almost every standard reliability textbook [11]. This curve is used as the basis to introduce a new distribution.

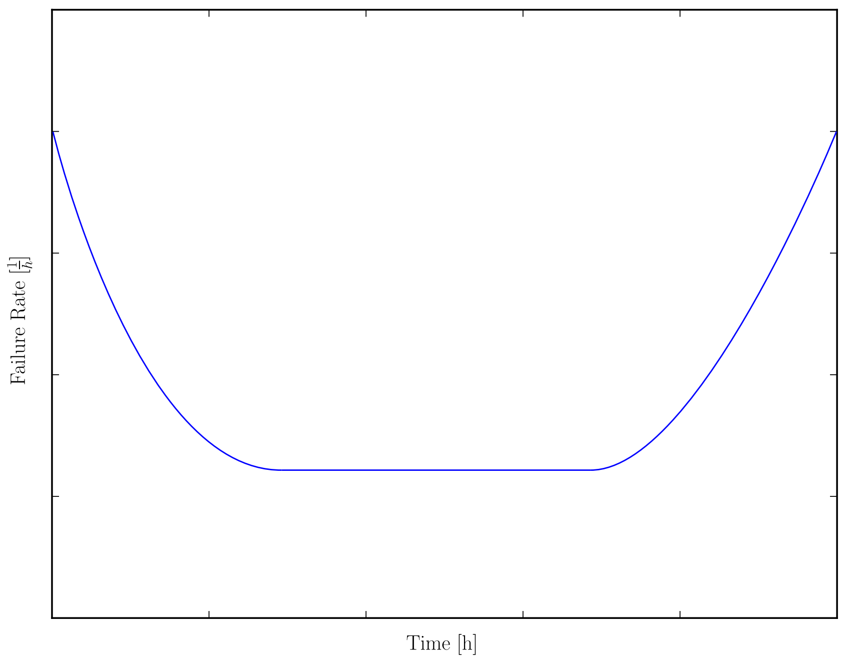


Figure 5.11: Traditional bathtub curve

The modification is based on an artificial idea that even during the "useful period" (flat part after infant mortality) a constant failure rate is unlikely. Fluctuations of the failure rate are still possible within this period. Sections of increased or decreased failure rate could occur. A visualization of this behavior can be seen in figure Figure 5.12.

The implementation of this idea used in this experiment is using a discreet interpretation of this curve. Figure 5.13 shows the failure rate for a set of parameters. The defining parameters can be seen in Table 5.5. This distribution is also used in the motivational case study (see section 3.3). All failure times for this experiment were self-generated based on this introduced distribution.

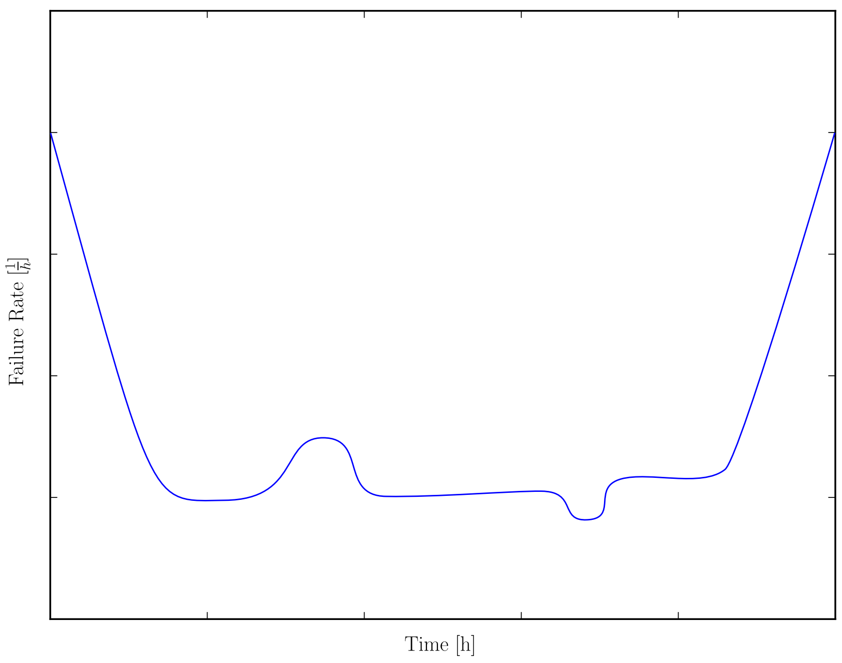


Figure 5.12: Modified bathtub curve

Point No.	Time [h]	Rate [h^{-1}]
#1	0	0.002
#2	100	0.00009
#3	600	0.00009
#4	620	0.0008
#5	850	0.0008
#6	900	0.0002
#7	6000	0.0002
#8	6050	0.000035
#9	7500	0.00035
#10	7550	0.0002
#11	9000	0.0002
#12	10000	0.003

Table 5.5: Used parameters in selfmade function

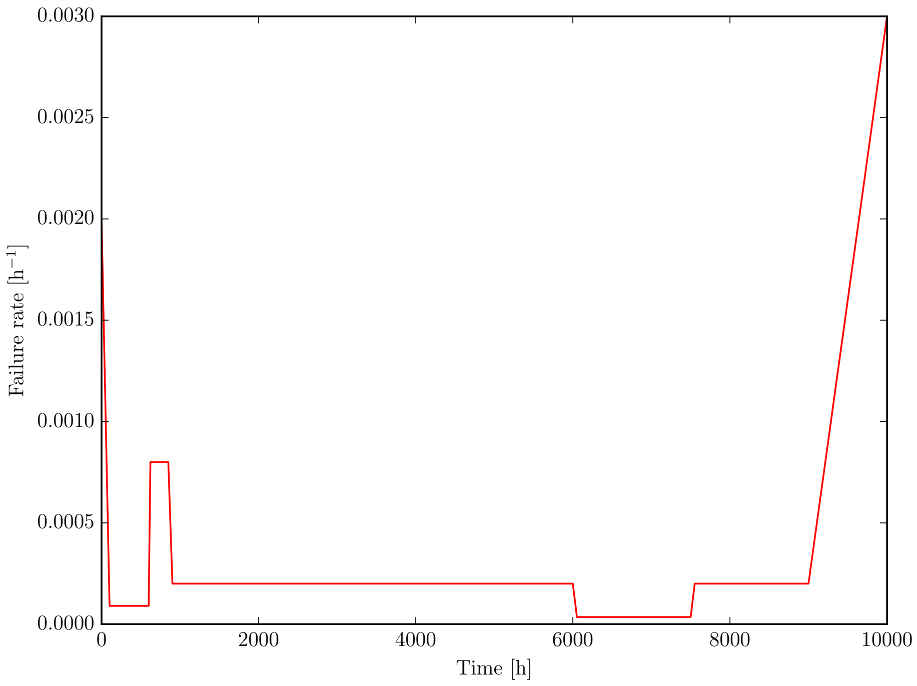


Figure 5.13: Selfmade distribution failure rate

5.5.2 Data characteristics

One thousand failure times are drawn based on this proposed distribution. All of them are listed as failures, no censoring. As the data is self-generated, the output is adjusted according to the required input of the processing tools. This is a CSV text-file with one line per event. The first column represents the event time and the second column the type (failure or censoring).

An excerpt of the file can be seen in Table 5.6.

Event Time	Type
9.150000000000000000e+02	1
9.370000000000000000e+02	1
9.510000000000000000e+02	1
9.580000000000000000e+02	1
9.660000000000000000e+02	1
9.810000000000000000e+02	1
1.002000000000000000e+03	1
1.003000000000000000e+03	1

Table 5.6: Excerpt of structure of the input data used for experiment 3

5.5.3 Data processing

5.5.3.1 Parametric

The estimator for a constant failure rate for the data can be seen in Equation 5.5. A plot showing the reliability based on this value in comparison with the true reliability is given in Figure 5.14.

$$\lambda_1 = \frac{\text{Number of failures}}{\text{Aggregated time in service}} = \frac{1000}{3612389} = 2.77 \times 10^{-4} \frac{1}{\text{h}} \tag{5.5}$$

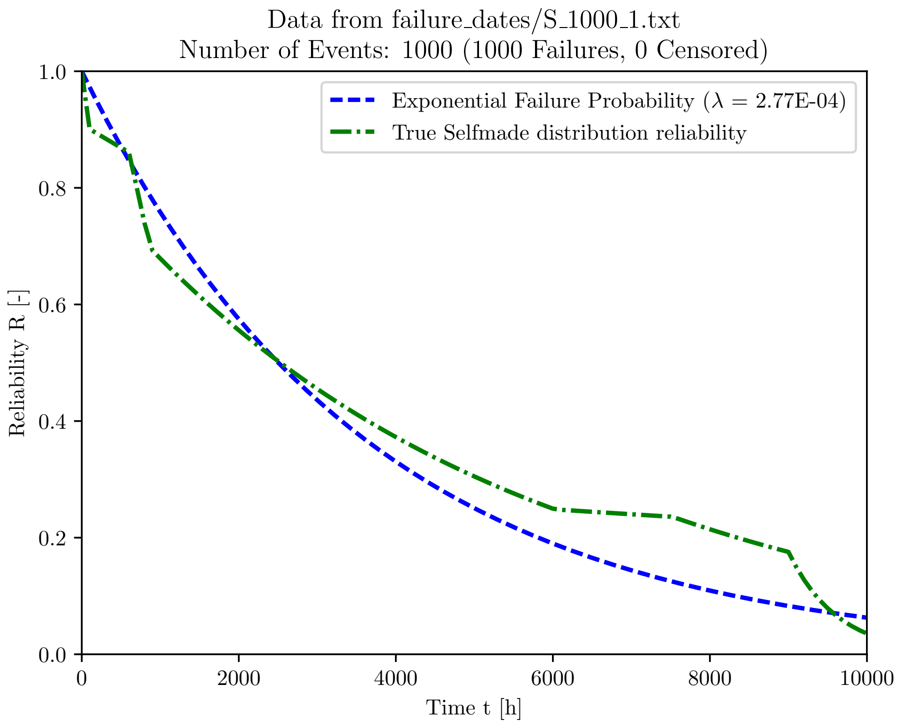


Figure 5.14: Third Dataset: Exponentially assumed failure distribution in comparison to the true reliability

5.5.3.2 Empirical (Reduced Kaplan-Meier)

Applying the reduced Kaplan-Meier algorithm to the data with a chosen threshold value of 0.0005 gives a representation with 11 segments. The resulting model can be seen in Figure 5.15.

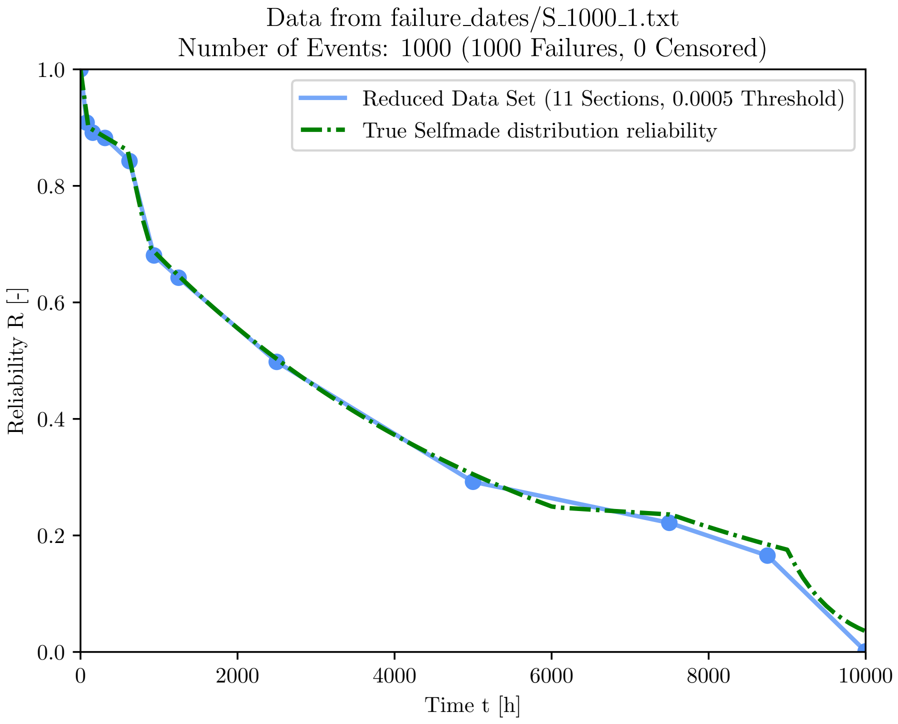


Figure 5.15: Third Dataset: Reduced Kaplan-Meier in comparison to the true reliability

5.5.3.3 Empirical (Polynomial)

Applying the polynomial regression to this data set gives Equation 5.6. The resulting plot is seen in Figure 5.16 in comparison with the true reliability distribution.

$$R(t) = 0.9556 - 2.703 \times 10^{-4}t + 3.714 \times 10^{-8}t^2 - 1.669 \times 10^{-12}t^3 - 3.016 \times 10^{-17}t^4 \quad (5.6)$$

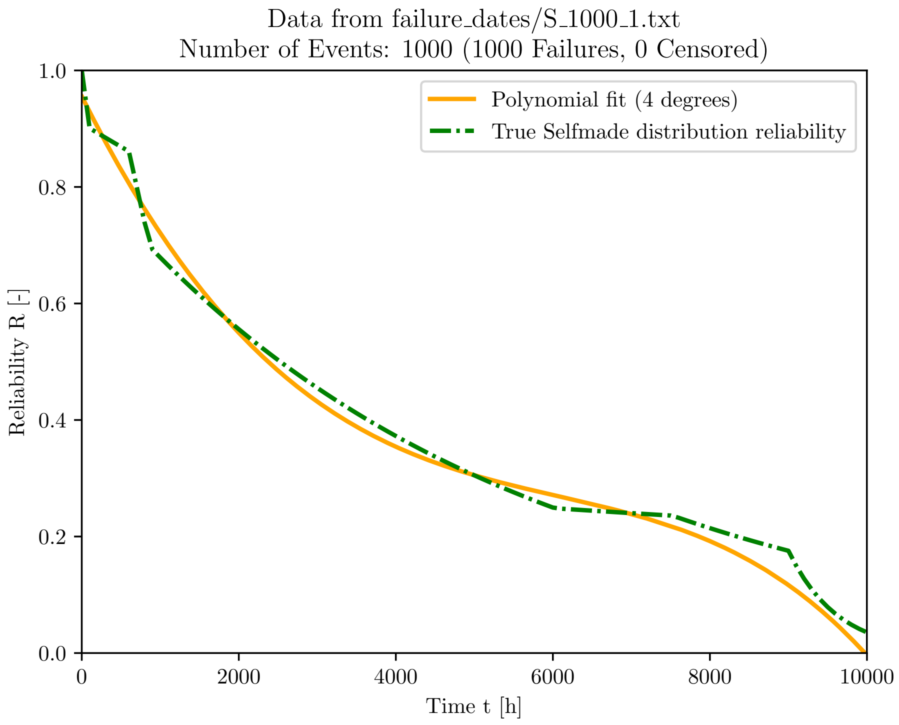


Figure 5.16: Third Dataset: Polynomial regression in comparison to the true reliability

5.5.4 Accuracy comparison

An illustration of all estimator shapes compared to the true reliability is found in Figure 5.17. It is visible that the exponential distribution is not able to match the unique characteristics of the underlying failure distribution.

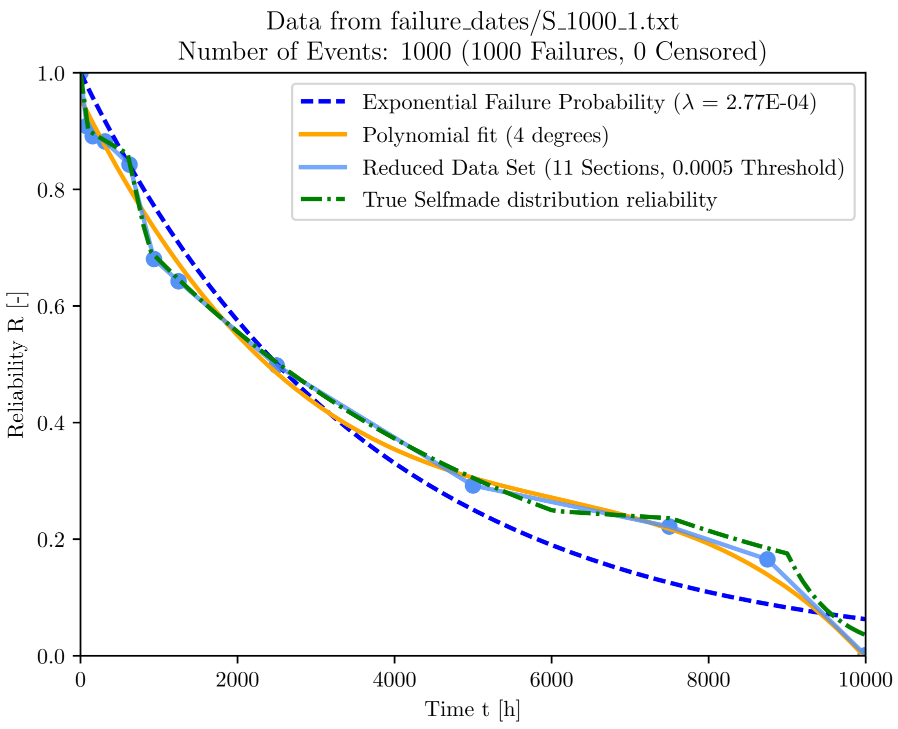


Figure 5.17: Third Dataset: Comparison of exponential, reduced Kaplan-Meier, polynomial regression and the true reliability diagram

A quantitative comparison of accuracy is given in Table 5.7. The shape of the true reliability characteristics of the underlying distribution cannot be matched with a constant failure rate (1,106 > 0,351). The reduced Kaplan-Meier however is sufficient for doing so (as 0,164 < 0,351). Utilizing a polynomial regression gives an estimator which is less accurate, but still sufficient (0,288 < 0,351).

	Value
Reduced Kaplan-Meier vs. True reliability	0,164
Polynomial regression vs. True reliability	0,288
Exponential distribution vs. True reliability	1,106
Critical Kolmogorov-Smirnov test value	0,351

Table 5.7: Comparison of Kolmogorov-Smirnov test results for the third data set

5.6 Fourth data set: Dynamic reliability analysis

5.6.1 Data origin

Based on the work of Manno et. al. [15], a failure distribution according to a dynamic reliability analysis is proposed. In their example a cooling unit is more likely to fail when used. It still has a small probability of failure due to aging even when not used. During the year, the change of weather is also changing the demand to use this cooling unit.

A simplified adaption of this behavior is implemented in a new distribution model. Piecewise linear decreasing reliability is defined for sections of usage and non-usage. Different rates are used for each.

The reliability distribution of three years of operation can be seen in Figure 5.18.

During the first half of a year (4380 hours), the reliability drops by 0.3 (Time in which the unit is used). In the following second half of the year, the reliability further drops by 0.033 (Time in which the unit is not used). At the end of a three-year cycle, the reliability reaches 0.

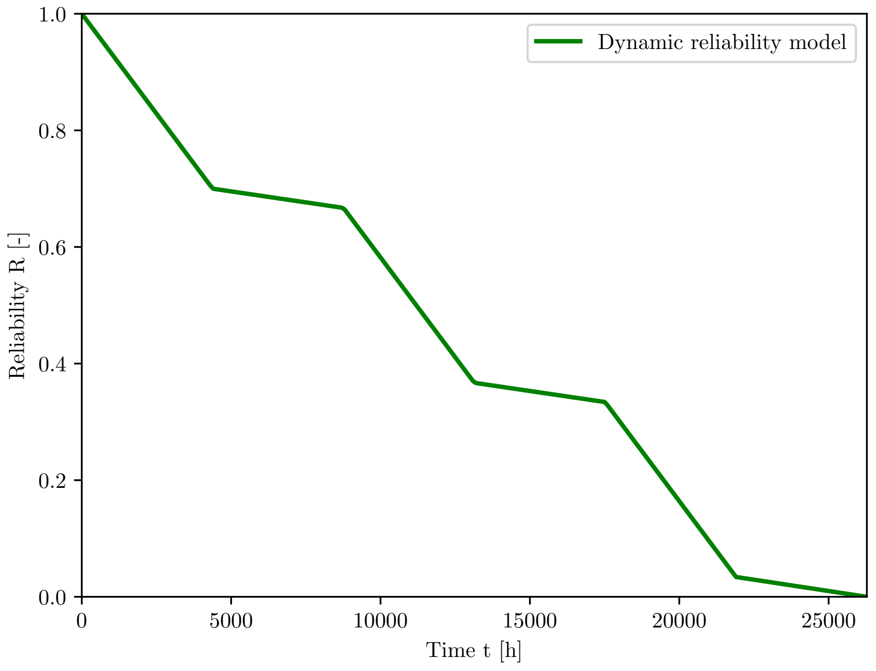


Figure 5.18: Dynamic reliability model

5.6.2 Data characteristics

As this dataset is self-generated, the data characteristic is similar to the 3rd experiment (see subsection 5.5.2). Only 100 failure dates are drawn from the distribution.

5.6.3 Data processing

5.6.3.1 Parametric

The estimator for a constant failure rate for the data can be seen in Equation 5.7. A plot showing the reliability based on this value is given in Figure 5.19.

$$\lambda_1 = \frac{\text{Number of failures}}{\text{Aggregated time in service}} = \frac{100}{1126397} = 8.88 \times 10^{-5} \frac{1}{\text{h}} \tag{5.7}$$

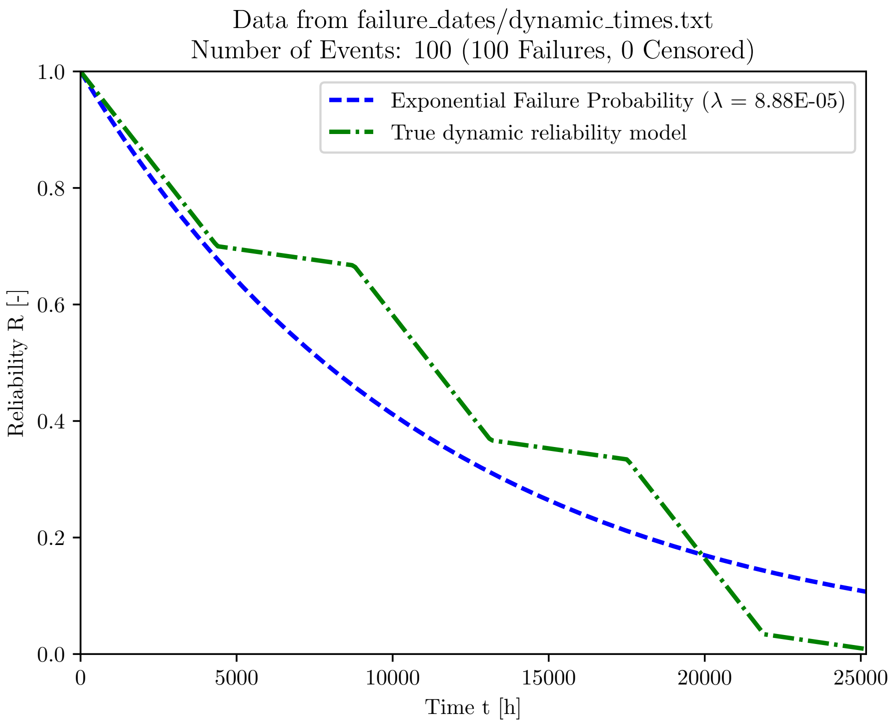


Figure 5.19: Fourth Dataset: Exponentially assumed failure distribution in comparison to the true reliability

5.6.3.2 Empirical (Reduced Kaplan-Meier)

Applying the reduced Kaplan-Meier algorithm to the data with a chosen threshold value of 0.0005 gives a representation with 7 segments. The resulting model can also be seen in Figure 5.20.

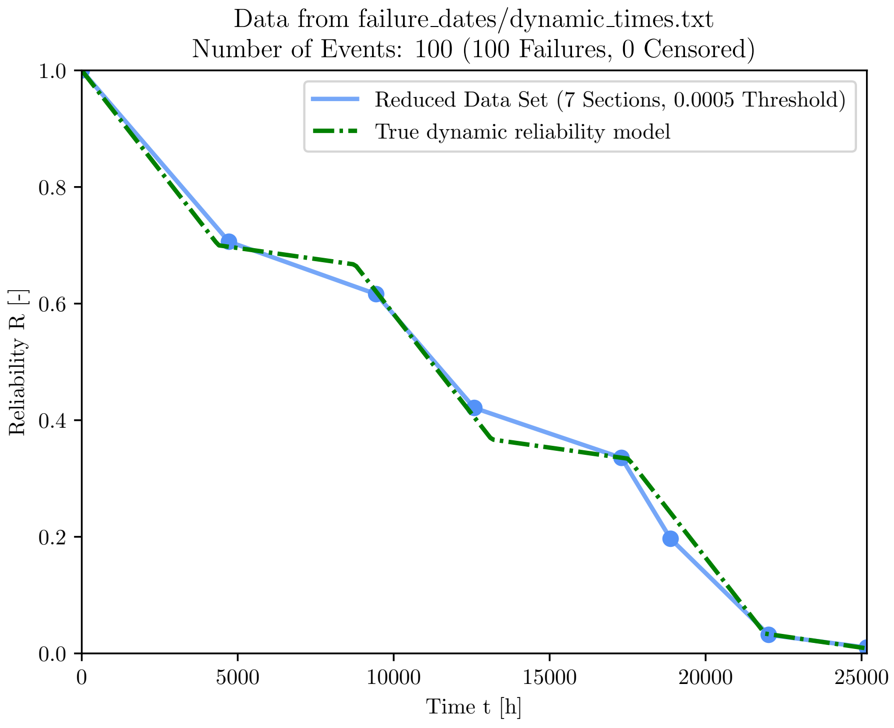


Figure 5.20: Fourth Dataset: Reduced Kaplan-Meier in comparison to the true reliability

5.6.3.3 Empirical (Polynomial)

Applying the polynomial regression to this data set returns Equation 5.8. The resulting reliability estimator is seen in Figure 5.21 in comparison with the true reliability distribution.

Having such a high number of resulting polynomial terms reveals problems. This estimator does not start at R(0) = 1 and at the very end a slight increase of the values can be seen, thus conflicting with the continuously decreasing nature of reliability.

$$\begin{aligned} R(t) =& 1.129 - 4.142 \times 10^{-4} t + 3.127 \times 10^{-7} t^2 \\ & - 1.363 \times 10^{-10} t^3 + 3.306 \times 10^{-14} t^4 - 4.725 \times 10^{-18} t^5 \\ & + 4.136 \times 10^{-22} t^6 - 2.242 \times 10^{-26} t^7 + 7.331 \times 10^{-31} t^8 \\ & - 1.325 \times 10^{-35} t^9 + 1.017 \times 10^{-40} t^{10} \end{aligned} \tag{5.8}$$

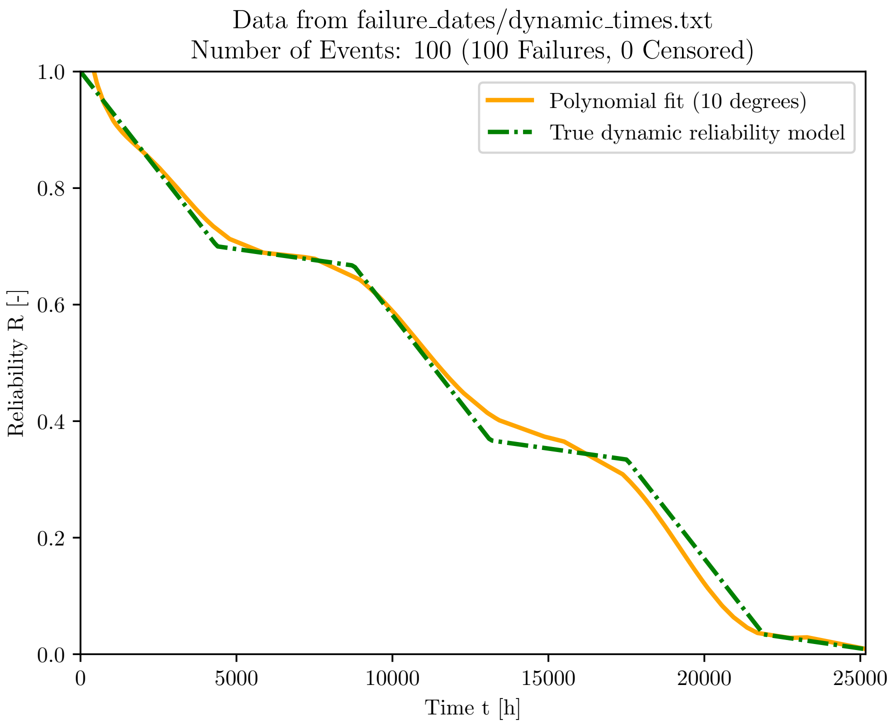


Figure 5.21: Fourth Dataset: Polynomial regression

5.6.4 Accuracy comparison

A visual comparison of the true dynamic reliability model, the reduced Kaplan-Meier, the polynomial estimator and the exponential estimator can be seen in Figure 5.22.

The constant failure rate is not able to reveal the changes throughout the observed three year period.

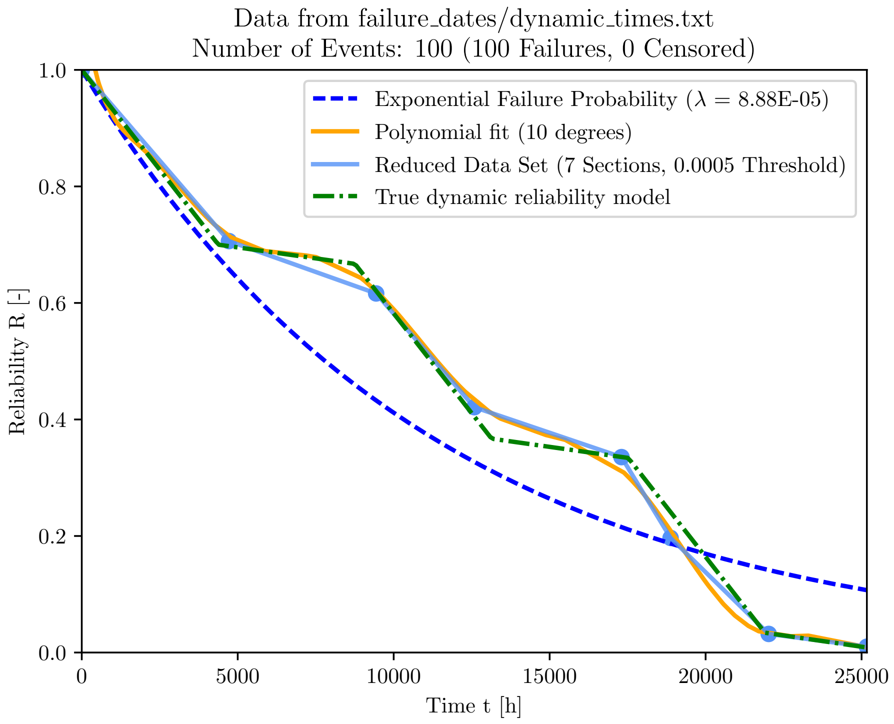


Figure 5.22: Fourth Dataset: Comparison of exponential estimator, reduced Kaplan-Meier, polynomial regression empirical and the true reliability diagram

A quantitative comparison of accuracy is given in Table 5.8. The shape of the true reliability characteristics of the underlying distribution cannot be matched with a constant failure rate (1,451 > 0,351). The reduced Kaplan-Meier however is sufficient for doing so (as 0,125 < 0,351). Less accurate but still sufficient is the estimator based on polynomial regression (0,239 < 0,351).

	Value
Reduced Kaplan-Meier vs. True reliability	0,125
Polynomial regression vs. True reliability	0,239
Exponential distribution vs. True reliability	1,451
Critical Kolmogorov-Smirnov test value	0,351

Table 5.8: Comparison of Kolmogorov-Smirnov test results for the fourth data set

5.7 Data requirements

The second dataset, as discussed in section 5.4, is the only experiment which yielded sufficient accurate reliability results for the parametric exponential estimation and both proposed empirical estimators.

The accuracy of both parametric and empirical reliability models is depending on the amount of input data. In case of a self-generated dataset, the number of failure dates is unlimited. Real reliability assessments however require the availability of real data. In order to evaluate the required amount of failure dates, a simulation is carried out.

A number N of failure dates is evaluated. In case of the harddisk dataset, those N failure dates are randomly drawn from the full dataset.

Based on those failure dates, the reliability estimators are calculated. Those may be the smoothend Kaplan-Meier, the polynomial fit and the constant failure rate. The reduced Kaplan-Meier is not considered, as its computational implementation made it not feasible to be included in the Monte-Carlo simulation.

The resulting reliability is then compared to the reference as described in section 4.6.

The result of the Kolmogorov-Smirnov test is stored together with the number of failure dates (N).

This procedure is repeated for $N+1$ failure dates. The maximum amount N_{max} of used failure dates is set to 6000 Each set of simulations from N to N_{max} is carried out 100 times. Afterwards the mean Kolmogorov-Smirnov test value is calculated for each N.

This gives an information how many samples in average have to be considered to give sufficient results.

In average minimum 380 failure dates are required to produce a smoothed Kaplan-Meier estimator, which is sufficiently accurate. To retrieve an accurate estimator based on a constant failure rate, in average minimum 333 failure dates have to be assessed. Using the polynomial regression, 355 failure dates have to be processed to give a satisfying result.

A graphical illustration of the Monte-Carlo simulation forcan be seen in Figure 5.23. It shows the how the Kolmogorov-Smirnov test value is decreasing with the increasing number of N. The green horizontal line indicates the critical value of the Kolmogorov-Smirnov test (0,351).

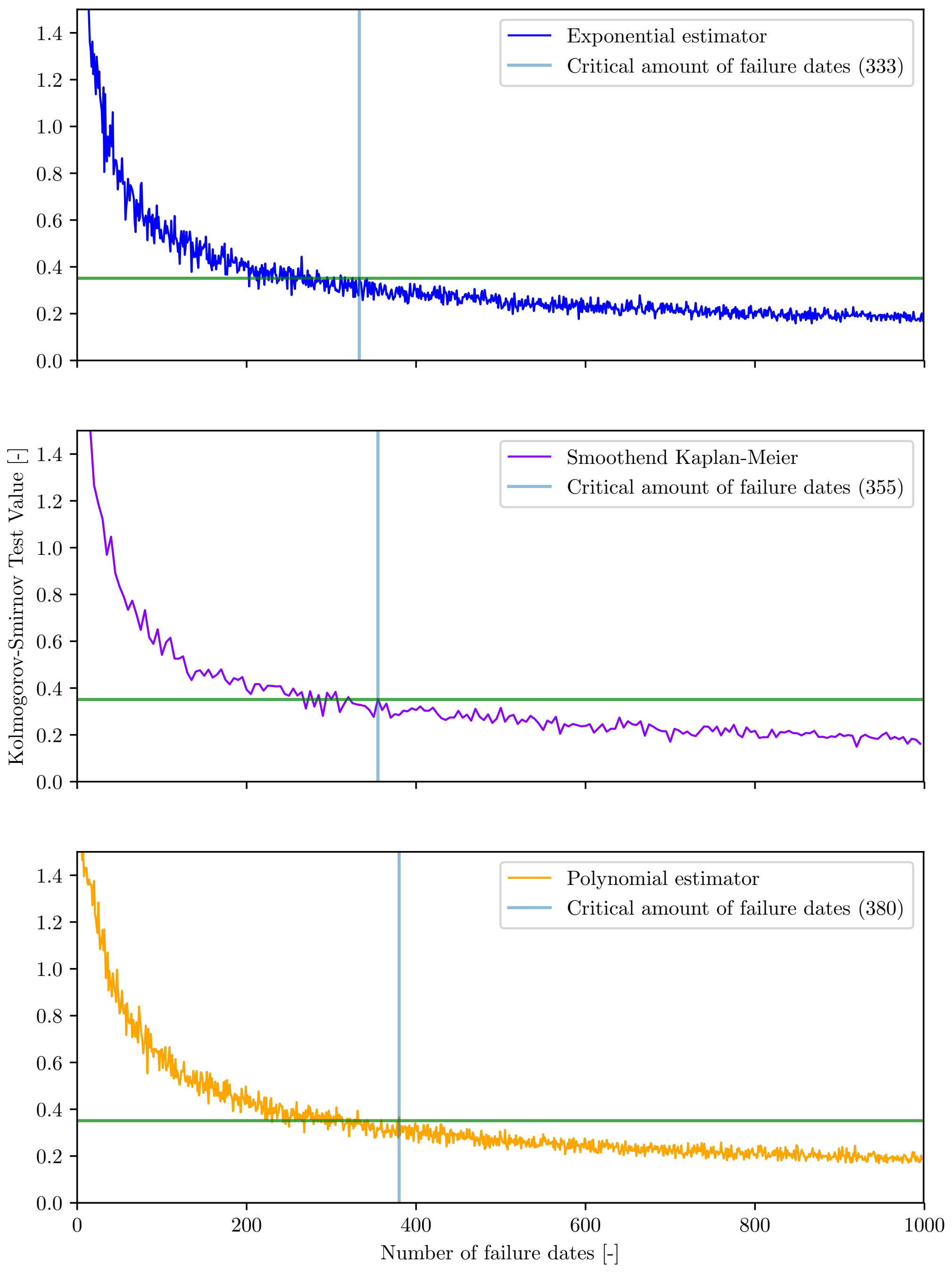


Figure 5.23: Comparison of data requirements for all three estimators

Chapter 6

Conclusion & Future Works

Based on the results for each of the four experiments, several conclusions can be drawn. With a given number failure dates, empirical distributions reveal more precise insights into the failure behavior. This is valid for all of the four presented experiments.

The exponential distribution is only sufficient enough for one of those (2^{nd} experiment, see section 5.4).

All of the calculated empirical estimators are superior in their accuracy, both from a qualitative visual evaluation and supported by a quantitative comparison.

According to the computational efficiency evaluation (see section 4.7), the processing time to retrieve reliability values for those estimators is negligible short.

Both parts of the experiment hypothesis could be validated on each of the four experiments: The empirical probability distributions do match sufficiently well the reference reliability and the computational efficiency is sufficiently good.

Implementing the proposed estimators as generic input of RAMS tools is possible and would be beneficial for reliability assessments.

Assessing the amount of required data to achieve the demonstrated accuracy levels revealed that, for the given dataset, each method requires a similar amount of data to be sufficiently accurate.

During the work on this thesis, several challenges and unsolved issues got revealed. These are to be addressed in future works.

- **Access to OREDA/ExproSoft database**
  Both OREDA and ExproSoft provide online databases for reliability data. All collected reliability data is available in those with rich detail level. Their policy however does not allow academic institutes to gain access to the full data. The underlying raw data is a useful resource to test and optimize empirical algorithms and compare them with the provided λ estimates.

- **Extend to more applications**
  The comparison of empirical and parametric methods is a generic issue. The work in this thesis is focusing on simple reliability estimation. In the given RAMS context, this discussion can also be extended to maintenance models or risk quantification.

- **Improve the algorithm for data reduction**
  The implementation for data-reduction done for this thesis is rather simple. It is not efficient and does give the best possible results. Setting the threshold value is cumbersome and the removal of unnecessary points is frequently omitting redundant points. Developing an approximation algorithm is not scope of this work. Implementing a well defined and tested model will be beneficial.

- **Improve polynomial fit**
  The polynomial regression as used throughout this thesis is a very basic mathematical implementation. It does not respect the limitations and assumptions of a reliability distribution. Implementing these (Always start at R(0) = 1, monotonous decreasing) would greatly improve the applicability.

- **Implementation into RAMS tools**
  RAMS engineers do not rely on academic examples and hand-calculations. Numerous software tools for RAMS related assessments are available. Further review is necessary to get an overview over the different tools and their current functionality in terms of empirical probability distributions. An implementation of the proposed method would allow to conduct comparing experiments.

- **Implementation of uncertainty**
  Having an understanding of uncertainty and confidence intervals is a critical criteria in assessing and comparing reliability information. There are measures to identify the confidence interval for the Kaplan-Meier estimator. These should be tested and expanded for the reduced Kaplan-Meier estimator and similar for the polynomial estimator. Some mathematical metrics exist to evaluate the quality of a polynomial regression, and these should be adapted to the field of reliability

Bibliography

[1] Backblaze [2017], ‘Hard drive test data - determining failure rates and more’. Accessed: 2017-05-25.
URL: `https://www.backblaze.com/b2/hard-drive-test-data.html`

[2] Bae, W., Choi, H., Park, B. U. and Kim, C. [2005], ‘Smoothing techniques for the bivariate kaplan–meier estimator’, *Communications in Statistics - Theory and Methods* **34**(7), 1659–1674.

[3] Bobrowski, S., Chen, H., Döring, M., Jensen, U. and Schinköthe, W. [2015], ‘Estimation of the lifetime distribution of mechatronic systems in the presence of a covariate: A comparison among parametric, semiparametric and nonparametric models’, *Reliability Engineering & System Safety* **139**, 105–112.

[4] Camponogara, E. and Nazari, L. F. [2015], ‘Models and algorithms for optimal piecewise-linear function approximation’, *Mathematical Problems in Engineering* **2015**, 1–9.
URL: *https://doi.org/10.1155%2F2015%2F876862*

[5] *Critical Values for the Two-sample Kolmogorov-Smirnov test (2-sided)* [n.d.]. Accessed: 2017-06-02.
URL: `https://www.webdepot.umontreal.ca/Usagers/angers/MonDepotPublic/STT3500H10/Critical_KS.pdf`

[6] ExproSoft [2017], ‘Exprosoft - about us’. Accessed: 2017-05-07.
URL: `http://www.exprosoft.com/about-us/`

[7] ISO 14224:2016 [2016], Petroleum, petrochemical and natural gas industries – Collection and exchange of reliability and maintenance data for equipment, Standard, International Organization for Standardization, Geneva.

[8] Jabot, F. [2015], ‘Why preferring parametric forecasting to nonparametric methods?’, *Journal of Theoretical Biology* **372**, 205–210.

[9] Kaplan, E. L. and Meier, P. [1958], ‘Nonparametric estimation from incomplete observations’, *Journal of the American Statistical Association* **53**(282), 457–481.
URL: *http://dx.doi.org/10.1080/01621459.1958.10501452*

[10] Kim, C., Park, B., Kim, W. and Lim, C. [2003], ‘Bezier curve smoothing of the kaplan-meier estimator’, *Annals of the Institute of Statistical Mathematics* **55**(2), 359–367.

[11] Klutke, G., Kiessler, P. and Wortman, M. [2003], 'A critical look at the bathtub curve', *IEEE Transactions on Reliability* **52**(1), 125–129.
URL: *http://dx.doi.org/10.1109/TR.2002.804492*

[12] Lazarus, R. [2016*a*], 'Hard drive test data - determining failure rates and more'. Accessed: 2017-05-25.
URL: `https://github.com/fubar2/backblazeKM`

[13] Lazarus, R. [2016*b*], 'Survival analysis of hard disk drive failure data'. Accessed: 2017-05-25.
URL: `http://bioinformare.blogspot.no/2016/02/survival-analysis-of-hard-disk-drive.html`

[14] Lewis, E. E. [1995], *Introduction to Reliability Engineering*, Wiley.

[15] Manno, G., Zymaris, A., Kakalis, N. M. P., Chiacchio, F., Cipollone, F. E., Compagno, L., Urso, D. D. and Trapani, N. [2013], *Dynamic reliability analysis of three nonlinear aging components with different failure modes characteristics*, CRC Press, pp. 3047–3055. doi:10.1201/b15938-457.
URL: *http://dx.doi.org/10.1201/b15938-457*

[16] Mokhtarian, P., Namzi-Rad, M. R., Ho, T. K. and Suesse, T. [2013], Bayesian nonparametric reliability analysis for a railway system at component level, IEEE Computer Society, pp. 197–202.

[17] OREDA [2017], 'Oreda - about us'. Accessed: 2017-05-07.
URL: `https://www.oreda.com/about-us/`

[18] OREDA (project), SINTEF, NTNU. [2015*a*], *OREDA : offshore and onshore reliability data handbook : Vol. 1 : Topside equipment*, OREDA Participants, Norway.

[19] OREDA (project), SINTEF, NTNU. [2015*b*], *OREDA : offshore and onshore reliability data handbook : Vol. 2 : Subsea equipment*, OREDA Participants, Norway.

[20] PDS Forum [2017], 'PDS - Reliability of Safety Instrumented Systems'. Accessed: 2017-05-22.
URL: `http://www.sintef.no/pds`

[21] Perretti, C. T. and Munch, S. B. [2015], 'On estimating the reliability of ecological forecasts'.

[22] Pons, O. [2013], *Statistical Tests of Nonparametric Hypotheses : Asymptotic Theory*, STATISTICAL TESTS OF NONPARAMETRIC HYPOTHESES, World Scientific Publishing Company, Singapore.

[23] Python Software Foundation [2017], 'timeit — measure execution time of small code snippets'. Accessed: 2017-06-09.
URL: `https://docs.python.org/3/library/timeit.html`

[24] Rausand, M. [2004], *System reliability theory : Models, statistical methods, and applications*, Wiley-Interscience, Hoboken, NJ.

[25] Solfrid Håbrekke, Stein Hauge, T. O. [2013], Reliability data for safety instrumented systems : PDS data handbook, Sintef rapport, SINTEF Technology and Society, Safety Research, Trondheim.

[26] Spjøtvoll, E. [1985], 'Estimation of failure rate from reliability data bases'.

[27] The SciPy community [2017*a*], 'numpy.interp'. Accessed: 2017-05-15.
URL: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.interp.html`

[28] The SciPy community [2017*b*], 'numpy.poly1d'. Accessed: 2017-06-05.
URL: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.poly1d.html`

[29] The SciPy community [2017*c*], 'numpy.polyfit'. Accessed: 2017-06-05.
URL: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html`

[30] Youndjé, E. [2016], 'Some heuristics about bandwidth selection for the smooth kaplan–meier estimator', *Journal of the Korean Statistical Society* **45**(4), 568 – 580.
URL: *http://www.sciencedirect.com/science/article/pii/S1226319215301344*

Appendix A

Appendix

A.1 Additional information for the case study

A.1.1 Failure rate and times

Example calculation for λ_A:

$$\lambda_A = \frac{\text{Number of failures for component A}}{\text{Aggregated time in service for component A}} = 4.11 \times 10^{-4}[\frac{1}{h}] \tag{A.1}$$

The failure data for component "A" is drawn from a Weibull distribution with shape-parameter $\alpha = 1.3$ and scale-paramter $\lambda = 0.0005$ based on equation Equation A.2. The censoring is done by a uniform randomization.

$$R(t) = e^{-(\lambda t)^{\alpha}} \tag{A.2}$$

The failure data for component "B" is drawn from an exponential distribution with failure rate $\lambda = 0.0005$ based on equation Equation A.3 The censoring is done by a uniform randomization.

$$R(t) = e^{-(\lambda t)} \tag{A.3}$$

The failure data for component "C" is drawn from a self created distribution based on a modified bathtub-shaped failure rate. More information on this distribution is given in section 5.5. The censoring is done by a uniform randomization.

A table of all drawn values can be found in Figure A.1.

```
[[  139.     1.]
 [  255.     1.]
 [  339.     0.]
 [  445.     1.]
 [  464.     0.]
 [  464.     1.]
 [  621.     1.]
 [  621.     0.]
 [  701.     0.]
 [  701.     1.]
 [  776.     1.]
 [  875.     0.]
 [ 1094.     1.]
 [ 1260.     1.]
 [ 1321.     1.]
 [ 1430.     0.]
 [ 1486.     0.]
 [ 1645.     1.]
 [ 1989.     1.]
 [ 2172.     1.]
 [ 2283.     0.]
 [ 2309.     1.]
 [ 2504.     1.]
 [ 2566.     1.]
 [ 2709.     0.]
 [ 3136.     1.]
 [ 3186.     1.]
 [ 3259.     1.]
 [ 3291.     0.]
 [ 4639.     1.]]
```

(a) Component A

```
[[   92.     1.]
 [  147.     1.]
 [  165.     0.]
 [  166.     1.]
 [  313.     1.]
 [  371.     1.]
 [  507.     0.]
 [  575.     1.]
 [  576.     1.]
 [  612.     1.]
 [  660.     1.]
 [  689.     0.]
 [  908.     1.]
 [  965.     1.]
 [ 1045.     1.]
 [ 1110.     1.]
 [ 1177.     0.]
 [ 1341.     1.]
 [ 1607.     1.]
 [ 1665.     0.]
 [ 1717.     1.]
 [ 1955.     1.]
 [ 2330.     1.]
 [ 2391.     1.]
 [ 2391.     0.]
 [ 2458.     0.]
 [ 2556.     1.]
 [ 2769.     1.]
 [ 3595.     1.]
 [ 4151.     0.]
 [ 5302.     1.]
 [ 5307.     1.]
 [ 5315.     1.]
 [ 6248.     0.]
 [ 7030.     1.]]
```

(b) Component B

```
[[    16.     1.]
 [    33.     0.]
 [    35.     1.]
 [    62.     0.]
 [    64.     0.]
 [    66.     1.]
 [   623.     0.]
 [   638.     1.]
 [   640.     1.]
 [   651.     0.]
 [   662.     0.]
 [   716.     1.]
 [   747.     0.]
 [   821.     0.]
 [   830.     1.]
 [  2099.     0.]
 [  2117.     0.]
 [  2607.     1.]
 [  2625.     0.]
 [  2838.     1.]
 [  5111.     1.]
 [  5135.     0.]
 [  5543.     0.]
 [  5769.     1.]
 [  6004.     0.]
 [  6796.     0.]
 [  9376.     1.]
 [ 10256.     0.]
 [ 11649.     0.]
 [ 12008.     1.]]
```

(c) Component C

Figure A.1: Used failure dates for all three component types in the case study

A.1.2 Calculation of Top-Event probability

The TOP Event probability is calculated using standard textbook [24] methodology, not using the approximation. Listing A.1 shows the Python code used to calculate the probability.

```python
A_System_Probability = A_Probability * A_Probability

B_System_Probability = B_Probability

C_SubSystem_Probability = C_Probability * C_Probability

C_System_Probability = 1 - ((1 - C_SubSystem_Probability) * (1 -
    C_SubSystem_Probability) * (1 - C_SubSystem_Probability))

TOP_Probability = 1 - ((1 - A_System_Probability) * (1 - B_System_Probability)
    * (1 - C_System_Probability))
```

Listing A.1: Calculation of probability of TOP event

A.2 Python program source code

```python
# Import numerical python module with short handle np
import numpy as np
import sys
# Import math module
import math
import random
# Import Plotting
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.offsetbox import AnchoredText
from matplotlib.pyplot import cm
import matplotlib.patches as patches
import matplotlib.lines as lines
# Import OS related module
import os
import glob
import generator
from scipy import optimize
from scipy import stats
from decimal import Decimal
from statistics import mean
import dynamic_reliability

np.set_printoptions(suppress=True, linewidth=1000)

SelfmadeData = generator.Florian(100.0,        # t1
600.0,                          # t2
620.0,                          # t3
850.0,                          # t4
900.0,                          # t5
6000.0,                         # t6
6050.0,                         # t7
7500.0,                         # t8
7550.0,                         # t9
9000.0,                         # t10
10000.0,                        # t11
0.002,                          # z1
0.00009,                        # z2
0.0008,                         # z3
0.0002,                         # z4
0.000035,                       # z5
0.0002,                         # z6
0.003)                          # z7

DynamicParameterArray = np.array([  [0,0],
                                    [4380,0.300],
                                    [8760,0.333],
                                    [13140,0.633],
                                    [17520,0.666],
                                    [21900,0.966],
                                    [26280,1.0]
                               ])

TrueDynamic = dynamic_reliability.Dynamic(DynamicParameterArray)

class Dataset:

    def __init__(self,filename,failure_indicator,censor_indicator,timebase,NumberOfReducedSections,ExpansionFactor,
       ReductionThreshold,PolyFitDegrees,IsNumpyArray,ArrayName):
        # Filename of the dataset
        self.filename = filename
        self.IsNumpyArray = bool(IsNumpyArray)
        self.ArrayName = ArrayName
        # What are the Censor and Failure indicators?
        self.censor_indicator = censor_indicator
        self.failure_indicator = failure_indicator

        # What is the timebase?
        # May be days or years
        # Standard is "hours"
        self.timebase = timebase

        # Prepare the variable for the maximum/highest event time based on first column of KaplanMeierArray
        self.MaximumEventTime = 0

        # Number Of Sections for the reduced section method
        self.NumberOfReducedSections = NumberOfReducedSections

        # Degrees for PlotPolyFit
        self.PolyFitDegrees = PolyFitDegrees

        # Factor for prediction time line Expansion
        self.ExpansionFactor = ExpansionFactor

        # Threshold for section reduction
        self.ReductionThreshold = ReductionThreshold
        self.AmountOfAutomaticallyFoundSegments = 0

        # Load "filename" into numpy array of name RawDataArray


        if self.IsNumpyArray == False:
            try:
                self.RawDataArray = np.loadtxt(str(self.filename), delimiter=";")
            except:
                sys.stderr.write('Unable to open file "%s"\n' % filename)
                sys.stderr.flush()
                exit()
                return
        if self.IsNumpyArray == True:
            self.RawDataArray = self.ArrayName

        # If timebase is hours, do nothing

        if self.timebase == "days":
            self.RawDataArray[:,0] *= 24

        if self.timebase == "years":
            self.RawDataArray[:,0] *= 24*365.25

        # Highest observed event time
        self.MaximumEventTime = np.max(self.RawDataArray[:,0])

        # Execute KaplanMeier() and place resulting array in GlobalKaplanMeierArray variable for future common use
        self.GlobalKaplanMeierArray = self.KaplanMeier()
        self.GlobalKaplanMeierArrayWithoutCensored = self.KaplanMeierArrayWithoutCensored()
        self.GlobalReducedArray = self.ReducedDataIntelligent(verbose=0)

        self.EstimatedLambda = self.EstimateLambda()

    def GetNumberOfFailures(self):

        # Read RawDataArray, only select the parts of the array where the second column (censoring value) is equal to Zero
        # This equals a Failure. Then give out the shape of the array and from that information take the first value. This is
       the amount of rows.

        NumberOfFailures = self.RawDataArray[self.RawDataArray[:, 1] == self.failure_indicator].shape[0]
        return NumberOfFailures

    def GetNumberOfCensored(self):

        # Read RawDataArray, only select the parts of the array where the second column (censoring value) is equal to One
        # This equals a Censoring. Then give out the shape of the array and from that information take the first value. This is
        the amount of rows.

        NumberOfCensored = self.RawDataArray[self.RawDataArray[:, 1] == self.censor_indicator].shape[0]
        return NumberOfCensored

    def GetNumberOfEvents(self):

        # Read RawDataArray. Then give out the shape of the array and from that information take the first value. This is the
       amount of rows.

        NumberOfFailures = self.RawDataArray.shape[0]
        return NumberOfFailures

    def AccumulatedServiceTime(self):

        # Sum up all the event times.
        AccumulatedServiceTime = np.sum(self.RawDataArray, axis=0)[0]
        return AccumulatedServiceTime

    def EstimateLambda(self):

        # Estimate Lambda based on the approach 'Number of Failures' / 'Accumulated Service Time'

        NumberOfFailures = self.GetNumberOfFailures()
        AccumulatedServiceTime = self.AccumulatedServiceTime()
        EstimatedLambda = NumberOfFailures / AccumulatedServiceTime

        return EstimatedLambda

    def GetUnworthySegments(self,ReducedArray,verbose=0):
        self.ReducedArray = ReducedArray
        self.verbose = 0

        CheckerNumber = 10

        UnworthySegments = []
        NumberOfSegments = int(ReducedArray.shape[0]-1)

        if self.verbose == 1:
            print("Running GetUnworthySegments")
            print("Number of Segments in Reduced Array: " + str(NumberOfSegments))
            print("This is the ReducedArray to start with:")
            print(self.ReducedArray)
            print("")

        for i in range (1,NumberOfSegments+1):

            StartingTime = self.ReducedArray[i-1,0]
            EndTime = self.ReducedArray[i,0]
            TimeDifference = EndTime - StartingTime

            # Create CheckerArray.
            # n (CheckerNumber) values evenly distributed between starting and ending time of each Segments

            DifferenceSquareList = []
            CheckerArray = np.linspace(StartingTime,EndTime,CheckerNumber)

            if self.verbose == 1:
                print("Checking Segment number " + str(i))
                print("\tStartingTime: " + str(StartingTime))
                print("\tEndTime: " + str(EndTime))
                print("\tTimeDifference: " + str(TimeDifference))
                print("\tCheckerArray: " + str(CheckerArray))

            for c in np.nditer(CheckerArray):
                KM_SmoothValue = self.GetKaplanMeierValueSmooth(c)
                KM_ReducedValue = np.interp(c, self.ReducedArray[:,0], self.ReducedArray[:,-1])
                DifferenceSquare = (KM_SmoothValue - KM_ReducedValue)**2
                DifferenceSquareList.append(DifferenceSquare)
                if self.verbose == 1:
                    print("\t\tChecking time: " + str(c))
                    print("\t\tKM_SmoothValue: " + str(KM_SmoothValue))
                    print("\t\tKM_ReducedValue: " + str(KM_ReducedValue))
                    print("\t\tDifferenceSquare: " + str(DifferenceSquare))
                    print("\n")

            MeanOfSquares = mean(DifferenceSquareList)
            if self.verbose == 1:
                print("Mean of Squares for this segment:")
                print(MeanOfSquares)

            if MeanOfSquares > self.ReductionThreshold:
                UnworthySegments.append(i)
                if self.verbose == 1:
                    print("\t--> Not Worthy! (Larger than " + str(self.ReductionThreshold) + ")")
                    print("\tAdding " + str(i) + " to UnworthySegments List")
            else:
                if self.verbose == 1:
                    print("\t--> Worthy! (Smaller than " + str(self.ReductionThreshold) + ")")
                pass

        if self.verbose == 1:
            print("")
            print("List of unworthy segments: " + str(UnworthySegments))
            print("")

        return UnworthySegments

    def TreatSegments(self,ReducedArray,verbose=1):
        self.ReducedArray = ReducedArray
        self.verbose = 0

        if verbose == 1:
            print("Run TreatSegments")

        TreatedArray = self.ReducedArray
        ListOfSegmentsToBeTreated = []
        ListOfSegmentsToBeTreated = self.GetUnworthySegments(self.ReducedArray,verbose=0)

        if verbose == 1:
            print("ListOfSegmentsToBeTreated: " + str(ListOfSegmentsToBeTreated))

        for i in ListOfSegmentsToBeTreated:
            NewTime = int(((self.ReducedArray[i,0] - self.ReducedArray[i-1,0]) / 2) + self.ReducedArray[i-1,0])
            NewValue = self.GetKaplanMeierValueSmooth(NewTime)

            TreatedArray = np.vstack([TreatedArray, np.array([NewTime, NewValue])])

            if verbose == 1:
                print("Segment to work on (i): " + str(i))
                print("NewTime: " + str(NewTime))
                print("NewValue: " + str(NewValue))
                print("Added to TreatedArray")
                print("")

        TreatedArray = TreatedArray[TreatedArray[:,0].argsort()]

        if verbose == 1:
            print("This is the TreatedArray:")
            print(TreatedArray)
            print("")

        return TreatedArray

    def RemoveUnnecessaryPoints(self,ReducedArray,verbose=1):

        self.ReducedArray = ReducedArray

        if verbose == 1:
            print("Run RemoveUnnecessaryPoints...")

            print("Starting Array:")
            print(self.ReducedArray)


        AmountOfPoints = int(self.ReducedArray.shape[0])
        if verbose == 1:
            print("Amount of Points: " + str(AmountOfPoints))
            print("\n")
        ListOfIndicesToBeRemoved = []


        for i in range(0,AmountOfPoints-1):
            if verbose == 1:
                print("Iteration Number " + str(i) + ":")

            if i > 0 & i != AmountOfPoints-1:
                # Removing that Index
                self.ReducedArray = ReducedArray
                ArrayWithIndexRemoved = np.delete(self.ReducedArray, i, axis=0)

                ListOfUnworthySegmentsAfterRemoval = self.GetUnworthySegments(ArrayWithIndexRemoved)

                if len(ListOfUnworthySegmentsAfterRemoval) != 0:
                    # print("ListOfUnworthySegmentsAfterRemoval is not empty, so we keep that index")
                    pass
                else:
                    # print("ListOfUnworthySegmentsAfterRemoval is empty, so this index is not important. Will be added to the
        ListOfIndicesToBeRemoved")
                    ListOfIndicesToBeRemoved.append(i)
                    # print("")
            else:
                if verbose == 1:
                    print("Iteration Number is 0, do nothing")
                    print("")
            pass

        if verbose == 1:
            print("ListOfIndicesToBeRemoved: " + str(ListOfIndicesToBeRemoved))

        self.ReducedArray = np.delete(ReducedArray, (ListOfIndicesToBeRemoved), axis=0)

        if verbose == 1:
            print(self.ReducedArray)

        return self.ReducedArray

    def ReducedDataIntelligent(self,verbose=0):

        base, ext = os.path.splitext(str(self.filename))

        ThresholdName = str(self.ReductionThreshold).replace(".", "-")

        FileNameForIntelligentlyReducedArray = str(base) + "_Intelligent_" + str(ThresholdName) + "_Array.txt"

        try:
            IntelligentlyReducedArray = np.loadtxt(FileNameForIntelligentlyReducedArray)
            # print("Loaded IntelligentlyReducedArray from existing text file " + str(FileNameForIntelligentlyReducedArray))

        except:
            # print("Running ReducedDataIntelligent()...")

            # Initial KM Array, smoothened version
            KaplanMeierArray = self.KaplanMeierArrayWithoutCensoredSmooth()

            # Create initial Reduced Array.
            # First Time: 0                  First Value: 1
            # Last Time: MaximumEventTime    Last Value: Last known Value
            IntelligentlyReducedArray = np.array([0, 1])
            LastTime = self.MaximumEventTime
            LastValue = KaplanMeierArray[-1,-1]

            IntelligentlyReducedArray = np.vstack([IntelligentlyReducedArray, np.array([LastTime, LastValue])])

            if self.verbose == 1:
                print("Start creating reduced Array:")
                print(IntelligentlyReducedArray)
                print("")

            AmountOfUnworhtySegments = len(self.GetUnworthySegments(IntelligentlyReducedArray,verbose=0))

            if self.verbose == 1:
                print("AmountOfUnworhtySegments in the beginning (should be 1): " + str(AmountOfUnworhtySegments))
                print("")

            while AmountOfUnworhtySegments > 0:
                IntelligentlyReducedArray = self.TreatSegments(IntelligentlyReducedArray,verbose=0)
                AmountOfUnworhtySegments = len(self.GetUnworthySegments(IntelligentlyReducedArray))

            IntelligentlyReducedArray = self.RemoveUnnecessaryPoints(IntelligentlyReducedArray,verbose=0)

            np.savetxt(FileNameForIntelligentlyReducedArray, IntelligentlyReducedArray)

        self.AmountOfAutomaticallyFoundSegments = int(IntelligentlyReducedArray.shape[0]) - 1

        return IntelligentlyReducedArray

    def ReturnValueArray(self,TimeArray,Method):

        self.TimeArray = TimeArray.reshape(-1,1)
        self.method = Method

        # Method: KM, EXP, REDUCED, DYNAMIC

        ValueArray = np.zeros([TimeArray.shape[0],1])
        ValueArray = np.hstack([self.TimeArray,ValueArray])

        if self.method == "KM":
            for i in range (0,int(TimeArray.shape[0])):
                ValueArray[i,1] = self.GetKaplanMeierValueSmooth(int(ValueArray[i,0]))

        if self.method == "EXP":
            for i in range (0,int(TimeArray.shape[0])):
                ValueArray[i,1] = self.GetExponentialValue(int(ValueArray[i,0]))

        if self.method == "REDUCED":
            for i in range (0,int(TimeArray.shape[0])):
                ValueArray[i,1] = self.GetReducedValue(self.GlobalReducedArray,int(ValueArray[i,0]))

        if self.method == "DYNAMIC":
            for i in range (0,int(TimeArray.shape[0])):
                ValueArray[i,1] = TrueDynamic.GetR(int(ValueArray[i,0]))

        return ValueArray

    def Kolmogorov(self,method1,method2):

        # KM, REDUCED, EXP, SELFMADE
        self.method1 = method1
        self.method2 = method2

        NumberOfKolmogorovCheckPoints = 30

        print("\nRun Kolmogorov()...")
        print("method1: " + str(self.method1))
        print("method2: " + str(self.method2))

        print("NumberOfKolmogorovCheckPoints: " +str(NumberOfKolmogorovCheckPoints))

        if NumberOfKolmogorovCheckPoints > 12:
            CriticialValue = 1.36 * math.sqrt((NumberOfKolmogorovCheckPoints+NumberOfKolmogorovCheckPoints) / (
      NumberOfKolmogorovCheckPoints * NumberOfKolmogorovCheckPoints))
        elif NumberOfKolmogorovCheckPoints == 10:
            CriticialValue = 0.7

        ListOfKolmogorovValues = []

        for c in range(0,100):
            print("Iteration Nr. " + str(c) + " / 100")

            CheckPointArray = np.sort(np.random.randint(0,self.MaximumEventTime,size=NumberOfKolmogorovCheckPoints)).reshape
      (-1,1)

            NumberOfRows = CheckPointArray.shape[0]
            ZeroArray = np.zeros([NumberOfRows,5])
            KolmogorovArray = np.hstack((CheckPointArray,ZeroArray))

            ## Fill KM and exponential values

            for i in range(0,NumberOfRows):

                TimeForPrediction = KolmogorovArray[i,0]

                if self.method1 == "KM":
                    KolmogorovArray[i,1] = self.GetKaplanMeierValueSmooth(TimeForPrediction)
                elif self.method1 == "EXP":
                    KolmogorovArray[i,1] = self.GetExponentialValue(TimeForPrediction)
                elif self.method1 == "REDUCED":
                    KolmogorovArray[i,1] = self.GetReducedValue(self.GlobalReducedArray,TimeForPrediction)
                elif self.method1 == "SELFMADE":
                    KolmogorovArray[i,1] = 1 - SelfmadeData.getF(int(TimeForPrediction))
                elif self.method1 == "POLY":
                    KolmogorovArray[i,1] = self.GetPolyValue(TimeForPrediction)
                elif self.method1 == "DYNAMIC":
                    KolmogorovArray[i,1] = Dynamic.GetR(TimeForPrediction)

                if self.method2 == "KM":
                    KolmogorovArray[i,2] = self.GetKaplanMeierValueSmooth(TimeForPrediction)
                elif self.method2 == "EXP":
                    KolmogorovArray[i,2] = self.GetExponentialValue(TimeForPrediction)
                elif self.method2 == "REDUCED":
                    KolmogorovArray[i,2] = self.GetReducedValue(self.GlobalReducedArray,TimeForPrediction)
                elif self.method2 == "SELFMADE":
                    KolmogorovArray[i,2] = 1 - SelfmadeData.getF(int(TimeForPrediction))
                elif self.method2 == "POLY":
                    KolmogorovArray[i,2] = self.GetPolyValue(TimeForPrediction)
                elif self.method2 == "DYNAMIC":
                    KolmogorovArray[i,2] = TrueDynamic.GetR(TimeForPrediction)

            for i in range(0,NumberOfRows):
                if i == 0:
                    KolmogorovArray[i,3] = KolmogorovArray[i,1]
                    KolmogorovArray[i,4] = KolmogorovArray[i,2]

                else:
                    KolmogorovArray[i,3] =  KolmogorovArray[i-1,3] + KolmogorovArray[i,1]
                    KolmogorovArray[i,4] =  KolmogorovArray[i-1,4] + KolmogorovArray[i,2]

            for i in range(0,NumberOfRows):
                KolmogorovArray[i,5] = abs(KolmogorovArray[i,3] - KolmogorovArray[i,4])

            KolmogorovValue = np.max(KolmogorovArray[:,5])
            KolmogorovIndex = np.argmax(KolmogorovArray[:,5])
            ListOfKolmogorovValues.append(KolmogorovValue)

        MeanKolmogorovValue = mean(ListOfKolmogorovValues)

        if MeanKolmogorovValue <= CriticialValue:
            print(str(MeanKolmogorovValue) + " < " + str(CriticialValue) + ": Samples based on same distribution!")

        if MeanKolmogorovValue > CriticialValue:
            print(str(MeanKolmogorovValue) + " > " + str(CriticialValue) + ": Samples NOT based on same distribution!")

        return KolmogorovArray, KolmogorovValue, KolmogorovIndex, MeanKolmogorovValue

    def FindNearestIndex(self,array,value):
        idx = (np.abs(array-value)).argmin()
        return idx

    def FindIndexLowerThanValue(self,array,value):
        DifferenceArray = (array-value)

        if value == 0:
            idx = 0
        else:
            idx = (np.argwhere(DifferenceArray<0)).argmax()

        return idx

    def FitPolyToKaplanMeier(self):
        KaplanMeierArray = self.KaplanMeier()
        X = KaplanMeierArray[:,0]
        Y = KaplanMeierArray[:,5]
        Z = np.polyfit(X, Y, self.PolyFitDegrees)
        p = np.poly1d(Z)
        return p

    def GetPolyValue(self,time):
        self.time = time
        Poly = self.FitPolyToKaplanMeier()
        return Poly(self.time)

    def KaplanMeier(self):

        base, ext = os.path.splitext(str(self.filename))

        FileNameForKaplanMeierArray = str(base) + "_KM_Array.txt"

        # if self.IsNumpyArray == False:
        try:
            KaplanMeierArray = np.loadtxt(FileNameForKaplanMeierArray)
            # print("Loaded KaplanMeierArray from existing text file " + str(FileNameForKaplanMeierArray))

        except:
            # Create an array with the Kaplan-Meier data inside

            # print("Running KaplanMeier()...")
            # Number of events in RawDataArray. Used for counting

            NumberOfEvents = int(self.GetNumberOfEvents())

            # Sort the RawDataArray according to the event times - failure times and censoring times
            # Save it in SortedArray

            SortedArray = self.RawDataArray[self.RawDataArray[:,0].argsort()]

            # Create RankArray from 1 to NumberOfEvents
            # Make it vertically afterwards

            RankArray = np.arange(1,NumberOfEvents+1)
            RankArray = np.vstack(RankArray)

            # Create a reversed version of the RankArray
            ReverseRankArray = RankArray[ : :-1]

            # Create array of ones
            OnesArray = np.ones([NumberOfEvents,2])

            # Stack all arrays horizontally

            KaplanMeierArray = np.hstack([SortedArray,RankArray,ReverseRankArray,OnesArray])


            # Calculate p in 4th column
            KaplanMeierArray[:,4] = (KaplanMeierArray[:,3]-1)/(KaplanMeierArray[:,3])


            # Ugly routine to set p to 1 for all censored sets
            for i in range (0, NumberOfEvents):
                if KaplanMeierArray[i,1] == self.censor_indicator:
                    KaplanMeierArray[i,4] = 1

            # Build Product for Kaplan-Meier Estimator in last column

            for i in range (0, NumberOfEvents):
                KaplanMeierArray[i,5] = np.product([KaplanMeierArray[0:i,4]])


```

```python
            if self.IsNumpyArray == False:
                np.savetxt(FileNameForKaplanMeierArray, KaplanMeierArray)

        return KaplanMeierArray

    def KaplanMeierArrayWithoutCensored(self):
        base, ext = os.path.splitext(str(self.filename))

        FileNameForKaplanMeierArrayWithoutCensored = str(base) + "_KM_Array_WO_Censored.txt"

        try:
            KaplanMeierArrayWithoutCensored = np.loadtxt(FileNameForKaplanMeierArrayWithoutCensored)
            # print("Loaded KaplanMeierArrayWithoutCensored from existing text file " + str(FileNameForKaplanMeierArrayWithoutCensored))

        except:
            # print("Running KaplanMeierArrayWithoutCensored()...")
            KaplanMeierArray = self.GlobalKaplanMeierArray

            KaplanMeierArrayWithoutCensored = KaplanMeierArray[KaplanMeierArray[:,1] == 1]
            if KaplanMeierArray[-1,1] == 0:
                KaplanMeierArrayWithoutCensored = np.vstack([KaplanMeierArrayWithoutCensored,KaplanMeierArray[-1:,]])

            np.savetxt(FileNameForKaplanMeierArrayWithoutCensored, KaplanMeierArrayWithoutCensored)

        return KaplanMeierArrayWithoutCensored

    def GetKaplanMeierValue(self,TimeForPrediction):
        self.TimeForPrediction = TimeForPrediction
        KaplanMeierArray = self.KaplanMeier()

        ResultArray = np.array([0])

        for x in np.nditer(TimeForPrediction):
            RowIndex = self.FindIndexLowerThanValue(KaplanMeierArray[:,0],x)
            ResultArray = np.hstack((ResultArray, KaplanMeierArray[RowIndex+1,-1]))

        ResultArray = np.delete(ResultArray, (0), axis=0)

        return ResultArray


    def GetKaplanMeierValueSmooth(self,TimeForPrediction):
        self.TimeForPrediction = TimeForPrediction
        KaplanMeierArray = self.GlobalKaplanMeierArrayWithoutCensored

        ResultArray = np.array([0])
        for x in np.nditer(TimeForPrediction):
            Result = np.interp(self.TimeForPrediction, KaplanMeierArray[:,0], KaplanMeierArray[:,-1])
            ResultArray = np.hstack((ResultArray, Result))

        return Result

    def GetReducedValue(self,ReducedArray,TimeForPrediction):

        self.ReducedArray = ReducedArray
        self.TimeForPrediction = TimeForPrediction

        X = self.ReducedArray[:,0]
        Y = self.ReducedArray[:,1]
        Estimation = np.interp(self.TimeForPrediction, X, Y)

        return Estimation

    def GetExponentialValue(self,TimeForPrediction):
        self.TimeForPrediction = TimeForPrediction
        return math.e ** (-(self.EstimatedLambda*self.TimeForPrediction))

    def CompareValues(self,TimeForPrediction,Method):

        self.TimeForPrediction = TimeForPrediction
        self.Method = Method

        KaplanMeierValue = self.GetKaplanMeierValue(self.TimeForPrediction)
        ExponentialValue = self.GetExponentialValue(self.TimeForPrediction)
        KaplanMeierValueSmooth = self.GetKaplanMeierValueSmooth(self.TimeForPrediction)

        if self.Method == "Exponential":
            ResultValue = ((ExponentialValue - KaplanMeierValue) / KaplanMeierValue) * 100

        if self.Method == "Reduced":
            ResultValue = ReducedValue - KaplanMeierValue

        if self.Method == "Smooth":
            ResultValue = ((ExponentialValue - KaplanMeierValueSmooth) / KaplanMeierValueSmooth) * 100

        return ResultValue

    def GetUpperConfidenceInterval(self,EstimatedFailureProbability,NumberOfFailures):
        self.EstimatedFailureProbability = EstimatedFailureProbability
        self.NumberOfFailures = NumberOfFailures

        UpperConfidenceInterval = self.EstimatedFailureProbability + 1.96 * ( self.EstimatedFailureProbability * ( 1 - self.EstimatedFailureProbability)/self.NumberOfFailures)**0.5

        return UpperConfidenceInterval

    def GetLowerConfidenceInterval(self, EstimatedFailureProbability, NumberOfFailures):
        self.EstimatedFailureProbability = EstimatedFailureProbability
        self.NumberOfFailures = NumberOfFailures

        LowerConfidenceInterval = self.EstimatedFailureProbability - 1.96 * ( self.EstimatedFailureProbability * ( 1 - self.EstimatedFailureProbability)/self.NumberOfFailures)**0.5

        return LowerConfidenceInterval

    def KaplanMeierWithConfidenceInterval(self):
        KaplanMeierArray = self.KaplanMeier()
        NumberOfEvents = self.GetNumberOfEvents()

        ConfidenceArray = np.zeros([NumberOfEvents,2])

        for i in range (0,NumberOfEvents):
            ConfidenceArray[i,0] = self.GetLowerConfidenceInterval(KaplanMeierArray[i,5],NumberOfEvents)
            ConfidenceArray[i,1] = self.GetUpperConfidenceInterval(KaplanMeierArray[i,5],NumberOfEvents)

        KaplanMeierWithConfidenceIntervalArray = np.hstack([KaplanMeierArray,ConfidenceArray])

        return KaplanMeierWithConfidenceIntervalArray

    def KaplanMeierArrayWithoutCensoredSmooth(self, verbose=0):

        KaplanMeierArrayWithoutCensored = self.KaplanMeierArrayWithoutCensored()

        KaplanMeierArrayWithoutCensored_X = KaplanMeierArrayWithoutCensored[:,0]
        KaplanMeierArrayWithoutCensored_Y = KaplanMeierArrayWithoutCensored[:,-1]

        NumberOfRows = KaplanMeierArrayWithoutCensored_X.shape[0]

        KaplanMeierArrayWithoutCensoredSmooth_Y = np.copy(KaplanMeierArrayWithoutCensored_Y)

        for i in range (0,NumberOfRows-1):

            First_Value = KaplanMeierArrayWithoutCensored_Y[i]
            Second_Value = KaplanMeierArrayWithoutCensored_Y[i+1]
            Difference = First_Value - Second_Value
            New_Value = Second_Value + (Difference / 2.0)
            KaplanMeierArrayWithoutCensoredSmooth_Y[i] = New_Value

            if verbose == 1:
                print("Working on Line number " + str(i))
                print("\tTime: " + str(KaplanMeierArrayWithoutCensored_X[i]))
                print("\tR-Value of Line number " + str(i) + ": " + str(First_Value))
                print("\tR-Value of Line number " + str(i+1) + ": " + str(Second_Value))
                print("\tDifference: " + str(Difference))
                print("\tNew Value: " + str(New_Value))
                print("\n")

        KaplanMeierArrayWithoutCensoredSmooth = np.column_stack([KaplanMeierArrayWithoutCensored_X, KaplanMeierArrayWithoutCensoredSmooth_Y])
        FirstLine = np.array([0,1])
        KaplanMeierArrayWithoutCensoredSmooth = np.vstack([FirstLine,KaplanMeierArrayWithoutCensoredSmooth])
        return KaplanMeierArrayWithoutCensoredSmooth

    def Plot(self,PlotKaplanMeier,PlotKaplanMeierSmooth,PlotKaplanMeierConfidenceInterval,PlotEstimatedExponential,PlotPolyFit,PlotReduced,PlotDifferences,ShowPrediction,PlotKolmogorov,PlotSelfmade,PlotDynamic,ShowPlot,SavePlot,PlotFile):

        self.PlotKaplanMeier = bool(PlotKaplanMeier)
        self.PlotKaplanMeierConfidenceInterval = bool(PlotKaplanMeierConfidenceInterval)
        self.PlotKaplanMeierSmooth = bool(PlotKaplanMeierSmooth)
        self.PlotEstimatedExponential = bool(PlotEstimatedExponential)
        self.PlotPolyFit = bool(PlotPolyFit)
        self.PlotReduced = bool(PlotReduced)
        self.PlotDifferences = bool(PlotDifferences)
        self.ShowPrediction = bool(ShowPrediction)
        self.PlotKolmogorov = bool(PlotKolmogorov)
        self.PlotSelfmade = bool(PlotSelfmade)
        self.PlotDynamic = bool(PlotDynamic)
        self.ShowPlot = bool(ShowPlot)
        self.SavePlot = bool(SavePlot)
        self.PlotFile = str(PlotFile)

        # Plots various graphs, according to parameters.
        # KaplanMeier - Plot KM if set to 1
        # Show - Show the plot
        # Save - Save the plot
        # PlotFile - Filename for the saved plot

        # Set generic plot options
        # Set LaTeX font (Computer Modern)
        rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})#, 'size'   : 16})
        rc('text', usetex=True)

        # Set-up figure and axes
        fig, ax1 = plt.subplots()#, sharex=True)

        # Define Labels for x and y axis
        plt.xlabel("Time t [h]")
        plt.ylabel("Reliability R [-]")

        # Axis range
        ymin = 0
        xmax_value = int(self.MaximumEventTime)
        plt.xlim(xmin=0, xmax=xmax_value)
        plt.ylim(ymin=ymin, ymax=1.0)

        # Set title. All "_" have to be replaced by "\_" for LaTeX escaping
        TitleName = str(self.filename).replace("_", "\_")
        Subtitle = "Number of Events: " + str(self.GetNumberOfEvents()) + " (" + str(self.GetNumberOfFailures()) + " Failures, " + str(self.GetNumberOfCensored()) + " Censored)"
        ax1.set_title("Data from " + str(TitleName) + "\n" + Subtitle)

        if PlotKaplanMeier == True:
            # Run the KaplanMeier() function
            KaplanMeierArray = self.GlobalKaplanMeierArray

            # Define x and y for the plot
            x_KM = KaplanMeierArray[:,[0]]
            y_KM = KaplanMeierArray[:,[-1]]
            ax1.step(x_KM,y_KM, color="red", linewidth=1, alpha=1, label="Kaplan-Meier Estimator")
            ax1.legend(loc=3)


        if PlotKaplanMeierSmooth == True:

            KaplanMeierArrayWithoutCensoredSmooth = self.KaplanMeierArrayWithoutCensoredSmooth()

            x_KM_Smooth = KaplanMeierArrayWithoutCensoredSmooth[:,0]
            y_KM_Smooth = KaplanMeierArrayWithoutCensoredSmooth[:,-1]
            ax1.plot(x_KM_Smooth,y_KM_Smooth, color = '#8c00ff', linewidth=1, alpha=1, label="Kaplan-Meier Estimator (Smooth)")

        if PlotKaplanMeierConfidenceInterval == True:
            KaplanMeierWithConfidenceIntervalArray = self.KaplanMeierWithConfidenceInterval()
            X = KaplanMeierWithConfidenceIntervalArray[:,0]
            Y_Lower = KaplanMeierWithConfidenceIntervalArray[:,6]
            Y_Upper = KaplanMeierWithConfidenceIntervalArray[:,7]

            ax1.step(X,Y_Upper, color="red", alpha=0.8, linestyle="-.", label="Upper KM Confidence Interval")
            ax1.step(X,Y_Lower, color="red", alpha=0.8, linestyle="-.", label="Lower KM Confidence Interval")

        if PlotEstimatedExponential == True:

            # Set X-Axis to range
            upper_limit = xmax_value
            x_Lambda = np.arange(0,upper_limit)

            EstimatedLambda = self.EstimateLambda()
            EstimatedLambdaSci = '%.2E' % Decimal(EstimatedLambda)
            ax1.plot(x_Lambda, math.e ** (-(EstimatedLambda*x_Lambda)), color="blue", linestyle="--", linewidth=2, alpha=1, label="Exponential Failure Probability ($\lambda$ = " + str(EstimatedLambdaSci) + ")")
            ax1.legend(loc=3)

        if PlotPolyFit == True:

            X = self.KaplanMeier()[:,[0]]
            Poly = self.FitPolyToKaplanMeier()
            print(Poly)
            ax1.plot(X, Poly(X), color="orange", linewidth=2, alpha=1, label="Polynomial fit (" + str(self.PolyFitDegrees) + " degrees)")

        if PlotReduced == True:

            ReducedArray = self.ReducedDataIntelligent()
            X = ReducedArray[:,0]
            Y = ReducedArray[:,1]

            ax1.scatter(X, Y, color="#5492f7", linewidth=1.5, alpha=1)
            ax1.plot(X, Y, color="#5492f7", linewidth=2, alpha=0.8, label="Reduced Data Set (" + str(self.AmountOfAutomaticallyFoundSegments) + " Sections, " + str(self.ReductionThreshold) + " Threshold)")

        if PlotDifferences == True:

            ax2 = ax1.twinx()

            Timerange = np.arange(0,self.MaximumEventTime,100)

            if PlotReduced == True:
                DifferenceArrayReduced = self.CompareValues(Timerange,"Reduced")
                ax2.plot(Timerange, DifferenceArrayReduced, color="cyan", alpha=0.7, linewidth=0.8, label="Value difference (Reduced to KM)")

            if PlotKaplanMeierSmooth == True:
                DifferenceArraySmooth = self.CompareValues(Timerange,"Smooth")
                ax2.plot(Timerange, DifferenceArraySmooth, color="green", alpha=0.7, linewidth=0.8, label="Value difference (Exponential to smooth Kaplan-Meier)")
                ax2.set_ylabel("Difference [\%]")
                ax2.tick_params('y', color="green")
                ax2.axhline(y=0, color="green", alpha=0.7, linewidth=0.5, linestyle="-")

                ax2.legend(loc=0)

        if ShowPrediction == True:

            # Expand the X axis. Use self.ExpansionFactor as a factor
            plt.xlim(xmin=0, xmax=int(self.MaximumEventTime)*self.ExpansionFactor)

            # Plot exponential for that area
            x_Lambda = np.arange(int(self.MaximumEventTime),int(self.MaximumEventTime)*self.ExpansionFactor)

            EstimatedLambda = self.EstimateLambda()
            EstimatedLambdaSci = '%.2E' % Decimal(EstimatedLambda)

            KaplanMeierArray = self.KaplanMeier()
            Offset = KaplanMeierArray[-1,-1] - math.e ** (-(EstimatedLambda*self.MaximumEventTime))
            OffsetSci = round(Offset, 2)

            SmoothKaplanMeierValueAtMaximumEventTime = self.GetKaplanMeierValueSmooth(self.MaximumEventTime)
            PredictLambda = -1 * (math.log(SmoothKaplanMeierValueAtMaximumEventTime) / self.MaximumEventTime)
            PredictLambdaSci = '%.2E' % Decimal(PredictLambda)

            if PlotEstimatedExponential == True:
                if Offset >= 0:
                    ax1.plot(x_Lambda, math.e ** (-(EstimatedLambda*x_Lambda)) , color="blue", linestyle="--", linewidth=2,
        alpha=1)
                else:
                    ax1.plot(x_Lambda, math.e ** (-(EstimatedLambda*x_Lambda)) , color="blue", linestyle="--", linewidth=2,
        alpha=0.5)
            else:
                if Offset >= 0:
                    ax1.plot(x_Lambda, math.e ** (-(EstimatedLambda*x_Lambda)) , color="blue", linestyle="--", linewidth=2,
        alpha=1, label="Original Exponential Failure Probability ($\lambda$ = " + str(EstimatedLambdaSci) + ")")
                else:
                    ax1.plot(x_Lambda, math.e ** (-(PredictLambda*x_Lambda)) , color="green", linestyle="--", linewidth=2,
        alpha=1, label="Adapted Exponential Failure Probability ($\lambda_{Prediction}$ = " + str(PredictLambdaSci)+ ")")
                    ax1.plot(x_Lambda, math.e ** (-(EstimatedLambda*x_Lambda)) , color="blue", linestyle="--", linewidth=2,
        alpha=1, label="Original Exponential Failure Probability ($\lambda$ = " + str(EstimatedLambdaSci) + ")")

            if PlotPolyFit == True:
                X = np.arange(int(self.MaximumEventTime),int(self.MaximumEventTime)*self.ExpansionFactor)
                Poly = self.FitPolyToKaplanMeier()


                ax1.plot(X, Poly(X), color="green", linestyle="--", linewidth=2, alpha=1)

            # Add hatching for predicted time frame

            patterns = ['-', '+', 'x', 'o', 'O', '.', '*', '|', '/']
            ax1.add_patch(
                patches.Rectangle(
                    (self.MaximumEventTime, -0.2),   # (x,y)
                    self.MaximumEventTime*self.ExpansionFactor,          # width
                    1.3,          # height
                    # hatch='/',
                    hatch = patterns[8],
                    edgecolor="red",
                    # alpha=0.5,
                    fill=False
                    )
                )


        if PlotKolmogorov == True:
            KolmogorovArray, KolmogorovValue, KolmogorovIndex, MeanKolmogorovValue, SameDistribution = self.Kolmogorov(1)
            KolmogorovTimes = KolmogorovArray[:,0]
            KolmogorovKMValues = KolmogorovArray[:,1]
            KolmogorovExpValues = KolmogorovArray[:,2]
            KolmogorovKM_Cum_Values = KolmogorovArray[:,3]
            KolmogorovExp_Cum_Values = KolmogorovArray[:,4]

            plt.ylim(ymin=ymin, ymax=5)

            ax1.plot(KolmogorovTimes, KolmogorovKM_Cum_Values, color='#8c00ff', alpha=1, linewidth=1, label="Cumulative Values
        for smoothened Kaplan-Meier")
            ax1.plot(KolmogorovTimes, KolmogorovExp_Cum_Values, color='blue', alpha=1, linewidth=1, label="Cumulative Values
        for exponential distribution")
            ax1.set_ylabel("Cumulative Values [-]")


            for i in range(0,len(KolmogorovTimes)):
                x = [KolmogorovTimes[i], KolmogorovTimes[i]]
                y = [KolmogorovKM_Cum_Values[i], KolmogorovExp_Cum_Values[i]]

                ax1.plot(x, y, marker = '', color = "green")

            # Maximum Kolmogorov line
            x_max = [KolmogorovTimes[KolmogorovIndex], KolmogorovTimes[KolmogorovIndex]]
            y_max = [KolmogorovKM_Cum_Values[KolmogorovIndex], KolmogorovExp_Cum_Values[KolmogorovIndex]]
            ax1.plot(x_max, y_max, marker = '', color = "red", label="Kolmogorov-Smirnov test value (" + str(round(
        KolmogorovValue,3)) + ")")


        if PlotSelfmade == True:

            upper_limit = xmax_value
            x_Selfmade = np.arange(0,10100,100)
            y_Selfmade = np.arange(0,10100,100)

            y_list = []
            for i in range(0,len(y_Selfmade)):
                Time = int(y_Selfmade[i])
                R_Value = 1 - SelfmadeData.getF(Time)

                y_list.append(R_Value)


            ax1.plot(x_Selfmade, y_list, color="green", linestyle="-.", linewidth=2, alpha=1, label="True Selfmade distribution reliability")

        if PlotDynamic == True:

            x_Dynamic = np.arange(0,26280,100)
            ax1.plot(x_Dynamic, TrueDynamic.GetR(x_Dynamic), color="green", linestyle="-.", linewidth=2, alpha=1, label="True dynamic reliability model")


        if ShowPlot == True:

            plt.legend(loc="upper right")
            plt.show()


        if SavePlot == True:
            plt.legend(loc="upper right")
            plt.savefig(PlotFile)

def KolmogorovArrayCheck(Array1,Array2):


    Array1 = Array1.reshape(-1,1)
    Array2 = Array2.reshape(-1,1)
    if Array1.shape[0] != Array2.shape[0]:
        print("The two input arrays do not have the same size! ABORT!")


    NumberOfKolmogorovCheckPoints = Array1.shape[0]

    if NumberOfKolmogorovCheckPoints <= 12:
        Critical_Values = [0,0,0,0,1,1,0.83,0.857,0.75,0.667,0.7,0.636,0.667]
        CriticialValue = Critical_Values[NumberOfKolmogorovCheckPoints]

    else:
        CriticialValue = 1.36 * math.sqrt((NumberOfKolmogorovCheckPoints+NumberOfKolmogorovCheckPoints) / (NumberOfKolmogorovCheckPoints * NumberOfKolmogorovCheckPoints))

    ListOfKolmogorovValues = []

    for c in range(0,100):
        NumberOfRows = NumberOfKolmogorovCheckPoints
        ZeroArray = np.zeros([NumberOfRows,3])
        ZeroArray2 = np.zeros([NumberOfRows,1])
        KolmogorovArray = np.hstack((ZeroArray2,Array1,Array2,ZeroArray))

        for i in range(0,NumberOfRows):
            if i == 0:
                KolmogorovArray[i,3] = KolmogorovArray[i,1]
                KolmogorovArray[i,4] = KolmogorovArray[i,2]

            else:
                KolmogorovArray[i,3] =  KolmogorovArray[i-1,3] + KolmogorovArray[i,1]
                KolmogorovArray[i,4] =  KolmogorovArray[i-1,4] + KolmogorovArray[i,2]

        for i in range(0,NumberOfRows):
            KolmogorovArray[i,5] = abs(KolmogorovArray[i,3] - KolmogorovArray[i,4])


        KolmogorovValue = np.max(KolmogorovArray[:,5])
        KolmogorovIndex = np.argmax(KolmogorovArray[:,5])
        ListOfKolmogorovValues.append(KolmogorovValue)

    MeanKolmogorovValue = mean(ListOfKolmogorovValues)

    if MeanKolmogorovValue <= CriticialValue:
        print(str(MeanKolmogorovValue) + "\t<\t " + str(CriticialValue) + ":\tSamples based on same distribution!")
        SameDistribution = 1

    if MeanKolmogorovValue > CriticialValue:
        print(str(MeanKolmogorovValue) + "\t>\t" + str(CriticialValue) + ":\tSamples NOT based on same distribution!")
        SameDistribution = 0

    return KolmogorovArray, KolmogorovValue, KolmogorovIndex, MeanKolmogorovValue, SameDistribution

def ReturnRandomSample(InputArray,Number):

    NumberOfEvents = int(InputArray.shape[0])
    ListOfIndices = random.sample(range(0, NumberOfEvents), Number)
    RandomSampleArray = np.sort(InputArray.take(ListOfIndices, axis=0), axis=0)

    return RandomSampleArray




def CensorArray(Array):

    Length = Array.shape[0]
    Ones = np.ones([Length,1])
    CensoredArray = np.hstack((Array,Ones))

    return CensoredArray

```

```python
def HowManyPoints(Montecarlo_Counter):

    DataSetToBeChecked = Selfmade

    # This is the Raw Data Array to use as source. Used for Exprosoft and Backblaze
    FullDataArray = DataSetToBeChecked.RawDataArray
    filename_prefix = "how_many_points_temp_data_selfmade/Selfmade"
    NumberOfKolmogorovCheckPoints = 30

    # This is the number of entries in that Raw Data Array
    NumberOfMaximumEntries = FullDataArray.shape[0]

    DictionaryOfSuccess = {}
    ListOfSuccess = []
    ListOfSamplesize = []

    ResultArray = np.array([0,999])

    for c in range(5,15,2):
        print("Monte Carlo Number " + str(Montecarlo_Counter) + ", Iteration with " + str (c) + " samples...")

        NumberOfSamples = c


        # What indices of the Raw Data Array are to be considered? For Wellmaster and Backblaze
        # ListOfIndicesToBeConsidered = random.sample(range(0, NumberOfMaximumEntries), NumberOfSamples)
        # TestingArray = np.sort(FullDataArray.take(ListOfIndicesToBeConsidered, axis=0), axis=0)

        TestingArray = np.array([])
        for i in range(0,c):
            TestingArray = np.append(TestingArray,SelfmadeData.GetSingleFailureTime())
            TestingArray = TestingArray.reshape(-1,1)

        TestingArray = CensorArray(TestingArray)

        # print("Our TestingArray:")

        FileNameForTempArray = filename_prefix + "_Raw_Data_Iteration_" + str(c).zfill(4) + "_Samples.txt"
        np.savetxt(FileNameForTempArray, TestingArray,delimiter=";")

        # print("\tArray " + str(FileNameForTempArray) + " saved...")

        TempDataset = Dataset( FileNameForTempArray,
                                timebase="hours",
                               failure_indicator = 1,
                               censor_indicator = 0,
                               NumberOfReducedSections = 5,
                               PolyFitDegrees = 4,
                               ExpansionFactor = 3,
                               ReductionThreshold = 0.0005,
                               IsNumpyArray = False,
                               ArrayName = "blank")

        # Create a random time array with NumberOfKolmogorovCheckPoints number of times between 0 and the maximum time in the
       current iteration
        CheckPointArray = np.sort(np.random.randint(0,TempDataset.MaximumEventTime,size=NumberOfKolmogorovCheckPoints)).reshape
       (-1,1)

        # Create array with KM-smooth values of this sample array
        KM_Sample_Array = np.zeros([NumberOfKolmogorovCheckPoints,1])
        for i in range(0,NumberOfKolmogorovCheckPoints):
            KM_Sample_Array[i] = TempDataset.GetKaplanMeierValueSmooth(CheckPointArray[i])

        # Create array with Reliability values of true array
        KM_Full_Array = np.zeros([NumberOfKolmogorovCheckPoints,1])
        for i in range(0,NumberOfKolmogorovCheckPoints):
            # KM_Full_Array[i] = DataSetToBeChecked.GetReducedValue(DataSetToBeChecked.GlobalReducedArray,CheckPointArray[i])
            KM_Full_Array[i] = 1 - SelfmadeData.getF(int(CheckPointArray[i]))

        KolmogorovArray, KolmogorovValue, KolmogorovIndex, MeanKolmogorovValue, SameDistribution = KolmogorovArrayCheck(
       KM_Sample_Array,KM_Full_Array)

        # Setting first column in ResultArray
        ResultArrayLine = np.array([c,KolmogorovValue])
        ResultArray = np.vstack([ResultArray, ResultArrayLine])
        print("")

    np.savetxt("how_many_points_temp_data_selfmade_results/Selfmade_HowManyPoints_Result_" + str(Montecarlo_Counter).zfill(4) +
        "_Samples.txt", ResultArray)

    return ResultArray

def MonteCarlo(runs):

    for i in range(0,runs):
        print("Iteration " + str(i))
        Resultarray = HowManyPoints(i)

        if i == 0:
            # print("First run!")
            MonteCarloArray = Resultarray

        if i > 0:
            # print("Not first run")
            MonteCarloArray = np.hstack([MonteCarloArray,Resultarray[:,-1].reshape(-1,1)])


        # print("Array so far: ")
        # print(MonteCarloArray)

        np.savetxt("how_many_points_temp_data_selfmade_results/Selfmade_MonteCarlo_Result_" + str(i).zfill(4) + "th_Run.txt",
       MonteCarloArray)

        files_to_be_deleted  = glob.glob('how_many_points_temp_data_selfmade\*')
        for f in files_to_be_deleted:
            os.remove(f)

    print("Finished")
    np.savetxt("how_many_points_temp_data_selfmade_results/Selfmade_MonteCarlo_FinalArray.txt", MonteCarloArray)



A = Dataset(  "failure_dates/A-Data.txt",
                    timebase="hours",
                    failure_indicator = 1,
                    censor_indicator = 0,
                    NumberOfReducedSections = 5,
                    PolyFitDegrees = 4,
                    ExpansionFactor = 3,
                    ReductionThreshold = 0.0004,
                    IsNumpyArray = False,
                    ArrayName = "blank")



A.Plot(     PlotKaplanMeier=0,
            PlotKaplanMeierSmooth=0,
            PlotKaplanMeierConfidenceInterval=0,
            PlotEstimatedExponential=1,
            PlotPolyFit=1,
            PlotReduced=1,
            PlotDifferences=0,
            ShowPrediction=0,
            PlotKolmogorov=0,
            PlotSelfmade=0,
            PlotDynamic=1,
            ShowPlot=1,
            SavePlot=0,
            PlotFile="C_KM_Exponential_Comparison.pdf")
```

Listing A.2: statistic_tools.py Python source code

```
import numpy as np
import math
import sys
import os

class Dynamic:
    def __init__(self,ParameterArray):
        self.ParameterArray = ParameterArray

    def GetF(self,TimeForPrediction):

        self.TimeForPrediction = TimeForPrediction
        Result = np.interp(self.TimeForPrediction, self.ParameterArray[:,0], self.ParameterArray[:,-1])
        return Result

    def GetR(self,TimeForPrediction):

        self.TimeForPrediction = TimeForPrediction
        Result = 1 - self.GetF(self.TimeForPrediction)
        return Result

    def GetTime(self,FailureProbability):
        self.FailureProbability = FailureProbability
        ResultTime = np.interp(self.FailureProbability, self.ParameterArray[:,-1], self.ParameterArray[:,0])
        return int(ResultTime)

def CreateFailureTimes(Filename,Modelname,Number):

    try:
       output = open(Filename, "w")

    except:
      sys.stderr.write('Unable to open file "%s"\n' % Filename)
      sys.stderr.flush()
      exit

    for i in range(0,Number):
        output.write(str(Modelname.GetTime(np.random.uniform(0,1.0))) + ";1\n")

    output.close()
```

Listing A.3: dynamic_reliability.py Python source code

```python
# Import numerical python module with short handle np
import numpy as np

# Import math module
import math
import os

# Generate a Treatment class

class Selfmade:
    def __init__(self,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,z1,z2,z3,z4,z5,z6,z7):
        self.t1 = t1
        self.t2 = t2
        self.t3 = t3
        self.t4 = t4
        self.t5 = t5
        self.t6 = t6
        self.t7 = t7
        self.t8 = t8
        self.t9 = t9
        self.t10 = t10
        self.t11= t11
        self.z1 = z1
        self.z2 = z2
        self.z3 = z3
        self.z4 = z4
        self.z5 = z5
        self.z6 = z6
        self.z7 = z7

    def getz(self,t):
        self.t = t
        if self.t < 0:
            t = 0
            print("Negative time entered - t set to zero")

        elif self.t > 0 and self.t <= self.t1:
            z = ((self.z2 - self.z1)/self.t1)*self.t+self.z1
            return z

        elif self.t > self.t1 and self.t <= self.t2:
            z = self.z2
            return z

        elif self.t > self.t2 and self.t <= self.t3:
            z = ((self.z3 - self.z2)/(self.t3 - self.t2))*(self.t-self.t2)+self.z2
            return z

        elif self.t > self.t3 and self.t <= self.t4:
            z = self.z3
            return z

        elif self.t > self.t4 and self.t <= self.t5:
            z = ((self.z4 - self.z3)/(self.t5 - self.t4))*(self.t-self.t4)+self.z3
            return z

        elif self.t > self.t5 and self.t <= self.t6:
            z = self.z4
            return z

        elif self.t > self.t6 and self.t <= self.t7:
            z = ((self.z5 - self.z4)/(self.t7 - self.t6))*(self.t-self.t6)+self.z4
            return z

        elif self.t > self.t7 and self.t <= self.t8:
            z = self.z5
            return z

        elif self.t > self.t8 and self.t <= self.t9:
            z = ((self.z6 - self.z5)/(self.t9 - self.t8))*(self.t-self.t8)+self.z5
            return z

        elif self.t > self.t9 and self.t <= self.t10:
            z = self.z6
            return z

        elif self.t > self.t10 and self.t <= self.t11:
            z = ((self.z7 - self.z6)/(self.t11 - self.t10))*(self.t-self.t10)+self.z6
            return z

    def getInt(self,t):
        self.t = t
        integral = 0
        if self.t < 0:
            t = 0
            print("Negative time entered - t set to zero")

        elif self.t >= 0 and self.t <= self.t1:
            integral_temp = np.trapz([self.z1,self.z2], x=[0,self.t])
            integral = integral_temp
            return integral

        elif self.t > self.t1 and self.t <= self.t2:
            integral_temp = np.trapz([self.z2,self.z2], x=[self.t1,self.t])
            integral = integral_temp + self.getInt(self.t1)
            return integral

        elif self.t > self.t2 and self.t <= self.t3:
            integral_temp = np.trapz([self.z2,self.z3], x=[self.t2,self.t])
            integral = integral_temp +  self.getInt(self.t2)
            return integral

        elif self.t > self.t3 and self.t <= self.t4:
            integral_temp = np.trapz([self.z3,self.z3], x=[self.t3,self.t])
            integral = integral_temp + self.getInt(self.t3)
            return integral

        elif self.t > self.t4 and self.t <= self.t5:
            integral_temp = np.trapz([self.z3,self.z4], x=[self.t4,self.t])
            integral = integral_temp + self.getInt(self.t4)
            return integral

        elif self.t > self.t5 and self.t <= self.t6:
            integral_temp = np.trapz([self.z4,self.z4], x=[self.t5,self.t])
            integral = integral_temp + self.getInt(self.t5)
            return integral

        elif self.t > self.t6 and self.t <= self.t7:
            integral_temp = np.trapz([self.z4,self.z5], x=[self.t6,self.t])
            integral = integral_temp + self.getInt(self.t6)
            return integral

        elif self.t > self.t7 and self.t <= self.t8:
            integral_temp = np.trapz([self.z5,self.z5], x=[self.t7,self.t])
            integral = integral_temp + self.getInt(self.t7)
            return integral

        elif self.t > self.t8 and self.t <= self.t9:
            integral_temp = np.trapz([self.z5,self.z6], x=[self.t8,self.t])
            integral = integral_temp + self.getInt(self.t8)
            return integral

        elif self.t > self.t9 and self.t <= self.t10:
            integral_temp = np.trapz([self.z6,self.z6], x=[self.t9,self.t])
            integral = integral_temp + self.getInt(self.t9)
            return integral

        elif self.t > self.t10 and self.t <= self.t11:
            integral_temp = np.trapz([self.z6,self.z7], x=[self.t10,self.t])
            integral = integral_temp + self.getInt(self.t10)
            return integral


    def getF(self,t):
        self.t = t
        return 1 - math.e ** (-1 * self.getInt(t))

    def getpdf(self,t):
        self.t = t
        z = self.getz(t)
        pdf = math.e ** (-1 * self.getInt(t)) * z
        return pdf

    def array(self,t):
        self.t = t
        data = []

        for i in range (1,int(self.t),1):
            temp = np.array([i, self.getz(i), self.getF(i), self.getpdf(i)])
            data.append(temp)

        return np.array(data)

    def save_parameters(self):
        np.savetxt("S_10000_Parameters.txt", try2.array(t), delimiter=";")



    def GetSingleFailureTime(self):
        dataset = np.loadtxt("S_10000_Parameters.txt", delimiter=";")
        times = dataset[:, [0]]

        # Third column are the F values
        failure = np.around(dataset[:, [2]], decimals=3)
        max_F = np.amax(failure)

        value = np.random.uniform(low=0.0001, high=max_F)
        random_index = np.searchsorted(np.ravel(failure), value, side="left")

        if random_index > 0 and (random_index == len(failure) or math.fabs(value - failure[random_index-1]) < math.fabs(value -
        failure[random_index])):
            T = times[random_index-1]
        else:
            T = times[random_index]

        return T






```

```python
    def failure_times(self, count, repeat, random):
        self.count = count
        self.repeat = repeat
        self.random = random

        mu = 1
        sigma = 0.2

        t = 10000
        # Load the generated parameters from the S function
        dataset = np.loadtxt("S_10000_Parameters.txt", delimiter=";")

        # First column are the times
        times = dataset[:, [0]]

        # Third column are the F values
        failure = np.around(dataset[:, [2]], decimals=3)
        max_F = np.amax(failure)

        for r in range (1, self.repeat+1):
            failure_times  = np.array([])

            for i in range (0,self.count):
                value = np.random.uniform(low=0.0001, high=max_F)
                random_index = np.searchsorted(np.ravel(failure), value, side="left")

                if random == True:
                    if random_index > 0 and (random_index == len(failure) or math.fabs(value - failure[random_index-1]) < math.fabs(value - failure[random_index])):
                        T = int(times[random_index-1] * np.absolute(np.random.normal(mu, sigma)))
                        failure_times = np.append(failure_times, T)
                    else:
                        T = int(times[random_index] * np.absolute(np.random.normal(mu, sigma)))
                        failure_times = np.append(failure_times, T)
                else:
                    if random_index > 0 and (random_index == len(failure) or math.fabs(value - failure[random_index-1]) < math.fabs(value - failure[random_index])):
                        T = times[random_index-1]
                        failure_times = np.append(failure_times, T)
                    else:
                        T = times[random_index]
                        failure_times = np.append(failure_times, T)

            if random == True:
                np.savetxt("failure_times/SN_" + str(self.count) + "_" + str(r) + ".txt", np.sort(failure_times), delimiter=";")
            else:
                np.savetxt("failure_times/S_" + str(self.count) + "_" + str(r) + ".txt", np.sort(failure_times), delimiter=";")
```

Listing A.4: generator.py Python source code