1 Quickstart for reliability
2 Introduction to the field of reliability engineering
3 Recommended resources
4 Equations of supported distributions
5 Creating and plotting distributions
6 Fitting a specific distribution to data
7 Fitting all available distributions to data
8 Mixture models
9 Competing risks models
10 Probability plots
11 Quantile-Quantile plots
12 Probability-Probability plots
13 Kaplan-Meier
14 Nelson-Aalen
15 Rank Adjustment
16 Stress-Strength interference for any distributions
17 Stress-Strength interference for normal distributions
18 Reliability growth
19 Optimal replacement time
20 ROCOF
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Mean cumulative function</td>
<td>121</td>
</tr>
<tr>
<td>22</td>
<td>SN diagram</td>
<td>127</td>
</tr>
<tr>
<td>23</td>
<td>Stress-strain and strain-life</td>
<td>131</td>
</tr>
<tr>
<td>24</td>
<td>Fracture mechanics</td>
<td>141</td>
</tr>
<tr>
<td>25</td>
<td>Creep</td>
<td>147</td>
</tr>
<tr>
<td>26</td>
<td>Palmgren-Miner linear damage model</td>
<td>151</td>
</tr>
<tr>
<td>27</td>
<td>Acceleration factor</td>
<td>153</td>
</tr>
<tr>
<td>28</td>
<td>Solving simultaneous equations with sympy</td>
<td>155</td>
</tr>
<tr>
<td>29</td>
<td>ALT probability plots</td>
<td>159</td>
</tr>
<tr>
<td>30</td>
<td>Fitting a model to ALT data</td>
<td>167</td>
</tr>
<tr>
<td>31</td>
<td>Similar Distributions</td>
<td>173</td>
</tr>
<tr>
<td>32</td>
<td>Convert dataframe to grouped lists</td>
<td>175</td>
</tr>
<tr>
<td>33</td>
<td>Crosshairs</td>
<td>177</td>
</tr>
<tr>
<td>34</td>
<td>Distribution explorer</td>
<td>179</td>
</tr>
<tr>
<td>35</td>
<td>Histogram</td>
<td>181</td>
</tr>
<tr>
<td>36</td>
<td>Make right censored data</td>
<td>185</td>
</tr>
<tr>
<td>37</td>
<td>Datasets</td>
<td>187</td>
</tr>
<tr>
<td>38</td>
<td>One sample proportion</td>
<td>191</td>
</tr>
<tr>
<td>39</td>
<td>Two proportion test</td>
<td>193</td>
</tr>
<tr>
<td>40</td>
<td>Sample size required for no failures</td>
<td>195</td>
</tr>
<tr>
<td>41</td>
<td>Sequential sampling chart</td>
<td>197</td>
</tr>
<tr>
<td>42</td>
<td>Reliability test planner</td>
<td>201</td>
</tr>
<tr>
<td>43</td>
<td>Reliability test duration</td>
<td>205</td>
</tr>
<tr>
<td>44</td>
<td>Chi-squared test</td>
<td>209</td>
</tr>
<tr>
<td>45</td>
<td>Kolmogorov-Smirnov test</td>
<td>213</td>
</tr>
<tr>
<td>46</td>
<td>Changelog</td>
<td>217</td>
</tr>
<tr>
<td>47</td>
<td>Development roadmap</td>
<td>227</td>
</tr>
<tr>
<td>48</td>
<td>Citing reliability in your work</td>
<td>229</td>
</tr>
<tr>
<td>49</td>
<td>How to request or contribute a new feature</td>
<td>231</td>
</tr>
<tr>
<td>50</td>
<td>How to get help</td>
<td>233</td>
</tr>
</tbody>
</table>
51 How to donate to the project 235
52 About the author 237
53 Credits to those who helped 239
reliability is a Python library for reliability engineering and survival analysis. It significantly extends the functionality of scipy.stats and also includes many specialist tools that are otherwise only available in proprietary software.

If you frequently use the Python Reliability Library, please consider filling out a quick survey to help guide the development of the library and this documentation.
1.1 Installation and upgrading

If you are new to using Python, you will first need to install a Python 3 interpreter and also install an IDE so that you can interact with the code. There are many good IDEs available including Pycharm, Spyder and Jupyter.

Once you have Python installed, to install reliability for the first time, open your command prompt and type:

```
pip install reliability
```

To upgrade a previous installation of reliability to the most recent version, open your command prompt and type:

```
pip install --upgrade reliability
```

If you would like to be notified by email of when a new release of reliability is uploaded to PyPI, there is a free service to do exactly that called NewReleases.io.

1.2 A quick example

In this example, we will create a Weibull Distribution, and from that distribution we will draw 20 random samples. Using those samples we will fit a 2-parameter Weibull Distribution. The fitting process generates the probability plot. We can then access the distribution object to plot the survival function.

```python
from reliability.Distributions import Weibull_Distribution
from reliability.Fitters import Fit_Weibull_2P
from reliability.Probability_plotting import plot_points
import matplotlib.pyplot as plt

dist = Weibull_Distribution(alpha=30, beta=2)  # creates the distribution object
data = dist.random_samples(20, seed=42)  # draws 20 samples from the distribution...
# Seeded for repeatability
plt.subplot(121)
```

(continues on next page)
A key feature of reliability is that probability distributions are created as objects, and these objects have many properties (such as the mean) that are set once the parameters of the distribution are defined. Using the dot operator
allows us to access these properties as well as a large number of methods (such as drawing random samples as seen in the example above).

Each distribution may be visualised in five different plots. These are the Probability Density Function (PDF), Cumulative Distribution Function (CDF), Survival Function (SF) [also known as the reliability function], Hazard Function (HF), and the Cumulative Hazard Function (CHF). Accessing the plot of any of these is as easy as any of the other methods. Eg. `dist.SF()` in the above example is what plots the survival function using the distribution object that was returned from the fitter.
CHAPTER 2

Introduction to the field of reliability engineering

Reliability engineering is a field of study that deals with the estimation, prevention, and management of failures by combining statistics, risk analysis, and physics. By understanding how failures may occur or have occurred, we are able to better predict the lifespan of a product, allowing us to manage its lifecycle and risks associated with failure. All engineering systems, components, and structures will eventually fail, and knowing how and when that failure will occur is of great interest to the owners and operators of those systems. Due to the similarities between the lifecycle of engineering systems and the lifecycle of humans, the field of study known as survival analysis has many concepts that are used in reliability engineering.

Everyone is acutely aware of the importance of reliability, particularly when something doesn’t work at a time we expect it to. Whether it be your car not starting, your computer’s hard drive failing, or the chance of being delayed on the runway because your aircraft’s airconditioning unit just stopped working, we know that system failure is something we all want to avoid, and when it can’t be avoided, we at least want to know when it is likely to occur. Reliability engineering is most frequently used for systems which are of critical safety importance (such as in the nuclear industry), or in systems which are numerous (such as vehicles or electronics) where the cost of fleetwide reliability problems is extremely expensive.

Much of reliability engineering involves the analysis of data (such as time to failure), to uncover the patterns in how failures occur and to use those patterns to forecast how the failures will occur throughout the lifetime of a population of items, or the lifetime of one or more repairable items. It is the data analysis part of reliability engineering that this python library is designed to help with.

Further reading is available on Wikipedia and in many other reliability resources.
CHAPTER 3

Recommended resources

The following collection of resources are things I have found useful during my reliability engineering studies and also while writing the Python reliability library. There are many other resources available (especially textbooks and academic papers), so I encourage you to do your own research. If you find something you think is worth adding here, please send me an email (alpha.reliability@gmail.com).

Textbooks

- Probability Distributions Used in Reliability Engineering (2011), by A. O’Conner, M. Modarres, and A. Mosleh.
- Recurrent Events Data Analysis for Product Repairs, Disease Recurrences, and Other Applications (2003), by W. Nelson
- Reliasoft has compiled a much more comprehensive list of textbooks.
- The reliability analytics toolkit (linked below in free online tools and calculators) has also compiled a much more comprehensive list of textbooks.

Free software

- Lifelines - a Python library for survival analysis. Very powerful collection of tools, only a few of which overlap with the Python reliability library.
- Parameter Solver v3.0 - a biostatistics tool for quickly making some simple calculations with probability distributions.
- Orange - a standalone data mining and data visualization program that runs using Python. Beautifully interactive data analysis workflows with a large toolbox. Not much reliability related content but good for data preprocessing.
- R (Programming Language) - R is one of the most popular programming languages for data science, and it has several libraries that are targeted towards reliability engineering and survival analysis. These include WeibullR, abrem, and survival.
• **CumFreq** - a program for cumulative frequency analysis with probability distribution fitting for a wide range of distributions. Limited functionality beyond fitting distributions.

• **OpenTURNS** - a Python library for the treatment of uncertainties, risks and statistics. This library contains many powerful statistical functions, some of which are applicable to reliability engineering (mainly the fitting of distributions). The syntax of the library requires many steps as shown in the tutorials.

### Paid software

The listing of a software package here does not imply my endorsement, and is only intended to give readers an understanding of the broad range of reliability engineering software packages that are available. It is difficult to find a comprehensive list of software resources since most developers of proprietary software rarely acknowledge the existence of any software other than their own. I have not used most of the paid software listed here due to the high cost, so most of my comments in this section are based purely on the content from their websites.

• **Minitab** - a great collection of statistical tools. A few reliability focussed tools included.

• **Reliasoft** - the industry leader for reliability engineering software.

• **SAS JMP** - lots of statistical tools for data modelling and visualization. A few purpose built reliability tools. Its utility for reliability engineering will depend on your application. SAS has also released the **SAS University Edition** which is a free software package that runs in VirtualBox and offers a reduced set of tools compared to the paid package.

• **PTC Windchill** - a powerful tool for risk and reliability. Similar to Reliasoft but it forms one part of the larger PTC suite of tools.

• **Isograph Reliability Workbench** - A collection of tools designed specifically for reliability engineering.

• **Item Software** - A collection of tools for reliability engineering including FMECA, fault trees, reliability prediction, and many others.

• **SuperSMITH** - This software is designed specifically for reliability engineering and has many useful tools. The user interface looks like it is from the early 1990s but the methods used are no less relevant today. This software was developed alongside the New Weibull Handbook, an excellent resource for interpreting the results of reliability engineering software.

• **RAM Commander** - A software tool for Reliability and Maintainability Analysis and Prediction, Spares Optimisation, FMEA/FMECA, Testability, Fault Tree Analysis, Event Tree Analysis and Safety Assessment.

• **RelCalc** - RelCalc for Windows automates the reliability prediction procedure of Telcordia SR-332, or MIL-HDBK-217, providing an alternative to tedious, time consuming, and error prone manual methods.

• **Relyence** - Relyence offers a range of products, similar to Reliasoft, each with a focus on a different area including Life Data Analysis, Accelerated Life Testing, Reliability Block Diagrams, FMEA, and several more.

• **@RISK** - A comprehensive Excel addon that allows for distribution fitting, reliability modelling, MC simulation and much more.

• **Quanterion Automated Reliability Toolkit (QuART)** - A collection of reliability tools including reliability prediction, FMECA, derating, stress-strength interference, and many other. Quanterion produces several software products so their tools are not all available in one place.

• **TopEvent FTA** - Fault Tree Analysis software. Tailored specifically for fault tree analysis so it lacks other RAM tools but it is good at its intended function. A demo version is available with size and data export limitations.

• **Maintenance Aware Design (MADe)** - FMECA and RCM software that is extremely useful at the product design stage to inform the design and service plan which then improves the inherent reliability and maintainability. There is an academic license which allows non-profit users to run the software for free.

### Free online tools and calculators
• **Reliability Analytics Toolkit** - a collection of tools which run using the Google App Engine. Includes a tool for fitting a Weibull_2P distribution.

• **Weibull App** - An online tool for fitting a Weibull_2P distribution. Download the example template to see what format the app is expecting your data to be in before you can upload your own data. The backend is powered by the abrem R package. This tool has limited applications beyond fitting a Weibull_2P distribution.

• **Distributome** - Provides PDF and CDF of a large number of probability distributions that can be easily changed using sliders for their parameters. It also includes a quantile / CDF calculator. Similar to the Distribution calculator below.

• **Distribution Calculator** - Provides PDF and CDF of a large number of probability distributions that can be easily changed using sliders for their parameters. It also includes a quantile / CDF calculator. Similar to Distributome above.

• **Kijima G-renewal process** - an online calculator for simulating the G-renewal process.

• **Prediction of future recurrent events** - an online calculator for predicting future recurrent events with different underlying probability functions.

• **Maintenance optimization** - an online calculator for optimal replacement policy (time) under Kijima imperfect repair model.

• **e-Fatigue** - This website provides stress concentration factors (Kt) for various notched geometries. You will need this if using the functions for fracture mechanics in the Physics of Failure section.

• **Fault Tree Analyser** - A simple online tool where you can build a fault tree, give each branch a failure rate and run a variety of reports including reliability prediction at time, minimal cut sets, and several others.

• **Wolfram Alpha** - an amazing computational knowledge engine. Great for checking your calculations.

• **Derivative calculator** - calculates derivatives. Slightly more user friendly input method than Wolfram alpha and doesn’t time out as easily for big calculations.

• **Integral calculator** - calculates integrals. Slightly more user friendly input method than Wolfram alpha and doesn’t time out as easily for big calculations.

• **GeoGebra** - An interactive calculator that is extremely useful for plotting equations. Also includes many mathematical operations (such as integrals and derivatives) that allow you to keep your equations in symbolic form. You can download your current calculator to save it. The only downside is that there are not many probability distribution functions inbuilt so you will need to enter the equations manually.

• **NewReleases.io** - This website allows you to setup email notifications for when a new release of reliability (or any other package) is uploaded to PyPI. While not exactly a tool for reliability engineering, it is very useful to let you know when it’s time to upgrade your version of reliability.

### Online information resources

• **Reliawiki** - an excellent reference written by Reliasoft that is intended largely as a guide to reliability engineering when using Reliasoft’s software but is equally as good to understand concepts without using their software.

• **Reliasoft’s Accelerated Life Testing Data Analysis Reference**

• **Reliasoft’s collection of Military Directives, Handbooks and Standards Related to Reliability**

• **Univariate distributions relationships** - a great interactive diagram for understanding more about probability distributions and how they are related. Some strange parametrisations are used in the documentation.

• **Cross Validated** - an forum for asking statistics and mathematics questions. Check for existing answers before posting your own question.

• **Stack Overflow** - a forum for programmers where you can post questions and answers related to programming. Check for existing answers before posting your own question.
• Wikipedia - it’s always worth checking if there’s an article on there about the topic you’re trying to understand.

### Getting free access to academic papers

• arXiv - a database run by Cornell university that provides open access to over 1.5 million academic papers that have been submitted. If you can’t find it here then check on Sci-Hub.

• Sci-Hub - paste in a DOI to get a copy of the academic paper. Accessing academic knowledge should be free and this site makes it possible.

---

RELIABILITY

A Python library for reliability engineering
Equations of supported distributions

The following expressions provide the equations for the Probability Density Function (PDF), Cumulative Distribution Function (CDF), Survival Function (SF) (this is the same as the reliability function \( R(t) \)), Hazard Function (HF), and Cumulative Hazard Function (CHF) of all supported distributions. Readers should note that there are many ways to write the equations for probability distributions and careful attention should be afforded to the parametrization to ensure you understand each parameter. For more equations of these distributions, see the textbook “Probability Distributions Used in Reliability Engineering” listed in recommended resources.

### 4.1 Weibull Distribution

\( \alpha = \) scale parameter \((\alpha > 0)\)  
\( \beta = \) shape parameter \((\beta > 0)\)  
Limits \((t \geq 0)\)

PDF: \[ f(t) = \frac{\beta t^{\beta-1}}{\alpha^\beta} e^{-\left(\frac{t}{\alpha}\right)^\beta} \]  
\[ = \frac{\beta}{\alpha} \left(\frac{t}{\alpha}\right)^{\beta-1} e^{-\left(\frac{t}{\alpha}\right)^\beta} \]

CDF: \[ F(t) = 1 - e^{-\left(\frac{t}{\alpha}\right)^\beta} \]

SF: \[ R(t) = e^{-\left(\frac{t}{\alpha}\right)^\beta} \]

HF: \[ h(t) = \frac{\beta}{\alpha} \left(\frac{t}{\alpha}\right)^{\beta-1} \]

CHF: \[ H(t) = \left(\frac{t}{\alpha}\right)^\beta \]

### 4.2 Exponential Distribution

\( \lambda = \) scale parameter \((\lambda > 0)\)  
Limits \((t \geq 0)\)
4.3 Normal Distribution

μ = location parameter \((-∞ < μ < ∞)\)

σ = scale parameter \((σ > 0)\)

Limits \((-∞ < t < ∞)\)

PDF:
\[
f(t) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{t - μ}{\sigma} \right)^2 \right]
\]

\[
= \frac{1}{\sigma} \phi \left( \frac{t - μ}{\sigma} \right)
\]

where \(\phi\) is the standard normal PDF with \(μ = 0\) and \(σ = 1\)

CDF:
\[
F(t) = \frac{1}{\sigma \sqrt{2\pi}} \int_{-∞}^{t} \exp \left[ -\frac{1}{2} \left( \frac{θ - μ}{σ} \right)^2 \right] dθ
\]

\[
= \frac{1}{2} + \frac{1}{2} \text{erf} \left( \frac{t - μ}{\sigma \sqrt{2}} \right)
\]

\[
= \Phi \left( \frac{t - μ}{\sigma} \right)
\]

where \(\Phi\) is the standard normal CDF with \(μ = 0\) and \(σ = 1\)

SF:
\[
R(t) = 1 - \Phi \left( \frac{t - μ}{σ} \right)
\]

\[
= \Phi \left( \frac{μ - t}{σ} \right)
\]

HF:
\[
h(t) = \frac{\phi \left( \frac{t - μ}{σ} \right)}{σ \Phi \left( \frac{μ - t}{σ} \right)}
\]

CHF:
\[
H(t) = -\ln \left[ \Phi \left( \frac{μ - t}{σ} \right) \right]
\]

4.4 Lognormal Distribution

μ = scale parameter \((-∞ < μ < ∞)\)

σ = shape parameter \((σ > 0)\)

Limits \((t ≥ 0)\)

PDF:
\[
f(t) = \frac{1}{σt \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{\ln(t) - μ}{σ} \right)^2 \right]
\]

\[
= \frac{1}{σ} \phi \left( \frac{\ln(t) - μ}{σ} \right)
\]

where \(\phi\) is the standard normal PDF with \(μ = 0\) and \(σ = 1\)

CDF:
\[
F(t) = \frac{1}{σ \sqrt{2π}} \int_0^t \frac{1}{θ} \exp \left[ -\frac{1}{2} \left( \frac{\ln(θ) - μ}{σ} \right)^2 \right] dθ
\]
\[
= \frac{1}{2} + \frac{1}{2} \text{erf} \left( \frac{\ln(t) - \mu}{\sigma \sqrt{2}} \right)
= \Phi \left( \frac{\ln(t) - \mu}{\sigma} \right)
\]
where \( \Phi \) is the standard normal CDF with \( \mu = 0 \) and \( \sigma = 1 \)

**SF:**
\[
R(t) = 1 - \Phi \left( \frac{\ln(t) - \mu}{\sigma} \right)
\]

**HF:**
\[
h(t) = \frac{\phi \left( \frac{\ln(t) - \mu}{\sigma} \right)}{\sigma \left( 1 - \Phi \left( \frac{\ln(t) - \mu}{\sigma} \right) \right)}
\]

**CHF:**
\[
H(t) = -\ln \left[ 1 - \Phi \left( \frac{\ln(t) - \mu}{\sigma} \right) \right]
\]

### 4.5 Gamma Distribution

\( \alpha \) = scale parameter \(( \alpha > 0 )\)

\( \beta \) = shape parameter \(( \beta > 0 )\)

**Limits** \(( t \geq 0 )\)

**PDF:**
\[
f(t) = \frac{t^{\alpha - 1}}{\Gamma(\alpha) \beta} e^{-\frac{t}{\beta}}
\]

where \( \Gamma(x) \) is the complete gamma function. \( \Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \)

**CDF:**
\[
F(t) = \frac{1}{\Gamma(\beta)} \gamma(\beta, \frac{t}{\alpha})
\]

where \( \gamma(\alpha, \beta) \) is the lower incomplete gamma function. \( \gamma(\alpha, \beta) = \frac{1}{\Gamma(\alpha)} \int_0^\beta t^{\alpha-1} e^{-t} dt \)

**SF:**
\[
R(t) = \frac{1}{\Gamma(\beta)} \Gamma(\beta, \frac{t}{\alpha})
\]

**HF:**
\[
h(t) = \frac{t^{\beta-1} \exp \left( -\frac{t}{\alpha} \right)}{\alpha^\beta \Gamma(\beta, \frac{t}{\alpha})}
\]

**CHF:**
\[
H(t) = -\ln \left[ \frac{1}{\Gamma(\beta)} \Gamma(\beta, \frac{t}{\alpha}) \right]
\]

Note that some parametrizations of the Gamma distribution use \( \frac{1}{\alpha} \) in place of \( \alpha \). There is also an alternative parametrization which uses shape and rate instead of shape and scale. See Wikipedia for an example of this.

### 4.6 Beta Distribution

\( \alpha \) = shape parameter \(( \alpha > 0 )\)

\( \beta \) = shape parameter \(( \beta > 0 )\)

**Limits** \(( 0 \leq t \leq 1 )\)

**PDF:**
\[
f(t) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha) \Gamma(\beta)} t^{\alpha-1} (1-t)^{\beta-1} = \frac{1}{B(\alpha, \beta)} t^{\alpha-1} (1-t)^{\beta-1}
\]

where \( \Gamma(x) \) is the complete gamma function. \( \Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \)

where \( B(x, y) \) is the complete beta function. \( B(x, y) = \int_0^1 t^{x-1} (1-t)^{y-1} dt \)

4.5. Gamma Distribution 15
CDF: \[ F(t) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^t \theta^{\alpha-1}(1 - \theta)^{\beta-1} d\theta \]
\[ = \frac{B_t(\alpha, \beta)}{B(\alpha, \beta)} \]
\[ = I_t(\alpha, \beta) \]
where \( B_t(\alpha, \beta) \) is the incomplete beta function. \( B_t(\alpha, \beta) = \int_0^t \theta^{\alpha-1}(1 - \theta)^{\beta-1} d\theta \)
where \( I_t(\alpha, \beta) \) is the regularized incomplete beta function which is defined in terms of the incomplete beta function and the complete beta function. \( I_t(\alpha, \beta) = \frac{B_t(\alpha, \beta)}{B(\alpha, \beta)} \)
SF: \[ R(t) = 1 - I_t(\alpha, \beta) \]
HF: \[ h(t) = \frac{t^{\alpha-1}(1-t)}{B(\alpha, \beta) - B_t(\alpha, \beta)} \]
CHF: \[ H(t) = -\ln \left[ 1 - I_t(\alpha, \beta) \right] \]

Note that there is a parameterization of the Beta distribution that changes the lower and upper limits beyond 0 and 1. For this parametrization, see the reference listed in the opening paragraph of this page.

### 4.7 Loglogistic Distribution

\( \alpha = \) scale parameter \((\alpha > 0)\)
\( \beta = \) shape parameter \((\beta > 0)\)

Limits \((t \geq 0)\)

PDF: \[ f(t) = \left( \frac{\beta}{\alpha} \right) \left( \frac{t}{1 + \left( \frac{t}{\alpha} \right)^{\beta}} \right)^{\beta-1} \]

CDF: \[ F(t) = \frac{1}{1 + \left( \frac{t}{\alpha} \right)^{\beta}} \]
\[ = \frac{\left( \frac{t}{\alpha} \right)^{\beta}}{1 + \left( \frac{t}{\alpha} \right)^{\beta}} \]
\[ = \frac{t^{\beta}}{\alpha^{\beta} + t^{\beta}} \]
SF: \[ R(t) = \frac{1}{1 + \left( \frac{t}{\alpha} \right)^{\beta}} \]
HF: \[ h(t) = \left( \frac{\beta}{\alpha} \right) \left( \frac{t}{1 + \left( \frac{t}{\alpha} \right)^{\beta}} \right)^{\beta-1} \]
CHF: \[ H(t) = \ln \left( 1 + \left( \frac{t}{\alpha} \right)^{\beta} \right) \]

There is another parameterization of the loglogistic distribution using \( \mu \) and \( \sigma \) which is designed to look more like the parametrization of the logistic distribution and is related to the above parametrization by \( \mu = ln(\alpha) \) and \( \sigma = \frac{1}{\beta} \). This parametrization can be found here.

### 4.8 Gumbel Distribution

\( \mu = \) location parameter \((-\infty < \mu < \infty)\)
\( \sigma = \) scale parameter \((\sigma > 0)\)

Limits \((-\infty < t < \infty)\)

PDF: \[ f(t) = \frac{1}{\sigma} e^{-e^{-\frac{t-\mu}{\sigma}}} \]
where \( z = \frac{t-\mu}{\sigma} \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF: ( F(t) )</td>
<td>( 1 - e^{-e^z} )</td>
</tr>
<tr>
<td>SF: ( R(t) )</td>
<td>( e^{-e^z} )</td>
</tr>
<tr>
<td>HF: ( h(t) )</td>
<td>( e^z )</td>
</tr>
<tr>
<td>CHF: ( H(t) )</td>
<td>( e^{e^z} )</td>
</tr>
</tbody>
</table>

The parametrization of the Gumbel Distribution shown above is also known as the Smallest Extreme Value (SEV) distribution. There are several types of extreme value distributions, and the article on Wikipedia is for the Largest Extreme Value (LEV) distribution. There is only a slight difference in the parametrisation between SEV and LEV distributions, but this change effectively flips the PDF about \( \mu \) to give the LEV positive skewness (a longer tail to the right), while the SEV has negative skewness (a longer tail to the left).

### 4.9 Location shifting the distributions

Within reliability the parametrisation of the Exponential, Weibull, Gamma, Lognormal, and Loglogistic distributions allows for location shifting using the gamma parameter. This will simply shift the distribution’s lower limit to the right from 0 to \( \gamma \). In the location shifted form of the distributions, the equations listed above are almost identical, except everywhere you see \( t \) replace it with \( t - \gamma \). The reason for using the location shifted form of the distribution is because some phenomena that can be modelled well by a certain probability distribution do not begin to occur immediately, so it becomes necessary to shift the lower limit of the distribution so that the data can be accurately modelled by the distribution.

If implementing this yourself, ensure you set all y-values to 0 for \( t \leq \gamma \) as the raw form of the location shifted distributions above will not automatically zeroise these values for you and may result in negative values. This zeroizing is done automatically within reliability.

### 4.10 Relationships between the five functions

The PDF, CDF, SF, HF, CHF of a probability distribution are inter-related and any of these functions can be obtained by applying the correct transformation to any of the others. The following list of transformations are some of the most useful:

- PDF = \( \frac{d}{dt} \) CDF
- CDF = \( \int_{-\infty}^{t} \) PDF
- SF = 1 - CDF
- HF = \( \frac{PDF}{SF} \)
- CHF = \( -\ln \) (SF)
Chapter 4. Equations of supported distributions
There are 8 standard probability distributions available in `reliability.Distributions`. These are:

- Weibull Distribution \((\alpha, \beta, \gamma)\)
- Exponential Distribution \((\lambda, \gamma)\)
- Gamma Distribution \((\alpha, \beta, \gamma)\)
- Normal Distribution \((\mu, \sigma)\)
- Lognormal Distribution \((\mu, \sigma, \gamma)\)
- Loglogistic Distribution \((\alpha, \beta, \gamma)\)
- Gumbel Distribution \((\mu, \sigma)\)
- Beta Distribution \((\alpha, \beta)\)

In all of the distributions which use \(\gamma\), the \(\gamma\) parameter is used to location shift the distribution to the right. If used, the \(\gamma\) parameter must be greater than or equal to 0. The Beta distribution is only defined in the range 0 to 1. All distributions except the Normal and Gumbel distributions are defined in the positive domain only \((x>0)\).

Understanding how to create and plot distributions is easiest with an example. In this first example, we will create a Weibull Distribution with parameters \(\alpha = 50\) and \(\beta = 2\). We will then plot the PDF of the distribution.

```python
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt

dist = Weibull_Distribution(alpha=50, beta=2)  # this created the distribution object
dist.PDF()  # this creates the plot of the PDF
plt.show()
```
Just as easily as we plotted the PDF in the above example, we can plot any of the 5 characteristic functions (PDF, CDF, SF, HF, CHF). If you would like to view all of these functions together, you can use the plot() method. In this second example, we will create a Lognormal Distribution with parameters mu=2 and sigma=0.5. From this distribution, we will use the plot() method to visualise the five functions and also provide a summary of the descriptive statistics.

```python
from reliability.Distributions import Lognormal_Distribution

dist = Lognormal_Distribution(mu=2, sigma=0.5)
dist.plot()
```
Probability distributions within *reliability* are Python objects, which allows us to specify just the type of distribution and its parameters. Once the distribution object is created, we can access a large number of methods (such as PDF or plot as we did above). Some of the methods require additional input and some have optional inputs.

The following methods are available for all distributions:

- **name** - a string of the distribution name. Eg. ‘Weibull’
- **name2** - a string of the distribution name including the number of parameters. Eg. ‘Weibull_2P’
- **param_title_long** - Useful in plot titles, legends and in printing strings. Varies by distribution. eg. ‘Weibull Distribution (a=5,b=2)’
- **param_title** - Useful in plot titles, legends and in printing strings. Varies by distribution. eg. ‘a=5,b=2’
- **parameters** - returns an array of parameters. These are in the order specified in the bullet points above, so for Lognormal it would return [mu,sigma, gamma].
- **alpha, beta, gamma, Lambda, mu, sigma** - these vary by distribution but will return the value of their respective parameter. Eg. dist.mu would return 2 in the above example.
- **mean**
- **variance**
- **standard_deviation**
- **skewness**

![Lognormal Distribution](image)

Probability density, cumulative distribution, and cumulative hazard functions with parameters \( \mu=2, \sigma=0.5 \).

- **Mean** = 8.3729
- **Median** = 7.3891
- **Mode** = 5.7546
- **5\text{th} quantile** = 3.2465
- **95\text{th} quantile** = 16.8176
- **Standard deviation** = 4.4623
- **Variance** = 19.9117
- **Skewness** = 1.7502
- **Excess kurtosis** = 5.8984
• kurtosis
• excess_kurtosis
• median
• mode
• b5 - the time at which 5% of units have failed. Same as dist.quantile(0.05)
• b95 - the time at which 95% of units have failed. Same as dist.quantile(0.95)
• plot() - plots all functions (PDF, CDF, SF, HF, CHF). Also accepts xvals, xmin, xmax.
• PDF() - plots the probability density function. Also accepts xvals, xmin, xmax, show_plot, and plot keywords (eg. color).
• CDF() - plots the cumulative distribution function. Also accepts xvals, xmin, xmax, show_plot, and plot keywords (eg. color).
• SF() - plots the survival function (also known as reliability function). Also accepts xvals, xmin, xmax, show_plot, and plot keywords (eg. color).
• HF() - plots the hazard function. Also accepts xvals, xmin, xmax, show_plot, and plot keywords (eg. color).
• CHF() - plots the cumulative hazard function. Also accepts xvals, xmin, xmax, show_plot, and plot keywords (eg. color).
• quantile() - Calculates the quantile (time until a fraction has failed) for a given fraction failing. Also known as ‘b’ life where b5 is the time at which 5% have failed. You must specify the y-value at which to calculate the quantile. Eg. dist.quantile(0.05) will give the b5 life.
• inverse_SF() - Calculates the inverse of the survival function. Useful when producing QQ plots. You must specify the y-value at which to calculate the inverse SF. Eg. dist.inverse_SF(0.8) will give the time at which 80% have not failed.
• mean_residual_life() - Average residual lifetime of an item given that the item has survived up to a given time. Effectively the mean of the remaining amount (right side) of a distribution at a given time. You must specify the x-value at which to calculate MRL. Eg. dist.mean_residual_life(10)
• stats() - prints all the descriptive statistics. Same as the statistics shown using .plot() but printed to console. No arguments are accepted.
• random_samples() - draws random samples from the distribution to which it is applied. Same as rvs in scipy.stats. You must specify the number of samples. Eg. data = dist.random_samples(100) will set data as a list of 100 random samples from the distribution. If you want repeatability, specify the seed E.g data = dist.random_samples(100, seed=1).

For all of the individual plotting functions (PDF, CDF, SF, HF, CHF), all standard matplotlib plotting keywords (such as label, color, linestyle, etc.) are accepted and used. If not specified they are preset. In specifying the plotting positions for the x-axis, there are optional keywords to be used. The first of these is ‘xvals’ which accepts a list of x-values to use for the horizontal axis. Alternatively, the user may specify ‘xmin’ and/or ‘xmax’ if there is a desired minimum or maximum value. If left unspecified these will be set automatically. xvals overrides xmin and xmax.

Note that .plot() does not require plt.show() to be used as it will automatically show, however the other 5 plotting functions will not be displayed until plt.show() is used. This is to allow the user to overlay multiple plots on the figure or change titles, labels, and legends as required. The plot can be turned off by specifying show_plot=False.

Each of the 5 functions (PDF, CDF, SF, HF, CHF) will always return the y-values for a given set of x-values (xvals). In this example, we want to know the value of the Survival Function at x=20.
As a final example, we will create a bathtub curve by creating and layering several distributions. The bathtub curve is only for the Hazard function as it shows how a variety of failure modes throughout the life of a population can shape the hazard into a bathtub shape. The three distinct regions are infant mortality, random failures, and wear out. In this example, the returned y-values are added together to produce the ‘combined’ array which is then plotted using matplotlib against the xvals. By specifying xvals in each HF we can ensure that the xvals used will all align. Leaving xvals unspecified would not be appropriate in this example as the default xvals will depend on the shape of the plot.
Further detail about all of the functions is available using the help function within Python. Simply type:

```python
from reliability.Distributions import Lognormal_Distribution
print(help(Lognormal_Distribution))
```
The module `reliability.Fitters` provides many probability distribution fitting functions. These functions can be thought of in three categories; non-location shifted distributions [e.g. Weibull ($\alpha, \beta$)], location shifted distributions [e.g. Weibull ($\alpha, \beta, \gamma$)], and special distributions (e.g. Weibull Mixture model). All of the distributions can be fitted to both complete and incomplete (right censored) data. All distributions in the Fitters module are named with their number of parameters (e.g. Fit_Weibull_2P uses $\alpha, \beta$, whereas Fit_Weibull_3P uses $\alpha, \beta, \gamma$). This is intended to remove ambiguity about what distribution you are fitting.

Distributions are fitted simply by using the desired function and specifying the data as failures or right_censored data. You must have at least as many failures as there are distribution parameters or the fit would be under-constrained. It is generally advisable to have at least 4 data points as the accuracy of the fit is proportional to the amount of data. Once fitted, the results are assigned to an object and the fitted parameters can be accessed by name, as shown in the examples below. The goodness of fit criterions are also available as AICc (Akaike Information Criterion corrected), BIC (Bayesian Information Criterion), and AD (Anderson-Darling), though these are more useful when comparing the fit of multiple distributions such as in the `Fit_Everything` function. As a matter of convenience, each of the modules in Fitters also generates a distribution object that has the parameters of the fitted distribution.

The available functions are:

- `Fit_Weibull_2P`
- `Fit_Weibull_3P`
- `Fit_Exponential_1P`
- `Fit_Exponential_2P`
- `Fit_Gamma_2P`
- `Fit_Gamma_3P`
- `Fit_Lognormal_2P`
- `Fit_Lognormal_3P`
- `Fit_Loglogistic_2P`
- `Fit_Loglogistic_3P`
- `Fit_Normal_2P`
• Fit_Gumbel_2P
• Fit_Beta_2P
• Fit_Weibull_Mixture (see the section on this)
• Fit_Weibull_CR (see the section on this)

Note: The Beta distribution is only for data in the range 0 to 1. Specifying data outside of this range will cause an error.

Note: If you have a very large amount of data (>100000 samples) then it is likely that your computer will take significant time to compute the results. This is a limitation of interpreted languages like Python compared to compiled languages like C++ which many commercial reliability software packages are written in. If you have very large volumes of data, you may want to consider using commercial software for faster computation time. The function Fit_Weibull_2P_grouped is designed to accept a dataframe which has multiple occurrences of some values (e.g., multiple values all right censored to the same value when the test was ended). Depending on the size of the data set and the amount of grouping in your data, Fit_Weibull_2P_grouped may be much faster than Fit_Weibull_2P and achieve the same level of accuracy. This difference is not noticeable if you have less than 10000 samples. For more information, see the example below on using Fit_Weibull_2P_grouped.

Note: Heavily censored data (>99.9% censoring) may result in a failure of the optimizer to find a solution. If you have heavily censored data, you may have a limited failure population problem. It is recommended that you do not try fitting one of these standard distributions to such a dataset as your results (while they may have achieved a successful fit) will be a poor description of your overall population statistic and you risk drawing the wrong conclusions when the wrong model is fitted. The limited failure population model is planned for a future release of reliability, though development on this model is yet to commence. In the meantime, see JMP Pro’s model for Defective Subpopulations.

Note: The confidence intervals are available for all of the standard distributions except for Fit_Gamma_2P, Fit_Gamma_3P and Fit_Beta_2P. This library is being actively developed and it is likely that Gamma and Beta distributions will have confidence intervals added in the next release.

If you do not know which distribution you want to fit, then please see the section on using the Fit_Everything function which will find the best distribution to describe your data. It is highly recommended that you always try to fit everything and accept the best fit rather than choosing a particular distribution for subjective reasons.

Each of the fitters listed above (except Fit_Weibull_Mixture, Fit_Weibull_CR, and Fit_Weibull_2P_grouped) has the following inputs and outputs:

Inputs:
• failures - an array or list of failure data
• right_censored - an array or list of right censored data. Optional input
• show_probability_plot - True/False. Defaults to True. Produces a probability plot of the failure data and fitted distribution.
• print_results - True/False. Defaults to True. Prints a dataframe of the point estimate, standard error, Lower CI and Upper CI for each parameter.
• CI - confidence interval for estimating confidence limits on parameters. Must be between 0 and 1. Default is 0.95 for 95% CI.
• percentiles - True/False (for default list) or specified list or array of percentiles (0 to 100) to print with the results. These are the percentiles on time based on the specified CI. Not available for Fit_Gamma_2P, Fit_Gamma_3P and Fit_Beta_2P due to confidence intervals not yet implemented.

• force_beta (in Fit_Weibull_2P) or force_sigma (in Fit_Normal_2P and Fit_Lognormal_2P). This allows the user to force the shape parameter to be a set value. Useful for ALT probability plotting. Optional input and only available for Fit_Weibull_2P, Fit_Normal_2P and Fit_Lognormal_2P as these are the distributions typically used in ALT probability plotting.

• keyword arguments are also accepted for the probability plot (eg. color, linestyle, marker)

Outputs (the following example outputs are for the Fit_Weibull_2P distribution but for other distributions the parameter names may be different from alpha and beta):

• alpha - the fitted Weibull_2P alpha parameter
• beta - the fitted Weibull_2P beta parameter
• loglik - Log-Likelihood (as used in Minitab and Reliasoft)
• loglik2 - Log-Likelihood*-2 (as used in JMP Pro)
• AICc - Akaika Information Criterion (goodness of fit statistic)
• BIC - Bayesian Information Criterion (goodness of fit statistic)
• AD - the Anderson-Darling goodness of fit statistic (uses the same formula as Minitab)
• distribution - a Distribution object with the parameters of the fitted distribution
• alpha_SE - the standard error (sqrt(variance)) of the parameter
• beta_SE - the standard error (sqrt(variance)) of the parameter. This will be ‘‘ if the shape parameter has been forced to a set value.
• Cov_alpha_beta - the covariance between the parameters. This will be ‘’ for Fit_Expon_1P or if the shape parameter has been forced to a set value.
• alpha_upper - the upper CI estimate of the parameter
• alpha_lower - the lower CI estimate of the parameter
• beta_upper - the upper CI estimate of the parameter. This will be ‘’ if the shape parameter has been forced to a set value.
• beta_lower - the lower CI estimate of the parameter. This will be ‘’ if the shape parameter has been forced to a set value.
• results - a dataframe of the results (point estimate, standard error, Lower CI and Upper CI for each parameter)
• percentiles - a dataframe of the percentiles.
• success - True/False. Indicated whether the solution was found by autograd. If success is False a warning will be printed indicating that the initial guess was used as autograd failed. This fit will not be accurate if there is censored data. Failure of autograd to find the solution should be rare and if it occurs, it is likely that the distribution is an extremely bad fit for the data. Try scaling your data, removing extreme values, or using another distribution.

To learn how we can fit a distribution, we will start by using a simple example with 30 failure times. These times were generated from a Weibull distribution with $\alpha=50$, $\beta=3$. Note that the output also provides the confidence intervals and standard error of the parameter estimates. The probability plot is generated be default (you will need to specify plt.show() to show it). See the section on probability plotting to learn how to interpret this plot.
from reliability.Fitters import Fit_Weibull_2P
import matplotlib.pyplot as plt

data = [58,75,36,63,65,22,17,28,64,23,40,73,45,52,36,52,60,13,55,82,55,34,57,23,42,
wb = Fit_Weibull_2P(failures=data)
plt.show()
The above probability plot is the typical way to visualise how the CDF (the blue line) models the failure data (the black points). If you would like to view the failure points alongside the PDF, CDF, SF, HF, or CHF without the axis being scaled then you can generate the scatter plot using the function plot_points which is available within reliability.Probability_plotting. In the example below we create some data, then fit a Weibull distribution to the data (ensuring we turn off the probability plot). From the fitted distribution object we plot the Survival Function (SF). We then use plot_points to generate a scatter plot of the plotting positions for the survival function.

For the function plot_points the inputs are:

- failures - an array or list of failure data
- right_censored - an array or list of right censored data. Optional input
- func - the function to be plotted. Must be ‘PDF’, ‘CDF’, ‘SF’, ‘HF’, or ‘CHF’. Default is ‘CDF’. Note that the options for ‘PDF’ and ‘HF’ will look much more scattered as they are found using the integral of a non-continuous function.
• a - this is the plotting heuristic. Default is 0.3. See probability plotting and Wikipedia for more details.

• keywords for the scatter plot are also accepted.

```python
from reliability.Distributions import Weibull_Distribution
from reliability.Fitters import Fit_Weibull_2P
from reliability.Probability_plotting import plot_points
import matplotlib.pyplot as plt

data = Weibull_Distribution(alpha=25, beta=4).random_samples(30)
weibull_fit = Fit_Weibull_2P(failures=data, show_probability_plot=False, print_results=False)
weibull_fit.distribution.SF(label='Fitted Distribution', color='steelblue')
plot_points(failures=data, func='SF', label='failure data', color='red', alpha=0.7)
plt.legend()
plt.show()```

It is beneficial to see the effectiveness of the fitted distribution in comparison to the original distribution. In this second example, we are creating 500 samples from a Weibull distribution and then we will right censor all of the data above our chosen threshold. Then we are fitting a Weibull_3P distribution to the data. Note that we need to specify “show_probability_plot=False, print_results=False” in the Fit_Weibull_3P to prevent the normal outputs of the fitting function from being displayed.

```python
from reliability.Distributions import Weibull_Distribution
from reliability.Fitters import Fit_Weibull_3P
from reliability.Other_functions import make_right_censored_data, histogram
```

(continues on next page)
import matplotlib.pyplot as plt

a = 30
b = 2
g = 20
threshold=55
dist = Weibull_Distribution(alpha=a, beta=b, gamma=g)  # generate a weibull distribution
raw_data = dist.random_samples(500, seed=2)  # create some data from the distribution
data = make_right_censored_data(raw_data, threshold=threshold)  # right censor some of the data
print('There are', len(data.right_censored), 'right censored items.')

wbf = Fit_Weibull_3P(failures=data.failures, right_censored=data.right_censored, show_probability_plot=False, print_results=False)  # fit the Weibull_3P distribution
print('Fit_Weibull_3P parameters:

Alpha:', wbf.alpha, '

Beta:', wbf.beta, '

Gamma', wbf.gamma)
histogram(raw_data, white_above=threshold)  # generates the histogram using optimal bin width and shades the censored part as white
dist.PDF(label='True Distribution')  # plots the true distribution's PDF
wbf.distribution.PDF(label='Fit_Weibull_3P', linestyle='--')  # plots to PDF of the fitted Weibull_3P
plt.title('Fitting comparison for failures and right censored data')
plt.legend()
plt.show()

There are 118 right censored items.
Fit_Weibull_3P parameters:
Alpha: 28.87483648004299
Beta: 2.0295019039127347
Gamma 20.383900235710296
As another example, we will fit a Gamma_2P distribution to some partially right censored data. To provide a comparison of the fitting accuracy as the number of samples increases, we will do the same experiment with varying sample sizes. The results highlight that the accuracy of the fit is proportional to the amount of samples, so you should always try to obtain more data if possible.

```python
from reliability.Distributions import Gamma_Distribution
from reliability.Fitters import Fit_Gamma_2P
from reliability.Other_functions import make_right_censored_data, histogram
import matplotlib.pyplot as plt

a = 30
b = 4
threshold = 180  # this is used when right censoring the data
trials = [10, 100, 1000, 10000]
subplot_id = 221
plt.figure(figsize=(9, 7))
for sample_size in trials:
    dist = Gamma_Distribution(alpha=a, beta=b)
    raw_data = dist.random_samples(sample_size, seed=2)  # create some data. Seeded for repeatability
    data = make_right_censored_data(raw_data, threshold=threshold)  # right censor the data
    gf = Fit_Gamma_2P(failures=data.failures, right_censored=data.right_censored,
                      show_probability_plot=False, print_results=False)  # fit the Gamma_2P distribution
    print('Fit_Gamma_2P parameters using', sample_size, 'samples:', '
Alpha:', gf.alpha, '
Beta:', gf.beta)  # print the results
```

(continues on next page)
plt.subplot(subplot_id)
    histogram(raw_data, white_above=threshold) # plots the histogram using optimal bin width and shades the right censored part white
dist.PDF(label='True') # plots the true distribution
gf.distribution.PDF(label='Fitted', linestyle='--') # plots the fitted Gamma_2P distribution
    plt.title(str(str(sample_size) + ' samples
' + r'$\alpha$ error: ' + str(round(abs(gf.alpha - a) / a * 100, 2)) + '%
' + r'$\beta$ error: ' + str(round(abs(gf.beta - b) / b * 100, 2)) + '%'))
    plt.ylim([0, 0.012])
    plt.xlim([0, 500])
    plt.legend()
    subplot_id += 1
plt.subplots_adjust(left=0.11, bottom=0.08, right=0.95, top=0.89, wspace=0.33, hspace=0.58)
plt.show()

Fit_Gamma_2P parameters using 10 samples:
Alpha: 19.426036067937076
Beta: 4.690126148584235

Fit_Gamma_2P parameters using 100 samples:
Alpha: 36.26423078012859
Beta: 3.292935791420746

Fit_Gamma_2P parameters using 1000 samples:
Alpha: 28.825237245698354
Beta: 4.062929181298052

Fit_Gamma_2P parameters using 10000 samples:
Alpha: 30.30127379917658
Beta: 3.96008626272073

...
To obtain details of the percentiles (lower estimate, point estimate, upper estimate), we can use the percentiles input for each Fitter. In this example, we will create some data and fit a Weibull_2P distribution. When percentiles are requested the results printed includes both the table of results and the table of percentiles. Setting percentiles as True will use a default list of percentiles (as shown in the first output). Alternatively we can specify the exact percentiles to use (as shown in the second output). The use of the crosshairs function is also shown which was used to annotate the plot manually. Note that the percentiles provided are the percentiles of the confidence intervals on time. Percentiles for the confidence intervals on reliability are not implemented, but can be accessed manually from the plots using the crosshairs function when confidence intervals on reliability have been plotted.

```python
from reliability.Distributions import Weibull_Distribution
from reliability.Fitters import Fit_Weibull_2P
from reliability.Other_functions import crosshairs
import matplotlib.pyplot as plt

dist = Weibull_Distribution(alpha=500, beta=6)
data = dist.random_samples(50, seed=1)  # generate some data
# this will produce the large table of percentiles below the first table of results
Fit_Weibull_2P(failures=data, percentiles=True, CI=0.8, show_probability_plot=False)
print('----------------------------------------------------------')
# repeat the process but using specified percentiles.
output = Fit_Weibull_2P(failures=data, percentiles=[5, 50, 95], CI=0.8)
# these points have been manually annotated on the plot using crosshairs
crosshairs()
```

(continues on next page)
plt.show()

# the values from the percentiles dataframe can be extracted as follows:
lower_estimates = output.percentiles['Lower Estimate'].values
print('
Lower estimates:',lower_estimates)

Results from Fit_Weibull_2P (80% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>489.117377</td>
<td>13.921709</td>
<td>471.597466</td>
<td>507.288155</td>
</tr>
<tr>
<td>Beta</td>
<td>5.207995</td>
<td>0.589270</td>
<td>4.505014</td>
<td>6.020673</td>
</tr>
</tbody>
</table>

Log-Likelihood: -301.6579198472162

Table of percentiles (80% CI bounds on time):

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Lower Estimate</th>
<th>Point Estimate</th>
<th>Upper Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>175.215220</td>
<td>202.211975</td>
<td>233.368328</td>
</tr>
<tr>
<td>5</td>
<td>250.234954</td>
<td>276.521341</td>
<td>305.569030</td>
</tr>
<tr>
<td>10</td>
<td>292.686602</td>
<td>317.508291</td>
<td>344.435017</td>
</tr>
<tr>
<td>20</td>
<td>344.277189</td>
<td>366.718796</td>
<td>390.623252</td>
</tr>
<tr>
<td>25</td>
<td>363.578675</td>
<td>385.050426</td>
<td>407.790228</td>
</tr>
<tr>
<td>50</td>
<td>437.690233</td>
<td>455.879011</td>
<td>474.823648</td>
</tr>
<tr>
<td>75</td>
<td>502.940450</td>
<td>520.776175</td>
<td>539.244407</td>
</tr>
<tr>
<td>80</td>
<td>517.547404</td>
<td>535.916489</td>
<td>554.937539</td>
</tr>
<tr>
<td>90</td>
<td>553.266964</td>
<td>574.067575</td>
<td>595.650206</td>
</tr>
<tr>
<td>95</td>
<td>580.174021</td>
<td>603.820155</td>
<td>628.430033</td>
</tr>
<tr>
<td>99</td>
<td>625.681232</td>
<td>655.789604</td>
<td>687.346819</td>
</tr>
</tbody>
</table>

Results from Fit_Weibull_2P (80% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>489.117377</td>
<td>13.921709</td>
<td>471.597466</td>
<td>507.288155</td>
</tr>
<tr>
<td>Beta</td>
<td>5.207995</td>
<td>0.589270</td>
<td>4.505014</td>
<td>6.020673</td>
</tr>
</tbody>
</table>

Log-Likelihood: -301.6579198472162

Table of percentiles (80% CI bounds on time):

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Lower Estimate</th>
<th>Point Estimate</th>
<th>Upper Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>250.234954</td>
<td>276.521341</td>
<td>305.569030</td>
</tr>
<tr>
<td>50</td>
<td>437.690233</td>
<td>455.879011</td>
<td>474.823648</td>
</tr>
<tr>
<td>95</td>
<td>580.174021</td>
<td>603.820155</td>
<td>628.430033</td>
</tr>
</tbody>
</table>

Lower estimates: [250.23495375 437.69023325 580.17402096]
6.1 Using Fit_Weibull_2P_grouped for large data sets

The function Fit_Weibull_2P_grouped is effectively the same as Fit_Weibull_2P, except for a few small differences that make it more efficient at handling grouped data sets. Grouped data sets are typically found in very large data that may be heavily censored. The function includes a choice between two optimizers and a choice between two initial guess methods for the initial guess that is given to the optimizer. These help in cases where the data is very heavily censored (>99.9%). The defaults for these options are usually the best but you may want to try different options to see which one gives you the lowest log-likelihood. The inputs and outputs are the same as for Fit_Weibull_2P except for the following:

- initial_guess_method - 'scipy' OR 'least squares'. Default is 'least squares'. Both do not take into account censored data but scipy uses MLE, and least squares is least squares regression of the plotting positions. Least
squares proved more accurate during testing.

- optimizer - ‘L-BFGS-B’ or ‘TNC’. These are both bound-constrained methods. If the bounded method fails, nelder-mead will be used. If nelder-mead fails then the initial guess will be returned with a warning. For more information on optimizers see the scipy documentation.

- dataframe - a pandas dataframe of the appropriate format. The requirements of the input dataframe are: The column titles MUST be ‘category’, ‘time’, ‘quantity’. The category values MUST be ‘F’ for failure or ‘C’ for censored (right censored). The time values are the failure or right censored times. The quantity is the number of items at that time. The quantity must be specified for all values even if the quantity is 1.

The following example shows how we can use Fit_Weibull_2P_grouped to fit a Weibull_2P distribution to grouped data from a spreadsheet (shown below) on the Windows desktop. We change the optimiser from the default (L-BFGS-B) to TNC as it is more successful for this dataset. In almost all cases the L-BFGS-B optimizer is better than TNC but it is worth trying both if the first does not look good. You may also want to try changing the initial_guess_method as the results from the optimizers can be sensitive to their initial guess for problems in which there are local minima or insufficient gradients to find the global minima. If you would like to access this data, it is available in reliability.Datasets.electronics and includes both the failures and right_censored format as well as the dataframe format. An example of this is provided in the code below (option 2).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>220</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>179</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>123</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>146</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>193</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>181</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>191</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>216</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>73</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>4479</td>
<td>817</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>6271</td>
<td>823</td>
<td>C</td>
</tr>
<tr>
<td>14</td>
<td>8147</td>
<td>815</td>
<td>C</td>
</tr>
<tr>
<td>15</td>
<td>8063</td>
<td>813</td>
<td>C</td>
</tr>
<tr>
<td>16</td>
<td>62716</td>
<td>804</td>
<td>C</td>
</tr>
</tbody>
</table>

```python
from reliability.Fitters import Fit_Weibull_2P_grouped
import pandas as pd

# option 1 for importing this dataset (from an excel file on your desktop)
filename = 'C:\Users\Current User\Desktop\data.xlsx'
df = pd.read_excel(io=filename)

# option 2 for importing this dataset (from the dataset in reliability)
# from reliability.Datasets import electronics
# df = electronics().dataframe

print(df.head(15),'

Fit_Weibull_2P_grouped(dataframe=df, optimizer='TNC', show_probability_plot=False)

#note that the TNC optimiser usually underperforms the default (L-BFGS-B) optimiser
#but in this case it is better
```

(continues on next page)
```
<table>
<thead>
<tr>
<th>time</th>
<th>quantity</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>179</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>123</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>146</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>199</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>181</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>191</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>216</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>73</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>44798</td>
<td>C</td>
</tr>
<tr>
<td>11</td>
<td>62715</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td>81474</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>80632</td>
<td>C</td>
</tr>
<tr>
<td>14</td>
<td>62716</td>
<td>C</td>
</tr>
</tbody>
</table>

Results from Fit_Weibull_2P_grouped (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>6.205380e+21</td>
<td>7.780317e+22</td>
<td>1.319435e+11</td>
<td>2.918427e+32</td>
</tr>
<tr>
<td>Beta</td>
<td>1.537356e-01</td>
<td>4.863029e-02</td>
<td>8.270253e-02</td>
<td>2.857789e-01</td>
</tr>
</tbody>
</table>

Log-Likelihood: -144.61675864154972
Number of failures: 10
Number of right censored: 4072
Fraction censored: 99.75502 %
```

6.2 How does the code work with censored data?

All functions in this module work using a Python library called autograd to find the derivative of the log-likelihood function. In this way, the code only needs to specify the log PDF and log SF in order to apply Maximum-Likelihood Estimation (MLE) to obtain the fitted parameters. Initial guesses of the parameters are essential for autograd and are obtained using scipy.stats on all the data as if it wasn’t censored (since scipy doesn’t accept censored data). If the distribution is an extremely bad fit or is heavily censored (>99% censored) then these guesses may be poor and the fit might not be successful. In this case, the scipy fit is used which will be incorrect if there is any censored data. If this occurs, a warning will be printed. Generally the fit achieved by autograd is highly successful.

A special thanks goes to Cameron Davidson-Pilon (author of the Python library lifelines and website dataorigami.net) for providing help with getting autograd to work, and for writing the python library autograd-gamma, without which it would be impossible to fit the Beta or Gamma distributions using autograd.
6.2. How does the code work with censored data?
Fitting all available distributions to data

To fit all of the distributions available in reliability, is a similar process to fitting a specific distribution. The user needs to specify the failures and any right censored data. The Beta distribution will only be fitted if you specify data that is in the range 0 to 1. The selection of what can be fitted is all done automatically based on the data provided. Manual exclusion of certain datasets is also possible. If you only provide 2 failures the 3P distributions will automatically be excluded from the fitting process.

Inputs:
- failures - an array or list of the failure times.
- right_censored - an array or list of the right censored failure times.
- sort_by - goodness of fit test to sort results by. Must be either ‘BIC’, ‘AIC’, or ‘AD’. Default is BIC.
- exclude - list or array of strings specifying which distributions to exclude. Default is None. Options are Weibull_2P, Weibull_3P, Normal_2P, Gamma_2P, Loglogistic_2P, Gamma_3P, Lognormal_2P, Lognormal_3P, Loglogistic_3P, Gumbel_2P, Exponential_2P, Exponential_1P, Beta_2P
- show_histogram_plot - True/False. Defaults to True. Will show the PDF and CDF of the fitted distributions along with a histogram of the failure data.
- show_PP_plot - True/False. Defaults to True. Provides a comparison of parametric vs non-parametric fit using a semiparametric Probability-Probability (PP) plot for each fitted distribution.
- show_probability_plot - True/False. Defaults to True. Provides a probability plot of each of the fitted distributions.
- print_results - True/False. Defaults to True. Will show the results of the fitted parameters and the goodness of fit tests in a dataframe.

Outputs:
- results - a dataframe of the fitted distributions and their parameters, along with the AICc, BIC and AD goodness of fit statistics. This is sorted automatically to provide the best fit first. Use the sort_by='BIC' to change the sort between AICc, BIC, and AD. Default sort is BIC. print_results controls whether this is printed. In displaying these results, the pandas dataframe is designed to use the common greek letter parametrisations rather than the scale, shape, location, threshold parametrisations which can become confusing for some distributions.
• a plot of the PDF and CDF of each fitted distribution along with a histogram of the failure data. The legend is not in any particular order.
• a probability plot of each of the fitted distributions.
• a semi-parametric probability-probability plot of parametric vs non-parametric distribution (a better fit is will lie on the red diagonal).
• best_distribution - a distribution object created based on the parameters of the best fitting distribution. The best distribution is created as a distribution object that can be used like any of the other distribution objects. See the examples below for how this can be used.
• best_distribution_name - the name of the best fitting distribution. E.g. ‘Weibull_3P’
• parameters and goodness of fit tests for each fitted distribution. For example, the Weibull_3P distribution values are: Weibull_3P_alpha, Weibull_3P_beta, Weibull_3P_gamma, Weibull_3P_BIC, Weibull_3P_AICc, Weibull_3P_AD.

Confidence intervals are shown on the plots but they are not reported for each of the fitted parameters as this would be a large number of outputs. If you need the confidence intervals for the fitted parameters you can repeat the fitting using just a specific distribution and the results will include the confidence intervals.

In this first example, we will use Fit_Everything on some data and will return only the dataframe of results. Note that we are actively supressing the 3 plots that would normally be shown to provide graphical goodness of fit indications.

```python
from reliability.Fitters import Fit_Everything
data = [4,4,2,4,7,4,1,2,7,1,4,3,6,6,6,3,2,3,4,3,2,4,6,5,2,4,3] # created using Weibull_Distribution(alpha=5,beta=2), and rounded to nearest int
Fit_Everything(failures=data, show_histogram_plot=False, show_probability_plot=False, show_PP_plot=False)
```

```
  Distribution    BIC   Alpha   Beta  Gamma  Mu  Sigma  Lambda  AICc
Weibull_2P      120.054175  4.21932  2.43761
Weibull_3P      120.76216  3.61252  2.02388  0.530239
Gamma_2P        120.816685  0.816685  4.57132
Gamma_3P        123.020072  3.49645  0.781773  0.9999
Normal_2P       122.055543  1.185387
Normal_3P       123.020072  1.198573
Lognormal_2P    123.020072  3.45096  3.48793
Lognormal_3P    123.020072  3.45096  3.48793
Loglogistic_2P  123.020072  3.61252  2.02388  0.530239
Loglogistic_3P  123.020072  3.61252  2.02388  0.530239
Exponential_2P  123.047337  0.999
Exponential_3P  123.047337  0.999
Gamma_3P        129.222968  3.73333  1.65193
Gamma_3P        129.222968  3.73333  1.65193
Exponential_1P  142.439287  4.710926
```

In this second example, we will create some right censored data and use Fit_Everything. All outputs are shown, and
The best fitting distribution is accessed and printed.

```python
from reliability.Fitters import Fit_Everything
from reliability.Distributions import Weibull_Distribution
from reliability.Other_functions import make_right_censored_data

raw_data = Weibull_Distribution(alpha=12, beta=3).random_samples(100, seed=2)  # create some data
data = make_right_censored_data(raw_data, threshold=14)  # right censor the data
results = Fit_Everything(failures=data.failures, right_censored=data.right_censored)  # fit all the models
print('The best fitting distribution was', results.best_distribution_name, 'which had parameters', results.best_distribution.parameters)
```

```

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Alpha</th>
<th>Beta</th>
<th>Gamma</th>
<th>Mu</th>
<th>Sigma</th>
<th>Lambda</th>
<th>AICc</th>
<th>BIC</th>
<th>AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weibull_2P</td>
<td>11.2773</td>
<td>3.30301</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>488.041154</td>
<td>493.127783</td>
<td>44.945028</td>
</tr>
<tr>
<td>Normal_2P</td>
<td>493.127783</td>
<td>44.945028</td>
<td>10.1194</td>
<td>3.37466</td>
<td></td>
<td></td>
<td>489.082213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gamma_2P</td>
<td>494.168842</td>
<td>44.909765</td>
<td>1.42315</td>
<td>7.21352</td>
<td></td>
<td></td>
<td>490.593729</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loglogistic_2P</td>
<td>495.680358</td>
<td>45.281749</td>
<td>9.86245</td>
<td>4.48433</td>
<td></td>
<td></td>
<td>491.300512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weibull_3P</td>
<td>496.387141</td>
<td>45.200181</td>
<td>10.0786</td>
<td>2.85825</td>
<td>1.15083</td>
<td></td>
<td>492.720018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal_3P</td>
<td>497.372839</td>
<td>44.992658</td>
<td>1.42315</td>
<td>7.21352</td>
<td></td>
<td>0</td>
<td>492.720018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lognormal_2P</td>
<td>500.285528</td>
<td>45.281749</td>
<td></td>
<td>2.26524</td>
<td>0.406436</td>
<td></td>
<td>495.693518</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lognormal_3P</td>
<td>500.780147</td>
<td>45.687381</td>
<td>0.883941</td>
<td>2.16125</td>
<td>0.465752</td>
<td></td>
<td>500.938298</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loglogistic_3P</td>
<td>500.780147</td>
<td>45.687381</td>
<td>9.86245</td>
<td>4.48433</td>
<td></td>
<td>0</td>
<td>493.426801</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponential_2P</td>
<td>500.992311</td>
<td>45.200181</td>
<td>2.82802</td>
<td>0.121869</td>
<td>538.150905</td>
<td></td>
<td>538.150905</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponential_1P</td>
<td>543.237534</td>
<td>51.777617</td>
<td>0.0870022</td>
<td>594.033742</td>
<td></td>
<td></td>
<td>594.033742</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponential_1P</td>
<td>596.598095</td>
<td>56.866106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>596.598095</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The best fitting distribution was Weibull_2P which had parameters [11.27730642  3.30300716  0. ]
```
Chapter 7. Fitting all available distributions to data
Semi-parametric Probability-Probability plots of each fitted distribution
Parametric (x-axis) vs Non-Parametric (y-axis)

All plots are ordered based on the goodness of fit order of the results. For the histogram this is reflected in the order of the legend. For the probability plots and PP plots, these are ordered left to right and top to bottom.

The histogram is scaled based on the amount of censored data. If your censored data is all above your failure data then the histogram bars should line up well with the fitted distributions (assuming you have enough data). However, if your censored data is not always greater than the max of your failure data then the heights of the histogram bars will be scaled down and the plot may look incorrect. This is to be expected as the histogram is only a plot of the failure data and the totals will not add to 100% if there is censored data.

**Note:** The confidence intervals shown on the probability plots are not available for Gamma_2P, Gamma_3P, or Beta_2P. This library is being actively developed the remaining confidence intervals will be added soon.
CHAPTER 8

Mixture models

8.1 What are mixture models?

Mixture models are a combination of two or more distributions added together to create a distribution that has a shape with more flexibility than a single distribution. Each of the mixture’s components must be multiplied by a proportion, and the sum of all the proportions is equal to 1. The mixture is generally written in terms of the PDF, but since the CDF is the integral (cumulative sum) of the PDF, we can equivalently write the Mixture model in terms of the PDF or CDF. For a mixture model with 2 distributions, the equations are shown below:

\[
PDF_{mixture} = p \times PDF_1 + (1 - p) \times PDF_2
\]

\[
CDF_{mixture} = p \times CDF_1 + (1 - p) \times CDF_2
\]

\[
SF_{mixture} = 1 - CDF_{mixture}
\]

\[
HF_{mixture} = \frac{PDF_{mixture}}{SF_{mixture}}
\]

\[
CHF_{mixture} = -\ln(SF_{mixture})
\]

Mixture models are useful when there is more than one failure mode that is generating the failure data. This can be recognised by the shape of the PDF and CDF being outside of what any single distribution can accurately model. On a probability plot, a mixture of failure modes can be identified by bends or S-shapes in the data that you might otherwise expect to be linear. An example of this is shown in the image below. You should not use a mixture model just because it can fit almost anything really well, but you should use a mixture model if you suspect that there are multiple failure modes contributing to the failure data you are observing. To judge whether a mixture model is justified, look at the goodness of fit criterion (AICc or BIC) which penalises the score based on the number of parameters in the model. The closer the goodness of fit criterion is to zero, the better the fit. Using AD or log-likelihood for this check is not appropriate as these goodness of fit criterions do not penalise the score based on the number of parameters in the model and are therefore prone to overfitting.

See also competing risk models for another method of combining distributions using the product of the SF rather than the sum of the CDF.
8.2 Creating a mixture model

Within `reliability.Distributions` is the `Mixture_Model`. This function accepts an array or list of distribution objects created using the `reliability.Distributions` module (available distributions are Exponential, Weibull, Gumbel, Normal, Lognormal, Logistic, Gamma, Beta). There is no limit to the number of components you can add to the mixture, but it is generally preferable to use as few as are required to fit the data appropriately (typically 2 or 3). In addition to the distributions, you can specify the proportions contributed by each distribution in the mixture. These proportions must sum to 1. If not specified the proportions will be set as equal for each component.

As this process is additive for the survival function, and may accept many distributions of different types, the mathematical formulation quickly gets complex. For this reason, the algorithm combines the models numerically rather than empirically so there are no simple formulas for many of the descriptive statistics (mean, median, etc.). Also, the accuracy of the model is dependent on xvals. If the xvals array is small (<100 values) then the answer will be “blocky” and inaccurate. The variable xvals is only accepted for PDF, CDF, SF, HF, and CHF. The other methods (like random samples) use the default xvals for maximum accuracy. The default number of values generated when xvals is not given is 1000. Consider this carefully when specifying xvals in order to avoid inaccuracies in the results.

The API is similar to the other probability distributions (Weibull, Normal, etc.) and has the following inputs and methods:

Inputs:
- distributions - a list or array of probability distributions used to construct the model.
- proportions - how much of each distribution to add to the mixture. The sum of proportions must always be 1.
Methods:

- name - ‘Mixture’
- name2 - ‘Mixture using 3 distributions’
- mean
- median
- mode
- variance
- standard_deviation
- skewness
- kurtosis
- excess_kurtosis
- b5 - The time where 5% have failed. Same as quantile(0.05)
- b95 - The time where 95% have failed. Same as quantile(0.95)
- plot() - plots all functions (PDF,CDF,SF,HF,CHF)
- PDF() - plots the probability density function
- CDF() - plots the cumulative distribution function
- SF() - plots the survival function (also known as reliability function)
- HF() - plots the hazard function
- CHF() - plots the cumulative hazard function
- quantile() - Calculates the quantile (time until a fraction has failed) for a given fraction failing. Also known as b life where b5 is the time at which 5% have failed.
- inverse_SF() - the inverse of the Survival Function. This is useful when producing QQ plots.
- mean_residual_life() - Average residual lifetime of an item given that the item has survived up to a given time. Effectively the mean of the remaining amount (right side) of a distribution at a given time.
- stats() - prints all the descriptive statistics. Same as the statistics shown using .plot() but printed to console.
- random_samples() - draws random samples from the distribution to which it is applied.

The following example shows how the Mixture_Model object can be created, visualised and used.

```python
from reliability.Distributions import Lognormal_Distribution, Gamma_Distribution,
˓→Weibull_Distribution, Mixture_Model
import matplotlib.pyplot as plt

# create the mixture model
d1 = Lognormal_Distribution(mu=2, sigma=0.8)
d2 = Weibull_Distribution(alpha=50, beta=5, gamma=100)
d3 = Gamma_Distribution(alpha=5, beta=3, gamma=30)
mixture_model = Mixture_Model(distributions=[d1, d2, d3], proportions=[0.3, 0.4, 0.3])

# plot the 5 functions using the plot() function
mixture_model.plot()

# plot the PDF and CDF

(continues on next page)
```
plot_components = True # this plots the component distributions. Default is False
plt.figure(figsize=(9, 5))
plt.subplot(121)
mixture_model.PDF(plot_components=plot_components, color='red', linestyle='--')
plt.subplot(122)
mixture_model.CDF(plot_components=plot_components, color='red', linestyle='--')
plt.subplots_adjust(left=0.1, right=0.95)
plt.show()

# extract the mean of the distribution
print('The mean of the distribution is:', mixture_model.mean)

... The mean of the distribution is: 74.91674657035722 ...

Mixture Model

Probability Density Function

Cumulative Distribution Function

Survival Function

Hazard Function

Cumulative Hazard Function

Mean = 74.9161
Median = 47.4346
Mode = 3.8959
5th quantile = 3.408
95th quantile = 157.8863
Standard deviation = 60.30283
Variance = 3636.4311
Skewness = 0.2589
Excess kurtosis = -1.6672
8.3 Fitting a mixture model

Within reliability.Fitters is Fit_Weibull_Mixture. This function will fit a weibull mixture model consisting of 2 x Weibull_2P distributions (this does not fit the gamma parameter). Just as with all of the other distributions in reliability.Fitters, right censoring is supported, though care should be taken to ensure that there still appears to be two groups when plotting only the failure data. A second group cannot be made from a mostly or totally censored set of samples.

Whilst some failure modes may not be fitted as well by a Weibull distribution as they may be by another distribution, it is unlikely that a mixture of data from two distributions (particularly if they are overlapping) will be fitted noticeably better by other types of mixtures than would be achieved by a Weibull mixture. For this reason, other types of mixtures are not implemented.

Inputs:
- failures - an array or list of the failure data. There must be at least 4 failures, but it is highly recommended to use another model if you have less than 20 failures.
- right_censored - an array or list of right censored failure data. Optional input.
- print_results - True/False. This will print results to console. Default is True
- CI - confidence interval for estimating confidence limits on parameters. Must be between 0 and 1. Default is 0.95 for 95% CI.
- show_probability_plot - True/False. This will show the probability plot with the fitted mixture CDF. Default is True.

Outputs:
- alpha_1 - the fitted Weibull_2P alpha parameter for the first (left) group
- beta_1 - the fitted Weibull_2P beta parameter for the first (left) group
- alpha_2 - the fitted Weibull_2P alpha parameter for the second (right) group
from reliability.Fitters import Fit_Weibull_Mixture
from reliability.Distributions import Weibull_Distribution
from reliability.Other_functions import histogram
import numpy as np
import matplotlib.pyplot as plt

# create some failures from two distributions
group_1 = Weibull_Distribution(alpha=10, beta=3).random_samples(40, seed=2)
group_2 = Weibull_Distribution(alpha=40, beta=4).random_samples(60, seed=2)
all_data = np.hstack([group_1, group_2])  # combine the data
results = Fit_Weibull_Mixture(failures=all_data)  # fit the mixture model

# this section is to visualise the histogram with PDF and CDF
# it is not part of the default output from the Fitter
plt.figure(figsize=(9, 5))
plt.subplot(121)
histogram(all_data)
results.distribution.PDF()
plt.subplot(122)
histogram(all_data, cumulative=True)
results.distribution.CDF()
plt.show()

Results from Fit_Weibull_Mixture (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 1</td>
<td>8.654923</td>
<td>0.394078</td>
<td>7.916006</td>
<td>9.462815</td>
</tr>
<tr>
<td>Beta 1</td>
<td>3.910594</td>
<td>0.509724</td>
<td>3.028959</td>
<td>5.048845</td>
</tr>
<tr>
<td>Alpha 2</td>
<td>38.097040</td>
<td>1.411773</td>
<td>35.428112</td>
<td>40.967028</td>
</tr>
<tr>
<td>Beta 2</td>
<td>3.818227</td>
<td>0.421366</td>
<td>3.075574</td>
<td>4.740207</td>
</tr>
<tr>
<td>Proportion</td>
<td>0.388206</td>
<td>0.050264</td>
<td>0.295325</td>
<td>0.489987</td>
</tr>
</tbody>
</table>

Log-Likelihood: -375.9906311550037
Chapter 8. Mixture models
In this second example, we will compare how well the Weibull Mixture performs vs a single Weibull_2P. Firstly, we generate some data from two Weibull distributions, combine the data, and right censor it above our chosen threshold. Next, we will fit the Mixture and Weibull_2P distributions. Then we will visualise the histogram and PDF of the fitted mixture model and Weibull_2P distributions. The goodness of fit measure is used to check whether the mixture model is really a much better fit than a single Weibull_2P distribution (which it is due to the lower BIC).

```python
from reliability.Fitters import Fit_Weibull_Mixture, Fit_Weibull_2P
from reliability.Distributions import Weibull_Distribution
from reliability.Other_functions import histogram, make_right_censored_data
import numpy as np
import matplotlib.pyplot as plt

# create some failures and right censored data
group_1 = Weibull_Distribution(alpha=10, beta=2).random_samples(700, seed=2)
group_2 = Weibull_Distribution(alpha=30, beta=3).random_samples(300, seed=2)
all_data = np.hstack([group_1, group_2])
data = make_right_censored_data(all_data, threshold=30)

# fit the Weibull Mixture and Weibull_2P
mixture = Fit_Weibull_Mixture(failures=data.failures, right_censored=data.right_censored, show_probability_plot=False, print_results=False)
single = Fit_Weibull_2P(failures=data.failures, right_censored=data.right_censored, show_probability_plot=False, print_results=False)
print('Weibull_Mixture BIC:', mixture.BIC, '
Weibull_2P BIC:', single.BIC)  # print the goodness of fit measure

# plot the Mixture and Weibull_2P
histogram(all_data, white_above=30)
mixture.distribution.PDF(label='Weibull Mixture')
single.distribution.PDF(label='Weibull_2P')
plt.title('Comparison of Weibull_2P with Weibull Mixture')
plt.legend()
plt.show()
```

(continues on next page)
Comparison of Weibull_2P with Weibull Mixture

Weibull_Mixture BIC: 6432.417425636481
Weibull_2P BIC: 6511.51175959736
CHAPTER 9

Competing risks models

9.1 What are competing risks models?

Competing risks models are a combination of two or more distributions that represent failure modes which are “competing” to end the life of the system being modelled. This model is similar to a mixture model in the sense that it uses multiple distributions to create a new model that has a shape with more flexibility than a single distribution. However, unlike a mixture models, we are not adding proportions of the PDF or CDF, but are instead multiplying the survival functions. The formula for the competing risks model is typically written in terms of the survival function (SF). Since we may consider the system’s reliability to depend on the reliability of all the parts of the system (each with its own failure modes), the equation is written as if the system was in series, using the product of the survival functions for each failure mode. For a competing risks model with 2 distributions, the equations are shown below:

\[ SF_{\text{Competing Risks}} = SF_1 \times SF_2 \]
\[ CDF_{\text{Competing Risks}} = 1 - SF_{\text{Competing Risks}} \]

Since \( SF = \exp(-CHF) \) we may equivalently write the competing risks model in terms of the hazard or cumulative hazard function as:

\[ HF_{\text{Competing Risks}} = HF_1 + HF_2 \]
\[ CHF_{\text{Competing Risks}} = CHF_1 + CHF_2 \]
\[ PDF_{\text{Competing Risks}} = HF_{\text{Competing Risks}} \times SF_{\text{Competing Risks}} \]

The image below illustrates the difference between the competing risks model and the mixture model, each of which is made up of the same two component distributions. Note that the PDF of the competing risks model is always equal to or to the left of the component distributions, and the CDF is equal to or higher than the component distributions. This shows how a failure mode that occurs earlier in time can end the lives of units under observation before the second failure mode has the chance to. This behaviour is characteristic of real systems which experience multiple failure modes, each of which could cause system failure.
Competing risks models are useful when there is more than one failure mode that is generating the failure data. This can be recognised by the shape of the PDF and CDF being outside of what any single distribution can accurately model. On a probability plot, a combination of failure modes can be identified by bends in the data that you might otherwise expect to be linear. An example of this is shown in the image below. You should not use a competing risks model just because it fits your data better than a single distribution, but you should use a competing risks model if you suspect that there are multiple failure modes contributing to the failure data you are observing. To judge whether a competing risks model is justified, look at the goodness of fit criterion (AICc or BIC) which penalises the score based on the number of parameters in the model. The closer the goodness of fit criterion is to zero, the better the fit. It is not appropriate to use the Log-likelihood or AD goodness of fit criterions as these do not penalise the score based on the number of parameters, therefore making the model susceptible to overfitting.

See also mixture models for another method of combining distributions using the sum of the CDF rather than the product of the SF.
9.2 Creating a competing risks model

Within reliability.Distributions is the Competing_Risks_Model. This function accepts an array or list of distribution objects created using the reliability.Distributions module (available distributions are Exponential, Weibull, Gumbel, Normal, Lognormal, Loglogistic, Gamma, Beta). There is no limit to the number of components you can add to the model, but it is generally preferable to use as few as are required to fit the data appropriately (typically 2 or 3). Unlike the mixture model, you do not need to specify any proportions.

As this process is multiplicative for the survival function (or additive for the hazard function), and may accept many distributions of different types, the mathematical formulation quickly gets complex. For this reason, the algorithm combines the models numerically rather than empirically so there are no simple formulas for many of the descriptive statistics (mean, median, etc.). Also, the accuracy of the model is dependent on xvals. If the xvals array is small (<100 values) then the answer will be “blocky” and inaccurate. The variable xvals is only accepted for PDF, CDF, SF, HF, and CHF. The other methods (like random samples) use the default xvals for maximum accuracy. The default number of values generated when xvals is not given is 1000. Consider this carefully when specifying xvals in order to avoid inaccuracies in the results.

The API is similar to the other probability distributions (Weibull, Normal, etc.) and has the following inputs and methods:

Inputs:
- distributions - a list or array of probability distributions used to construct the model

Methods:
- name - ‘Competing risks’
- name2 - ‘Competing risks using 3 distributions’
- mean
- median
- mode
- variance
- standard_deviation
- skewness
- kurtosis
- excess_kurtosis
- b5 - The time where 5% have failed. Same as quantile(0.05)
- b95 - The time where 95% have failed. Same as quantile(0.95)
- plot() - plots all functions (PDF,CDF,SF,HF,CHF)
- PDF() - plots the probability density function
- CDF() - plots the cumulative distribution function
- SF() - plots the survival function (also known as reliability function)
- HF() - plots the hazard function
- CHF() - plots the cumulative hazard function
- quantile() - Calculates the quantile (time until a fraction has failed) for a given fraction failing. Also known as b life where b5 is the time at which 5% have failed.
- inverse_SF() - the inverse of the Survival Function. This is useful when producing QQ plots.
- mean_residual_life() - Average residual lifetime of an item given that the item has survived up to a given time. Effectively the mean of the remaining amount (right side) of a distribution at a given time.
- stats() - prints all the descriptive statistics. Same as the statistics shown using .plot() but printed to console.
- random_samples() - draws random samples from the distribution to which it is applied.

The following example shows how the Competing_Risks_Model object can be created, visualised and used.

```python
from reliability.Distributions import Lognormal_Distribution, Gamma_Distribution, Weibull_Distribution, Competing_Risks_Model
import matplotlib.pyplot as plt

# create the competing risks model
d1 = Lognormal_Distribution(mu=4, sigma=0.1)
d2 = Weibull_Distribution(alpha=50, beta=2)
d3 = Gamma_Distribution(alpha=30,beta=1.5)
CR_model = Competing_Risks_Model(distributions=[d1, d2, d3])

# plot the 5 functions using the plot() function
CR_model.plot()

# plot the PDF and CDF
plot_components = True # this plots the component distributions. Default is False
plt.figure(figsize=(9, 5))
```

(continues on next page)
plt.subplot(121)
CR_model.PDF(plot_components=plot_components, color='red', linestyle='--')
plt.subplot(122)
CR_model.CDF(plot_components=plot_components, color='red', linestyle='--')
plt.show()

# extract the mean of the distribution
print('The mean of the distribution is:', CR_model.mean)

...  
The mean of the distribution is: 27.04449126273065

9.2. Creating a competing risks model
9.3 Fitting a competing risks model

Within reliability.Fitters is Fit_Weibull_CR. This function will fit a Weibull Competing Risks model consisting of 2 x Weibull_2P distributions (this does not fit the gamma parameter). Just as with all of the other distributions in reliability.Fitters, right censoring is supported.

Whilst some failure modes may not be fitted as well by a Weibull distribution as they may be by another distribution, it is unlikely that a competing risks model of data from two distributions (particularly if they are overlapping) will be fitted noticeably better by other types of competing risks models than would be achieved by a Weibull competing risks model. For this reason, other types of competing risks models are not implemented.

Inputs:

• failures - an array or list of the failure data. There must be at least 4 failures, but it is highly recommended to use another model if you have less than 20 failures.

• right_censored - an array or list of right censored data. Optional input.

• print_results - True/False. This will print results to console. Default is True.

• CI - confidence interval for estimating confidence limits on parameters. Must be between 0 and 1. Default is 0.95 for 95% CI.

• show_probability_plot - True/False. This will show the probability plot with the fitted Weibull_CR CDF. Default is True.

Outputs:

• alpha_1 - the fitted Weibull_2P alpha parameter for the first distribution

• beta_1 - the fitted Weibull_2P beta parameter for the first distribution

• alpha_2 - the fitted Weibull_2P alpha parameter for the second distribution

• beta_2 - the fitted Weibull_2P beta parameter for the second distribution
In this first example, we will create some data using a competing risks model from two Weibull distributions. We will then fit the Weibull mixture model to the data and will print the results and show the plot.

```python
from reliability.Distributions import Weibull_Distribution, Competing_Risks_Model
from reliability.Fitters import Fit_Weibull_CR
from reliability.Other_functions import histogram
import matplotlib.pyplot as plt

# create some data that requires a competing risks model
d1 = Weibull_Distribution(alpha=50, beta=2)
d2 = Weibull_Distribution(alpha=40, beta=10)
CR_model = Competing_Risks_Model(distributions=[d1, d2])
data = CR_model.random_samples(100, seed=2)

# fit the Weibull competing risks model
results = Fit_Weibull_CR(failures=data)

# this section is to visualise the histogram with PDF and CDF
# it is not part of the default output from the Fitter
plt.figure(figsize=(9, 5))
plt.subplot(121)
histogram(data)
results.distribution.PDF()
plt.subplot(122)
histogram(data, cumulative=True)
results.distribution.CDF()
plt.show()
```

(continues on next page)
Results from Fit_Weibull_CR (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 1</td>
<td>55.185550</td>
<td>14.385243</td>
<td>33.108711</td>
<td>91.983192</td>
</tr>
<tr>
<td>Beta 1</td>
<td>1.896577</td>
<td>0.454578</td>
<td>1.185637</td>
<td>3.033816</td>
</tr>
<tr>
<td>Alpha 2</td>
<td>38.192099</td>
<td>1.083595</td>
<td>36.126262</td>
<td>40.376067</td>
</tr>
<tr>
<td>Beta 2</td>
<td>7.978213</td>
<td>1.181428</td>
<td>5.968403</td>
<td>10.664810</td>
</tr>
</tbody>
</table>

Log-Likelihood: -352.47978488894165
In this second example, we will compare the mixture model to the competing risks model. The data is generated from a competing risks model so we expect the Weibull competing risks model to be more appropriate than the Mixture model. Through comparison of the AICc or BIC, we can see which model is more appropriate. Since the AICc and BIC penalise the goodness of fit criterion based on the number of parameters and the mixture model has 5 parameters compared to the competing risk model’s 4 parameters, we expect the competing risks model to have a lower (closer to zero) goodness of fit than the Mixture model, and this is what we observe in the results. Notice how the log-likelihood and AD statistics of the mixture model indicates a better fit (because the value is closer to zero), but this does not take into account the number of parameters in the model.

```python
from reliability.Distributions import Weibull_Distribution, Competing_Risks_Model
from reliability.Fitters import Fit_Weibull_CR, Fit_Weibull_Mixture
import matplotlib.pyplot as plt
import pandas as pd

# create some data from a competing risks model
d1 = Weibull_Distribution(alpha=250, beta=2)
d2 = Weibull_Distribution(alpha=210, beta=10)
CR_model = Competing_Risks_Model(distributions=[d1, d2])
data = CR_model.random_samples(50, seed=2)

CR_fit = Fit_Weibull_CR(failures=data)  # fit the Weibull competing risks model
MM_fit = Fit_Weibull_Mixture(failures=data)  # fit the Weibull mixture model
plt.legend()
plt.show()

# create a dataframe to display the goodness of fit criterion as a table
goodness_of_fit = {'Model': ['Competing Risks', 'Mixture'],
                   'AICc': [CR_fit.AICc, MM_fit.AICc],
                   'BIC': [CR_fit.BIC, MM_fit.BIC],
                   'AD': [CR_fit.AD, MM_fit.AD]}
df = pd.DataFrame(goodness_of_fit, columns=['Model', 'AICc', 'BIC', 'AD'])
print(df)

Results from Fit_Weibull_CR (95% CI):
```

(continues on next page)


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 1</td>
<td>229.832608</td>
<td>51.142184</td>
<td>148.594556</td>
<td>355.484273</td>
</tr>
<tr>
<td>Beta 1</td>
<td>2.501343</td>
<td>0.746502</td>
<td>1.393607</td>
<td>4.489586</td>
</tr>
<tr>
<td>Alpha 2</td>
<td>199.720752</td>
<td>8.561263</td>
<td>183.626538</td>
<td>217.225567</td>
</tr>
<tr>
<td>Beta 2</td>
<td>9.201575</td>
<td>2.200422</td>
<td>5.758505</td>
<td>14.703291</td>
</tr>
</tbody>
</table>

Log-Likelihood: -255.44383742100507

Results from Fit_Weibull_Mixture (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 1</td>
<td>110.618178</td>
<td>29.929617</td>
<td>65.090789</td>
<td>187.989443</td>
</tr>
<tr>
<td>Beta 1</td>
<td>3.632755</td>
<td>1.543561</td>
<td>1.579646</td>
<td>8.354346</td>
</tr>
<tr>
<td>Alpha 2</td>
<td>191.064502</td>
<td>3.430400</td>
<td>184.457963</td>
<td>197.907661</td>
</tr>
<tr>
<td>Beta 2</td>
<td>8.049745</td>
<td>1.161811</td>
<td>6.066373</td>
<td>10.681571</td>
</tr>
<tr>
<td>Proportion 1</td>
<td>0.248562</td>
<td>0.074431</td>
<td>0.131549</td>
<td>0.419395</td>
</tr>
</tbody>
</table>

Log-Likelihood: -254.69963111152975

<table>
<thead>
<tr>
<th>Model</th>
<th>AICc</th>
<th>BIC</th>
<th>AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Competing Risks</td>
<td>519.776564</td>
<td>526.535767</td>
<td>0.582669</td>
</tr>
<tr>
<td>1 Mixture</td>
<td>520.762899</td>
<td>528.959377</td>
<td>0.550508</td>
</tr>
</tbody>
</table>

Chapter 9. Competing risks models
9.3. Fitting a competing risks model
Probability plots are a general term for several different plotting techniques. One of these techniques is a graphical method for comparing two data sets and includes probability-probability (PP) plots and quantile-quantile (QQ) plots. The second plotting technique is used for assessing the goodness of fit of a distribution by plotting the empirical CDF of the failures against their failure time and scaling the axes in such a way that the distribution appears linear. This method allows the reliability analyst to fit the distribution parameters using a simple “least squares” fitting method for a straight line and was popular before computers were capable of calculating the MLE estimates of the parameters. While we do not typically favour the use of least squares as a fitting method, we can still use probability plots to assess the goodness of fit. The module `reliability.Probability_plotting` contains functions for each of the standard distributions supported in `reliability`. These functions are:

- Weibull_probability_plot
- Normal_probability_plot
- Lognormal_probability_plot
- Gamma_probability_plot
- Beta_probability_plot
- Exponential_probability_plot
- Exponential_probability_plot_Weibull_Scale
- Loglogistic_probability_plot
- Gumbel_probability_plot

There is also a function to obtain the plotting positions called plotting_positions. This function is mainly used by other functions and is not discussed further here. For more detail, consult the help file of the function. To obtain a scatter plot of the plotting positions in the form of the PDF, CDF, SF, HF, or CHF, you can use the function plot_points. This is explained in the second example here.

Within each of the above probability plotting functions you may enter failure data as well as right censored data. For those distributions that have a function in `reliability.Fitters` for fitting location shifted distributions (Weibull_3P, Gamma_3P, Lognormal_3P, Exponential_2P), you can explicitly tell the probability plotting function to fit the gamma parameter using fit_gamma=True. By default the gamma parameter is not fitted. Fitting the gamma parameter will
also change the x-axis to time-gamma such that everything will appear linear. An example of this is shown in the second example below.

**Note:** Beta and Gamma probability plots have their y-axes scaled based on the distribution’s parameters so you will find that when you overlay two Gamma or two Beta distributions on the same Gamma or Beta probability paper, one will be a curved line if they have different shape parameters. This is unavoidable due to the nature of Gamma and Beta probability paper and is the reason why you will never find a hardcopy of such paper and also the reason why these distributions are not used in ALT probability plotting.

**Note:** Confidence intervals will not appear on Gamma or Beta probability plots. These confidence intervals are in development.

**Inputs:**
- failures - the array or list of failure times
- right_censored - the array or list of right censored failure times
- fit_gamma - this is only included for Weibull, Gamma, Lognormal, and Exponential probability plots. Specify fit_gamma=True to fit the location shifted distribution.
- a - this is the heuristic for plotting positions of the form (k-a)/(n+1-2a). Default is a=0.3 which is the median rank method (same as in Minitab). You can specify any heuristic that follows this form. For more heuristics, see wikipedia.
- show_fitted_distribution - True/False. If True, the fitted distribution will be plotted on the probability plot. Defaults to True. If you want a probability plot with just the data points and no line for the distribution then set this to False.
- plotting keywords are also accepted where relevant and they are mostly applied to the fitted distribution line. The exception to this is for color which defaults to blue line and black points but if specified the chosen color will be applied to both line and points. This is useful when overlaying multiple datasets on a single probability plot.

**Outputs:**
- The plot is the only output. Use plt.show() to show it.

In the example below we generate some samples from a Normal Distribution and provide these to the probability plotting function. It is also possible to overlay other plots of the CDF as is shown by the dashed line.

```python
from reliability.Distributions import Normal_Distribution
from reliability.Probability_plotting import Normal_probability_plot
import matplotlib.pyplot as plt

dist = Normal_Distribution(mu=50, sigma=10)
failures = dist.random_samples(100, seed=5)
Normal_probability_plot(failures=failures)  #generates the probability plot
dist.CDF(linestyle='--', label='True CDF')  #this is the actual distribution provided for comparison
plt.legend()
plt.show()
```
In this second example, we will fit an Exponential distribution to some right censored data. To create this data, we will draw it from an Exponential distribution that has a location shift of 12. Once again, the true CDF has also been plotted to provide the comparison. Note that the x-axis is time-gamma as it is necessary to subtract gamma from the x-plotting positions if we want the plot to appear linear.

```python
from reliability.Distributions import Exponential_Distribution
from reliability.Probability_plotting import Exponential_probability_plot
import matplotlib.pyplot as plt
from reliability.Other_functions import make_right_censored_data

dist = Exponential_Distribution(Lambda=0.25, gamma=12)
raw_data = dist.random_samples(100, seed=42)  # draw some random data from an exponential distribution
data = make_right_censored_data(raw_data, threshold=17)  # right censor the data at 17
```
In this third example, we will see how probability plotting can be used to highlight the importance of getting as much data as possible. This code performs a loop in which increasing numbers of samples are used for fitting a Weibull distribution and the accuracy of the results (shown both in the legend and by comparison with the True CDF) increases with the number of samples.
In this fourth example, we will take a look at the special case of the Exponential probability plot using the Weibull Scale. This plot is essentially a Weibull probability plot, but the fitting and plotting functions are Exponential. The reason for plotting an Exponential distribution on Weibull probability paper is to achieve parallel lines for different Lambda parameters rather than having the lines radiating from the origin as we see in the Exponential probability plot on Exponential probability paper. This has applications in ALT probability plotting and is the default plot provided from Fit_Exponential_1P and Fit_Exponential_2P. An example of the differences between the plots are shown below. Remember that the alpha parameter from the Weibull distribution is equivalent to 1/Lambda from the Exponential distribution and a Weibull distribution with Beta = 1 is the same as an Exponential distribution.
In this final example, we take a look at how a probability plot can show us that there’s something wrong with our assumption of a single distribution. To generate the data, the random samples are drawn from two different distributions which are shown in the left image. In the right image, the scatterplot of failure times is clearly non-linear. The green line is the attempt to fit a single Weibull_2P distribution and this will do a poor job of modelling the data. Also note that the points of the scatterplot do not fall on the True CDF of each distribution. This is because the median rank method of obtaining the plotting positions does not work well if the failure times come from more than one distribution. If you see a pattern like this, try a mixture model or a competing risks model. Always remember that cusps, corners, and doglegs indicate a mixture of failure modes.
10.1 What does a probability plot show me?

A probability plot shows how well your data is modelled by a particular distribution. By scaling the axes in such a way that the fitted distribution’s CDF appears to be a straight line, we can judge whether the empirical CDF of the failure data (the black dots) are in agreement with the CDF of the fitted distribution. Ideally we would see that all of the black dots would lie on the straight line but most of the time this is not the case. A bad fit is evident when the line or curve formed by the black dots is deviating significantly from the straight line. We can usually tolerate a little bit of deviation at the tails of the distribution but the majority of the black dots should follow the line. A historically popular test was the ‘fat pencil test’ which suggested that if a fat pencil could cover the majority of the data points then the fit was probably suitable. Such a method makes no mention of the size of the plot window which could easily affect the result so it is best to use your own judgement and experience. This approach is not a substitute for statistical inference so it is often preferred to use quantitative measures for goodness of fit such as AICc and BIC. Despite being an imprecise measure, probability plots remain popular among reliability engineers and in reliability engineering software.

from reliability.Probability_plotting import Weibull_probability_plot, Exponential_probability_plot
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt

data = Weibull_Distribution(alpha=5,beta=3).random_samples(100, seed=1)
plt.subplot(121)
Weibull_probability_plot(failures=data)
plt.title('Example of a good fit')
plt.subplot(122)
Exponential_probability_plot(failures=data)
plt.title('Example of a bad fit')
plt.subplots_adjust(bottom=0.1, right=0.94, top=0.93, wspace=0.34)  # adjust the formatting
plt.show()
10.1. What does a probability plot show me?
This section contains two different styles of quantile-quantile plots. These are the fully parametric quantile-quantile plot (reliability.Probability_plotting.QQ_plot_parametric) and the semi-parametric quantile-quantile plot (reliability.Probability_plotting.QQ_plot_semiparametric). These will be described separately below. A quantile-quantile (QQ) plot is made by plotting failure units vs failure units for shared quantiles. A quantile is the value of random variable (time, cycles, landings, etc.) which corresponds to a given fraction failing (that ranges from 0 to 1).

11.1 Parametric Quantile-Quantile plot

To generate this plot we calculate the failure units (these may be units of time, strength, cycles, landings, rounds fired, etc.) at which a certain fraction has failed (0.01, 0.02, 0.03, ..., 0.99). We do this for each distribution so we have an array of failure units and then we plot these failure units against eachother. The time (or any other failure unit) at which a given fraction has failed is found using the inverse survival function. If the distributions are similar in shape, then the QQ plot should be a reasonably straight line (but not necessarily a 45 degree line). By plotting the failure times at equal quantiles for each distribution we can obtain a conversion between the two distributions. Such conversions are useful for accelerated life testing (ALT) to easily convert field time to test time.

Inputs:

- X_dist - a probability distribution. The failure times at given quantiles from this distribution will be plotted along the X-axis.
- Y_dist - a probability distribution. The failure times at given quantiles from this distribution will be plotted along the Y-axis.
- show_fitted_lines - True/False. Default is True. These are the Y=mX and Y=mX+c lines of best fit.
- show_diagonal_line - True/False. Default is False. If True the diagonal line will be shown on the plot.

Outputs:

- The QQ_plot will always be output. Use plt.show() to show it.
- [m,m1,c1] - these are the values for the lines of best fit. m is used in Y=mX, and m1 and c1 are used in Y=m1X+c1
In the example below, we have determined that the field failures follow a Weibull distribution (\( \alpha=350, \beta=2.01 \)) with time represented in months. By using an accelerated life test we have replicated the failure mode and Weibull shape parameter reasonably closely and the Lab failures follow a Weibull distribution (\( \alpha=128, \beta=2.11 \)) with time measured in hours. We would like to obtain a simple Field-to-Lab conversion for time so we know how much lab time is required to simulate 10 years of field time. The QQ plot will automatically provide the equations for the lines of best fit. If we use the \( Y=mX \) equation we see that \( \text{Field(months)}=2.757 \times \text{Lab(hours)} \). Therefore, to simulate 10 years of field time (120 months) we need to run the accelerated life test for approximately 43.5 hours in the Lab.

```python
from reliability.Probability_plotting import QQ_plot_parametric
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt

Field = Weibull_Distribution(alpha=350, beta=2.01)
Lab = Weibull_Distribution(alpha=128, beta=2.11)

QQ_plot_parametric(X_dist=Lab, Y_dist=Field)
plt.show()
```

### 11.2 Semiparametric Quantile-Quantile plot

This plot is still a Quantile-Quantile plot (plotting failure units vs failure units for shared quantiles), but instead of using two parametric distributions, we use the failure data directly as one set of quantiles. We then estimate what the quantiles of the parametric distribution would be and plot the parametric (theoretical) failure units against the actual failure units. To generate this plot we begin with the failure units (these may be units of time, strength, cycles, landings, etc.). We then obtain an empirical CDF using either Kaplan-Meier, Nelson-Aalen, or Rank Adjustment. The empirical CDF is then plotted against the theoretical CDF for the parametric distribution. The resulting plot allows for a visual assessment of how well the parametric distribution fits the failure data.
CDF gives us the quantiles we will use to equate the actual and theoretical failure times. Once we have the empirical CDF, we use the inverse survival function of the specified distribution to obtain the theoretical failure units and then plot the actual and theoretical failure units together. The primary purpose of this plot is as a graphical goodness of fit test. If the specified distribution is a good fit to the data then the QQ plot should be a reasonably straight line along the diagonal.

Inputs:
- **X_data_failures** - the failure times in an array or list. These will be plotted along the X-axis.
- **X_data_right_censored** - the right censored failure times in an array or list. Optional input.
- **Y_dist** - a probability distribution. The quantiles of this distribution will be plotted along the Y-axis.
- **show_fitted_lines** - True/False. Default is True. These are the Y=mX and Y=mX+c lines of best fit.
- **show_diagonal_line** - True/False. Default is False. If True the diagonal line will be shown on the plot.

Outputs:
- The QQ_plot will always be output. Use plt.show() to show it.
- **[m,m1,c1]** - these are the values for the lines of best fit. m is used in Y=mX, and m1 and c1 are used in Y=m1X+c1

In the example below, we generate 100 random samples from a Normal distribution. We then fit a Weibull_2P distribution to this data and using QQ_plot_semiparametric we compare the actual quantile (the original data) with the theoretical quantiles (from the fitted distribution). The lines of best fit are automatically provided and the Y=0.992X shows the relationship is very close to perfect with only some deviation around the tails of the distribution. The final example on this page compares a QQ_plot_semiparametric with a PP_plot_semiparametric for the same dataset to show the differences between the two.

```python
from reliability.Probability_plotting import QQ_plot_semiparametric
from reliability.Fitters import Fit_Weibull_2P
from reliability.Distributions import Normal_Distribution
import matplotlib.pyplot as plt
data = Normal_Distribution(mu=50, sigma=12).random_samples(100)
fitted_dist = Fit_Weibull_2P(failures=data, print_results=False, show_probability_plot=False).distribution
QQ_plot_semiparametric(X_data_failures=data, Y_dist=fitted_dist)
plt.show()
```
11.3 Comparing PP plots with QQ plots

In this example we compare a QQ_plot_parametric with a PP_plot_parametric for the same pair of distributions. Normally, it is not practical to compare the output of the two plots as they are so vastly different and are used for different purposes, but the comparison is provided for the reader’s understanding. The differences between these plots are so significant because one is the time at which the fraction has failed (the Quantile) and the other is the fraction failing (the CDF). Parametric PP plots are not very common as their only use is in providing a graphical understanding of the differences between the CDFs of two distributions, such as how one lags or leads the other at various times. See Probability-Probability plots for more detail on the uses of parametric PP plots.

```python
from reliability.Probability_plotting import QQ_plot_parametric, PP_plot_parametric
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt
Field = Weibull_Distribution(alpha=350, beta=2.01)
Lab = Weibull_Distribution(alpha=128, beta=2.11)
plt.figure(figsize=(10,5))
plt.subplot(121)
QQ_plot_parametric(X_dist=Lab, Y_dist=Field, show_diagonal_line=True, show_fitted_lines=False)
plt.subplot(122)
PP_plot_parametric(X_dist=Lab, Y_dist=Field, show_diagonal_line=True)
plt.show()
```
In this example we compare a QQ_plot_semiparametric with a PP_plot_semiparametric for the same dataset. Both plots are intended to be used as graphical goodness of fit tests. In a PP plot we get a lot of resolution in the center of the distributions, but less at the tails, whereas the QQ plot gives very good resolution at the tails, but less in the center. Because most data analysts are more concerned about the extremes (tails) of a distribution, QQ plots are the more commonly used plot between the two.

```python
from reliability.Probability_plotting import PP_plot_semiparametric, QQ_plot_semiparametric
from reliability.Fitters import Fit_Normal_2P
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt

data = Weibull_Distribution(alpha=100,beta=3).random_samples(100)  #create some data
dist = Fit_Normal_2P(failures=data,print_results=False,show_probability_plot=False).distribution  #fit a normal distribution
plt.figure(figsize=(10,5))
plt.subplot(121)
QQ_plot_semiparametric(X_data_failures=data,Y_dist=dist,show_fitted_lines=False,show_diagonal_line=True)
plt.subplot(122)
PP_plot_semiparametric(X_data_failures=data,Y_dist=dist)
plt.show()
```

11.3. Comparing PP plots with QQ plots

83
Chapter 11. Quantile-Quantile plots
This section contains two different styles of probability-probability plots. These are the fully parametric probability-probability plot (reliability.Probability_plotting.PP_plot_parametric) and the semi-parametric probability-probability plot (reliability.Probability_plotting.PP_plot_semiparametric). These will be described separately below. A probability-probability (PP) plot is made by plotting the fraction failing (CDF) of one distribution vs the fraction failing (CDF) of another distribution. In the semiparametric form, when we only have the failure data and one hypothesised distribution, the CDF for the data can be obtained non-parametrically to generate an Empirical CDF.

12.1 Parametric Probability-Probability plot

To generate this plot we simply plot the CDF of one distribution vs the CDF of another distribution. If the distributions are very similar, the points will lie on the 45 degree diagonal. Any deviation from this diagonal indicates that one distribution is leading or lagging the other. Fully parametric PP plots are rarely used as their utility is limited to providing a graphical comparison of the similarity between two CDFs. To aide this comparison, the PP_plot_parametric function accepts x and y quantile lines that will be traced across to the other distribution.

Inputs:
• X_dist - a probability distribution. The CDF of this distribution will be plotted along the X-axis.
• Y_dist - a probability distribution. The CDF of this distribution will be plotted along the Y-axis.
• y_quantile_lines - starting points for the trace lines to find the X equivalent of the Y-quantile. Optional input. Must be list or array.
• x_quantile_lines - starting points for the trace lines to find the Y equivalent of the X-quantile. Optional input. Must be list or array.
• show_diagonal_line - True/False. Default is False. If True the diagonal line will be shown on the plot.

Outputs:
• The PP_plot is the only output. Use plt.show() to show it.
In the example below, we generate two parametric distributions and compare them using a PP plot. We are interested in the differences at specific quantiles so these are specified and the plot traces them across to the opposing distribution.

```python
from reliability.Probability_plotting import PP_plot_parametric
from reliability.Distributions import Weibull_Distribution, Normal_Distribution
import matplotlib.pyplot as plt

Field = Normal_Distribution(mu=100, sigma=30)
Lab = Weibull_Distribution(alpha=120, beta=3)

PP_plot_parametric(X_dist=Field, Y_dist=Lab, x_quantile_lines=[0.3, 0.6], y_quantile_lines=[0.1, 0.6])
plt.show()
```

### 12.2 Semiparametric Probability-Probability plot

A semiparametric PP plot is still a probability-probability plot, but since we only have one parametric distribution to give us the CDF, we must use the failure data to obtain the non-parametric estimate of the empirical CDF. To create a semiparametric PP plot, we must provide the failure data and the non-parametric method (‘KM’, ‘NA’, ‘RA’ for Kaplan-Meier, Nelson-Aalen, and Rank Adjustment respectively) to estimate the empirical CDF, and we must also provide the parametric distribution for the parametric CDF. The failure units (times, cycles, rounds fired, strength units, etc.) are the limiting values here so the parametric CDF is only calculated at the failure units since that is the result we get from the empirical CDF. Note that the empirical CDF also accepts X_data_right_censored just as KaplanMeier, NelsonAalen and RankAdjustment will also accept right censored data. If the fitted distribution is a good fit the PP plot will follow the 45 degree diagonal line. Assessing goodness of fit in a graphical way is the main purpose of this.
type of plot. The `Fit_everything` function also uses a semiparametric PP plot to show the goodness of fit in a graphical way.

Inputs:

- `X_data_failures` - the failure times in an array or list. The empirical CDF of this data will be plotted along the X-axis.
- `X_data_right_censored` - the right censored failure times in an array or list. This is an optional input.
- `Y_dist` - a probability distribution. The CDF of this distribution will be plotted along the Y-axis.
- `show_diagonal_line` - True/False. Default is True. If True the diagonal line will be shown on the plot.

Outputs:

- The PP_plot is the only output. Use `plt.show()` to show it.

In the example below, we create 100 random samples from a Weibull distribution. We hypothesise that a Normal distribution may fit this data well so we fit the Normal distribution and then plot the CDF of the fitted distribution against the empirical CDF (obtained using the Kaplan-Meier estimate). We see that the plot follows the 45 degree diagonal quite well so we may consider that the fitted Normal distribution is reasonably good at describing this data. Ideally, this comparison should be made against other distributions as well and the graphical results are often hard to tell apart which is why we often use quantitative goodness of fit measures like AICc and BIC.

```python
def from reliability.Probability_plotting import PP_plot_semiparametric
def from reliability.Fitters import Fit_Normal_2P
def from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt
data = Weibull_Distribution(alpha=5,beta=3).random_samples(100)
dist = Fit_Normal_2P(failures=data,show_probability_plot=False,print_results=False).distribution
PP_plot_semiparametric(X_data_failures=data,Y_dist=dist)
plt.show()
```
To see how semiparametric PP plots compare with semiparametric QQ plots as a graphical goodness of fit test, please see the second example in the section on comparing PP plots with QQ plots.
The Kaplan-Meier estimator provides a method by which to estimate the survival function (reliability function) of a population without assuming that the data comes from a particular distribution. Due to the lack of parameters required in this model, it is a non-parametric method of obtaining the survival function. With a few simple transformations, the survival function (SF) can be used to obtain the cumulative hazard function (CHF) and the cumulative distribution function (CDF). It is not possible to obtain a useful version of the probability density function (PDF) or hazard function (HF) as this would require the differentiation of the CDF and CHF respectively, which results in a very spikey plot due to the non-continuous nature of these plots.

The Kaplan-Meier estimator is very similar in result (but quite different in method) to the Nelson-Aalen estimator and Rank Adjustment estimator. While none of the three has been proven to be more accurate than the others, the Kaplan-Meier estimator is generally more popular as a non-parametric means of estimating the SF. Confidence intervals are provided using the Greenwood method with Normal approximation.

The Kaplan-Meier estimator can be used with both complete and right censored data. This function can be accessed from `reliability.Nonparametric.KaplanMeier`.

Inputs:

- failures - an array or list of failure times.
- right_censored - an array or list of right censored failure times. Defaults to None.
- show_plot - True/False. Default is True. Plots the SF.
- print_results - True/False. Default is True. Will display a pandas dataframe of results in the console.
- plot_CI - shades the upper and lower confidence interval
- CI - confidence interval between 0 and 1. Default is 0.95 for 95% CI.
- plot_type - SF, CDF, or CHF. Default is SF.

Outputs:

- results - dataframe of results
- KM - list of Kaplan-Meier column from results dataframe. This column is the non parametric estimate of the Survival Function (reliability function).
• xvals - the x-values to plot the stepwise plot as seen when show_plot=True
• SF - survival function stepwise values (these differ from the KM values as there are extra values added in to make the plot into a step plot)
• CDF - cumulative distribution function stepwise values
• CHF - cumulative hazard function stepwise values
• SF_lower - survival function stepwise values for lower CI
• SF_upper - survival function stepwise values for upper CI
• CDF_lower - cumulative distribution function stepwise values for lower CI
• CDF_upper - cumulative distribution function stepwise values for upper CI
• CHF_lower - cumulative hazard function stepwise values for lower CI
• CHF_upper - cumulative hazard function stepwise values for upper CI

Other plotting keywords (such as color, label, linestyle, etc.) are accepted and used on the point estimate line. The color of the confidence intervals is matched automatically to the point estimate line, but no other keywords are carried across to the confidence interval plot as it is only a shaded region.

In this first example, we will provide Kaplan-Meier with a list of failure times and right censored times. By leaving everything else unspecified, the plot will be shown with the confidence intervals shaded. We will layer this first Kaplan-Meier plot with a second one using just the failure data. As can be seen in the example below, the importance of including censored data is paramount to obtain an accurate estimate of the reliability, because without it the population’s survivors are not included so the reliability will appear much lower than it truly is.

```python
from reliability.Nonparametric import KaplanMeier
import matplotlib.pyplot as plt

f = [5248, 7454, 16890, 17200, 38700, 45000, 49390, 69040, 72280, 131900]
rc = [3961, 4007, 4734, 6054, 7298, 10190, 23060, 27160, 28690, 37100, 40060, 45670, 53000, 67000, 69630, 77350, 78470, 91680, 105700, 106300, 150400]
KaplanMeier(failures=f, right_censored=rc, label='Failures + right censored')
KaplanMeier(failures=f, label='Failures only')
plt.title('Kaplan-Meier estimates showing the importance of including censored data')
plt.xlabel('Miles to failure')
plt.legend()
plt.show()
```

(continues on next page)
<table>
<thead>
<tr>
<th>Failure times</th>
<th>Lower CI bound</th>
<th>Upper CI bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>5248.0</td>
<td>0.714061</td>
<td>1.000000</td>
</tr>
<tr>
<td>7454.0</td>
<td>0.415974</td>
<td>0.845217</td>
</tr>
<tr>
<td>105700</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>106300</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>131900</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>150400</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>69630</td>
<td>0.396904</td>
<td>0.836731</td>
</tr>
<tr>
<td>69040</td>
<td>0.396904</td>
<td>0.836731</td>
</tr>
<tr>
<td>67000</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>49390</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>53000</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>72280</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>77350</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>84700</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>40060</td>
<td>0.633417</td>
<td>0.957580</td>
</tr>
<tr>
<td>45500</td>
<td>0.560893</td>
<td>0.924037</td>
</tr>
<tr>
<td>131900</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>5248.0</td>
<td>0.714061</td>
<td>1.000000</td>
</tr>
<tr>
<td>7454.0</td>
<td>0.552082</td>
<td>1.000000</td>
</tr>
<tr>
<td>16890.0</td>
<td>0.415974</td>
<td>0.984026</td>
</tr>
<tr>
<td>10190</td>
<td>0.826513</td>
<td>1.000000</td>
</tr>
<tr>
<td>16890</td>
<td>0.763170</td>
<td>1.000000</td>
</tr>
<tr>
<td>23060</td>
<td>0.705334</td>
<td>0.985101</td>
</tr>
<tr>
<td>27160</td>
<td>0.705334</td>
<td>0.985101</td>
</tr>
<tr>
<td>28690</td>
<td>0.705334</td>
<td>0.985101</td>
</tr>
<tr>
<td>37100</td>
<td>0.705334</td>
<td>0.985101</td>
</tr>
<tr>
<td>38700</td>
<td>0.633417</td>
<td>0.957580</td>
</tr>
<tr>
<td>40060</td>
<td>0.633417</td>
<td>0.957580</td>
</tr>
<tr>
<td>45500</td>
<td>0.560893</td>
<td>0.924037</td>
</tr>
<tr>
<td>45670</td>
<td>0.560893</td>
<td>0.924037</td>
</tr>
<tr>
<td>49390</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>53000</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>67000</td>
<td>0.486210</td>
<td>0.884496</td>
</tr>
<tr>
<td>69630</td>
<td>0.396904</td>
<td>0.836731</td>
</tr>
<tr>
<td>69040</td>
<td>0.396904</td>
<td>0.836731</td>
</tr>
<tr>
<td>72280</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>77350</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>84700</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>91680</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>105700</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>106300</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>131900</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
<tr>
<td>150400</td>
<td>0.300949</td>
<td>0.778481</td>
</tr>
</tbody>
</table>
In this second example, we will create some data from a Weibull distribution, and then right censor the data above our chosen threshold. We will then fit a Weibull_2P distribution to the censored data, and also obtain the Kaplan-Meier estimate of this data. Using the results from the Fit_Weibull_2P and the Kaplan-Meier estimate, we will plot the CDF, SF, and CHF, for both the Weibull and Kaplan-Meier results. Note that the default plot from KaplanMeier will only give you the SF, but the results object provides everything you need to reconstruct the SF plot yourself, as well as what we need to plot the CDF and CHF.
from reliability.Distributions import Weibull_Distribution
from reliability.Fitters import Fit_Weibull_2P
from reliability.Nonparametric import KaplanMeier
from reliability.Other_functions import make_right_censored_data
import matplotlib.pyplot as plt

dist = Weibull_Distribution(alpha=5, beta=2)  # create a distribution
raw_data = dist.random_samples(100, seed=2)    # get some data from the distribution.
    # Seeded for repeatability
data = make_right_censored_data(raw_data, threshold=9)
wbf = Fit_Weibull_2P(failures=data.failures, right_censored=data.right_censored, show_probability_plot=False, print_results=False)  # Fit the Weibull_2P

# Create the subplots and in each subplot we will plot the parametric distribution and obtain the Kaplan Meier fit.
# Note that the plot_type is being changed each time
plt.figure(figsize=(12, 5))
plt.subplot(131)
KaplanMeier(failures=data.failures, right_censored=data.right_censored, plot_type='SF', print_results=False, label='Kaplan-Meier')
wbf.distribution.SF(label='Parametric')
plt.legend()
plt.title('SF')
plt.subplot(132)
KaplanMeier(failures=data.failures, right_censored=data.right_censored, plot_type='CDF', print_results=False, label='Kaplan-Meier')
wbf.distribution.CDF(label='Parametric')
plt.legend()
plt.title('CDF')
plt.subplot(133)
KaplanMeier(failures=data.failures, right_censored=data.right_censored, plot_type='CHF', print_results=False, label='Kaplan-Meier')
wbf.distribution.CHF(label='Parametric')
plt.legend()
plt.title('CHF')
plt.subplots_adjust(left=0.07, right=0.95, top=0.92, wspace=0.25)  # format the plot layout
plt.show()
The Nelson-Aalen estimator provides a method by which to estimate the hazard function of a population without assuming that the data comes from a particular distribution. From the hazard function, the Nelson-Aalen method obtains the cumulative hazard function, which is then used to obtain the survival function. Due to the lack of parameters required in this model, it is a non-parametric method of obtaining the survival function. As with the Kaplan-Meier estimator, once we have the survival function (SF), then we also have the cumulative hazard function (CHF) and the cumulative distribution function (CDF). It is not possible to obtain a useful version of the probability density function (PDF) or hazard function (HF). While the hazard function is obtained directly by the Nelson-Aalen method, it is a useless function on its own as it is a very spikey plot due to the non-continuous nature of the hazard. It is only when we smooth the results out using the cumulative hazard function that we obtain some utility from the results.

The Nelson-Aalen estimator is very similar in result (but quite different in method) to the Kaplan-Meier estimator and Rank Adjustment estimator. While none of the three have been proven to be more accurate than the others, the Kaplan-Meier estimator is generally more popular as a non-parametric means of estimating the SF. Confidence intervals are provided using the Greenwood method with Normal approximation.

The Nelson-Aalen estimator can be used with both complete and right censored data. This function can be accessed from reliability.Nonparametric.NelsonAalen.

**Inputs:**
- failures - an array or list of failure times.
- right_censored - an array or list of right censored failure times. Defaults to None.
- show_plot - True/False. Default is True. Plots the SF.
- print_results - True/False. Default is True. Will display a pandas dataframe of results in the console.
- plot_CI - shades the upper and lower confidence interval
- CI - confidence interval between 0 and 1. Default is 0.95 for 95% CI.
- plot_type - SF, CDF, or CHF. Default is SF.

**Outputs:**
- results - dataframe of results
- NA - list of Nelson-Aalen column from results dataframe. This column is the non parametric estimate of the Survival Function (reliability function).
- xvals - the x-values to plot the stepwise plot as seen when show_plot=True
- SF - survival function stepwise values (these differ from the NA values as there are extra values added in to make the plot into a step plot)
- CDF - cumulative distribution function stepwise values
- CHF - cumulative hazard function stepwise values
- SF_lower - survival function stepwise values for lower CI
- SF_upper - survival function stepwise values for upper CI
- CDF_lower - cumulative distribution function stepwise values for lower CI
- CDF_upper - cumulative distribution function stepwise values for upper CI
- CHF_lower - cumulative hazard function stepwise values for lower CI
- CHF_upper - cumulative hazard function stepwise values for upper CI

Other plotting keywords (such as color, label, linestyle, etc.) are accepted and used on the point estimate line. The color of the confidence intervals is matched automatically to the point estimate line, but no other keywords are carried across to the confidence interval plot as it is only a shaded region.

In the example below, we will compare the results from the Nelson-Aalen estimator with the results from the Kaplan-Meier estimator and Rank Adjustment estimator. We will also extract the column of point estimates from the results and print these for each method in a dataframe.

```python
from reliability.Nonparametric import KaplanMeier, NelsonAalen, RankAdjustment
import matplotlib.pyplot as plt
import pandas as pd

failures = [5248, 7454, 16890, 17200, 38700, 45000, 49390, 69040, 72280, 131900]
censored = [3961, 4007, 4734, 6054, 7298, 10190, 23060, 27160, 28690, 37100, 40060, 45670, 53000, 67000, 69630, 78470, 91680, 105700, 106300, 150400]
KM = KaplanMeier(failures=failures, right_censored=censored, label='Kaplan-Meier', print_results=False)
NA = NelsonAalen(failures=failures, right_censored=censored, label='Nelson-Aalen', print_results=False)
RA = RankAdjustment(failures=failures, right_censored=censored, label='Rank Adjustment', print_results=False)

plt.title('Comparison of Kaplan-Meier, Nelson-Aalen, and Rank Adjustment with 95% CI bounds')
plt.legend()

# print a table of the SF estimates for each method
data = {'Kaplan-Meier': KM.KM, 'Nelson-Aalen': NA.NA, 'Rank Adjustment': RA.RA}
df = pd.DataFrame(data, columns=['Kaplan-Meier', 'Nelson-Aalen', 'Rank Adjustment'])
print(df)

plt.show()
```

Kaplan-Meier Nelson-Aalen Rank Adjustment
0 1.000000 1.000000 1.000000
1 1.000000 1.000000 1.000000
2 1.000000 1.000000 1.000000

(continues on next page)
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.964286</td>
<td>0.964916</td>
<td>0.974412</td>
</tr>
<tr>
<td>2</td>
<td>0.964286</td>
<td>0.964916</td>
<td>0.974412</td>
</tr>
<tr>
<td>3</td>
<td>0.964286</td>
<td>0.964916</td>
<td>0.974412</td>
</tr>
<tr>
<td>4</td>
<td>0.925714</td>
<td>0.927081</td>
<td>0.936568</td>
</tr>
<tr>
<td>5</td>
<td>0.925714</td>
<td>0.927081</td>
<td>0.936568</td>
</tr>
<tr>
<td>6</td>
<td>0.885466</td>
<td>0.887637</td>
<td>0.897146</td>
</tr>
<tr>
<td>7</td>
<td>0.845217</td>
<td>0.848193</td>
<td>0.857724</td>
</tr>
<tr>
<td>8</td>
<td>0.845217</td>
<td>0.848193</td>
<td>0.857724</td>
</tr>
<tr>
<td>9</td>
<td>0.845217</td>
<td>0.848193</td>
<td>0.857724</td>
</tr>
<tr>
<td>10</td>
<td>0.845217</td>
<td>0.848193</td>
<td>0.857724</td>
</tr>
<tr>
<td>11</td>
<td>0.795499</td>
<td>0.799738</td>
<td>0.809542</td>
</tr>
<tr>
<td>12</td>
<td>0.795499</td>
<td>0.799738</td>
<td>0.809542</td>
</tr>
<tr>
<td>13</td>
<td>0.742465</td>
<td>0.748161</td>
<td>0.758348</td>
</tr>
<tr>
<td>14</td>
<td>0.742465</td>
<td>0.748161</td>
<td>0.758348</td>
</tr>
<tr>
<td>15</td>
<td>0.685353</td>
<td>0.692768</td>
<td>0.703498</td>
</tr>
<tr>
<td>16</td>
<td>0.685353</td>
<td>0.692768</td>
<td>0.703498</td>
</tr>
<tr>
<td>17</td>
<td>0.685353</td>
<td>0.692768</td>
<td>0.703498</td>
</tr>
<tr>
<td>18</td>
<td>0.616817</td>
<td>0.626842</td>
<td>0.638675</td>
</tr>
<tr>
<td>19</td>
<td>0.616817</td>
<td>0.626842</td>
<td>0.638675</td>
</tr>
<tr>
<td>20</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>21</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>22</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>23</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>24</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>25</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>26</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>27</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>28</td>
<td>0.539715</td>
<td>0.553186</td>
<td>0.566650</td>
</tr>
<tr>
<td>29</td>
<td>0.269858</td>
<td>0.335524</td>
<td>0.374582</td>
</tr>
<tr>
<td>30</td>
<td>0.269858</td>
<td>0.335524</td>
<td>0.374582</td>
</tr>
</tbody>
</table>
Further examples are provided in the documentation for the **Kaplan-Meier estimator** as this function is written to work exactly the same way as the Nelson-Aalen estimator.
The Rank Adjustment estimator provides a method by which to estimate the survival function (reliability function) of a population without assuming that the data comes from a particular distribution. Due to the lack of parameters required in this model, it is a non-parametric method of obtaining the survival function. With a few simple transformations, the survival function (SF) can be used to obtain the cumulative hazard function (CHF) and the cumulative distribution function (CDF). It is not possible to obtain a useful version of the probability density function (PDF) or hazard function (HF) as this would require the differentiation of the CDF and CHF respectively, which results in a very spikey plot due to the non-continuous nature of these plots.

The Rank Adjustment estimator is very similar in result (but quite different in method) to the Kaplan-Meier estimator and Nelson-Aalen estimator. While none of the three has been proven to be more accurate than the others, the Kaplan-Meier estimator is generally more popular as a non-parametric means of estimating the SF. Confidence intervals are provided using the Greenwood method with Normal approximation.

The Rank Adjustment estimator can be used with both complete and right censored data. This function can be accessed from `reliability.Nonparametric.RankAdjustment`. The Rank-adjustment algorithm is the same as is used in `Probability_plotting.plotting_positions` to obtain y-values for the scatter plot. As with `plotting_positions`, the heuristic constant “a” is accepted, with the default being 0.3 for median ranks.

Inputs:

- failures - an array or list of failure times. Sorting is automatic so times do not need to be provided in any order.
- right_censored - an array or list of right censored failure times. Defaults to None.
- show_plot - True/False. Default is True. Plots the CDF, SF, or CHF as specified by plot_type.
- plot_type - SF, CDF, or CHF. Default is SF.
- print_results - True/False. Default is True. Will display a pandas dataframe in the console.
- plot_CI - shades the upper and lower confidence interval
- CI - confidence interval between 0 and 1. Default is 0.95 for 95% CI.
- a - the heuristic constant for plotting positions of the form (k-a)/(n+1-2a). Default is a=0.3 which is the median rank method (same as the default in Minitab). Must be in the range 0 to 1. For more heuristics, see Wikipedia.

Outputs:
• results - dataframe of results for the SF
• RA - list of rank-adjustment column from results dataframe. This column is the non parametric estimate of the Survival Function (reliability function).
• xvals - the x-values to plot the stepwise plot as seen when show_plot=True
• SF - survival function stepwise values (these differ from the RA values as there are extra values added in to make the plot into a step plot)
• CDF - cumulative distribution function stepwise values
• CHF - cumulative hazard function stepwise values
• SF_lower - survival function stepwise values for lower CI
• SF_upper - survival function stepwise values for upper CI
• CDF_lower - cumulative distribution function stepwise values for lower CI
• CDF_upper - cumulative distribution function stepwise values for upper CI
• CHF_lower - cumulative hazard function stepwise values for lower CI
• CHF_upper - cumulative hazard function stepwise values for upper CI

Other plotting keywords (such as color, label, linestyle, etc.) are accepted and used on the point estimate line. The color of the confidence intervals is matched automatically to the point estimate line, but no other keywords are carried across to the confidence interval plot as it is only a shaded region.

In this first example, we will see how Rank Adjustment compares with Kaplan-Meier and Nelson-Aalen for a large censored dataset. The plots show these three methods arrive at a similar result, with Kaplan-Meier giving the lowest estimate of the survival function, followed by Nelson-Aalen, and finally Rank-Adjustment. Note that this is when the median ranks are used in the rank adjustment heuristic. As sample size is increased, the differences between the three methods reduces.

```python
import matplotlib.pyplot as plt
from reliability.Other_functions import make_right_censored_data
from reliability.Nonparametric import KaplanMeier, NelsonAalen, RankAdjustment
from reliability.Distributions import Weibull_Distribution

dist = Weibull_Distribution(alpha=500, beta=2)
plt.figure(figsize=(12, 7))
samples = [10, 100, 1000]
for i, s in enumerate(samples):
    raw_data = dist.random_samples(number_of_samples=s, seed=42)
data = make_right_censored_data(data=raw_data, fraction_censored=0.5, seed=42) # this will multiply-censor 50% of the data
    plt.subplot(131 + i)
    plt.scatter(xvals, RA, color='red', label='Rank Adjustment')
    plt.title(str(str(s) + ' samples'))
plt.legend()
plt.suptitle('Comparison of Kaplan-Meier, Nelson-Aalen, and Rank Adjustment for varying sample sizes with 50% censoring')
```

(continues on next page)
In this second example we will look at the effect of the plotting heuristic “a”. The default heuristic used is 0.3 which gives the median ranks, but there are many other heuristics available by varying a from 0 to 1. Here we will look at the effect of setting “a” to be 0, 0.3, and 1. The effect is fairly minimal, though there is a difference which leads to the use of different heuristics. The median ranks (a=0.3) is most popular and is the default in Minitab for obtaining plotting positions.

```python
from reliability.Nonparametric import RankAdjustment
import matplotlib.pyplot as plt
f = [5248, 7454, 16890, 17200, 38700, 45000, 49390, 69040, 72280, 131900]
rc = [3961, 4007, 4734, 6054, 7298, 10190, 23060, 27160, 28690, 37100, 40060, 45670, 53000, 67000, 69630, 77350, 78470, 91680, 105700, 106300, 150400]
a_trials = [0, 0.3, 1]
for a in a_trials:
    RankAdjustment(failures=f, right_censored=rc, print_results=False, a=a,
                    label=str(a))
plt.legend(title="Heuristic 'a'")
plt.title('Effect of rank adjustment heuristic')
plt.show()
```
Chapter 15. Rank Adjustment
Stress-Strength interference is a model to predict the probability of failure when the probability distributions of the stress and the strength are known. Failure is defined as when stress > strength. The model calculates the probability of failure through integration.

If both the stress and strength distributions are normal distributions, then there exists a simple analytical solution which will give an exact result. For this method, use the function `Probability_of_failure_normdist`. For two Normal distributions, the difference between the integration method and the analytical method is very small (around 1e-11).

**Inputs:**
- `stress` - a probability distribution from the Distributions module
- `strength` - a probability distribution from the Distributions module
- `show_distribution_plot` - True/False (default is True)
- `print_results` - True/False (default is True)
- `warn` - a warning will be issued if both stress and strength are Normal as you should use `Probability_of_failure_normdist`. You can supress this using `warn=False`

**Outputs:**
- the probability of failure
- the distribution plot (only shown if `show_distribution_plot=True`)
- results printed to console (only shown if `print_results=True`)

In this example, we will create a stress and strength distribution, and leaving everything else as default, we will see the results printed and the distribution plot.

```python
from reliability import Distributions
from reliability.Stress_strength import Probability_of_failure
import matplotlib.pyplot as plt

stress = Distributions.Weibull_Distribution(alpha=2, beta=3, gamma=1)
strength = Distributions.Gamma_Distribution(alpha=2, beta=3, gamma=3)
```

(continues on next page)
reliability Documentation, Release latest

Probability of failure: 0.0017078235941570912

Note: In Version 0.5.0 the calculation method was changed from monte-carlo to integration. This resulted in an API change. See the Changelog for details.

Easter Egg: The logo for reliability was created using the stress strength interference plot of two Weibull distributions.
RELIABILITY
A Python library for reliability engineering
Stress-Strength interference is a model to predict the probability of failure when the probability distributions of the stress and the strength are known. The model calculates the probability of failure by determining the probability that a random stress (drawn from a stress distribution) is greater than a random strength (drawn from a strength distribution). If you are working with stress and strength distributions that are not both Normal distributions, then you will need to use the integration method provided in Probability_of_failure. However, if both the stress and strength distributions are Normal distributions then the exact solution can be found using an analytical approach provided here in Probability_of_failure_normdist.

Inputs:
- stress - a Normal probability distribution from the Distributions module
- strength - a Normal probability distribution from the Distributions module
- show_distribution_plot - True/False (default is True)
- print_results - True/False (default is True)

Outputs:
- the probability of failure
- the distribution plot (only shown if show_distribution_plot=True)
- results printed to console (only shown if print_results=True)

In this example, we will create a stress and strength distribution (both of which are Normal distributions), and leaving everything else as default, we will see the results plotted and printed. Unlike the integration method this approach is analytical, though a comparison of the two methods reveals the error in the integration method is negligible (around 1e-11).

```python
code{from reliability import Distributions
from reliability.Stress_strength import Probability_of_failure_normdist
import matplotlib.pyplot as plt

stress = Distributions.Normal_Distribution(mu=20, sigma=6)
strength = Distributions.Normal_Distribution(mu=40, sigma=7)
```
Probability of failure: 0.015029783946206214
Reliability growth occurs involves gradual product improvement through the elimination of design deficiencies. In repairable systems, reliability growth is observable through an increase in the interarrival times of failures. Reliability growth is applicable to all levels of design decomposition from complete systems down to components.

The Duane method of modeling reliability growth involves the use of the total time on test \([t]\) (we may also use distance, cycles, etc.) when the failure occurred and the sequence of the failure \([N]\). The cumulative mean time between failures (MTBF) is \([t/N]\). By plotting \(\ln(t) vs \ln(t/N)\) we obtain a straight line which is used get the parameters lambda and beta. Using these parameters, we can model the instantaneous MTBF in the form \((t^{(1-beta)}/(\lambda*\beta))\). The function `reliability_growth` accepts the failure times and performs this model fitting to obtain the parameters lambda and beta, as well as produce the reliability growth plot. It is often of interest to know how much total time on test we need to meet a target MTBF. This can be found analytically and is included in this function.

**Inputs:**
- `times` - array of list of failure times
- `xmax` - xlim to plot up to. Default is 1.5*max(times)
- `target_MTBF` - specify the target MTBF to obtain the total time on test required to reach it.
- `show_plot` - True/False. Defaults to True. Other keyword arguments (such as color, title, etc) are also accepted and used.
- `print_results` - True/False. Defaults to True.

**Outputs:**
- `Lambda` - the lambda parameter from the Duane model
- `Beta` - the beta parameter from the Duane model
- `time_to_target` - The time (from the start of the test) until the target MTBF is reached. time_to_target is only returned if target_MTBF is specified.
- If `show_plot` is True, it will plot the reliability growth. Use `plt.show()` to show the plot.
- If `print_results` is True, it will print a summary of the fitted parameters and time to target MTBF (if target is specified).
In the example below, we supply the total time on test when each failure occurred, and we also supply the target_MTBF as 50000 so that we can find out how much total time on test will be needed to reach the target MTBF.

```python
from reliability.Repairable_systems import reliability_growth
import matplotlib.pyplot as plt

times = [10400, 26900, 43400, 66400, 89400, 130400, 163400, 232000, 242000, 340700]
reliability_growth(times=times, target_MTBF=50000, label='Reliability growth curve', xmax=500000)
plt.legend()
plt.show()

Reliability growth model parameters:
lambda: 0.002355878294089656
beta: 0.6638280053477188
Time to reach target MTBF: 428131.18039448344
```

![Reliability Growth](image)
When conducting maintenance planning, we must optimise the frequency of preventative maintenance (PM) for the minimum overall cost. If PM is conducted too frequently then we will have high costs, but if not conducted often enough then failures will result and we incur the higher cost of corrective maintenance (CM). Depending on the underlying failure distribution, it is possible to model these costs for a range of PM intervals, with the lowest cost per unit time resulting from the optimal replacement time. This function calculates the cost per unit time to determine how cost varies with replacement time. The cost model can be used for HPP (ie. the maintenance makes the system “as good as new”) or Power Law NHPP (ie. the maintenance makes the system “as good as old”). The default is for “as good as new”. Currently, this model is only implemented to use the Weibull distribution as the underlying failure distribution.

Inputs:

- cost_PM - cost of preventative maintenance (must be smaller than cost_CM)
- cost_CM - cost of corrective maintenance (must be larger than cost_PM)
- weibull_alpha - scale parameter of the underlying Weibull distribution
- weibull_beta - shape parameter of the underlying Weibull distribution. Should be greater than 1 otherwise conducting PM is not economical.
- q - restoration factor. q=1 is Power Law NHPP (as good as old), q=0 is HPP (as good as new). Default is q=0 (as good as new).
- show_plot - True/False. Defaults to True. Other plotting keywords are also accepted and used.
- print_results - True/False. Defaults to True

Outputs:

- ORT - the optimal replacement time
- min_cost - the minimum cost per unit time
- Plot of cost model if show_plot is set to True. Use plt.show() to display it.
- Printed results if print_results is set to True.
In the example below, we provide the cost of preventative maintenance (cost_PM), and the cost of corrective maintenance (cost_CM), as well as the weibull parameters of the failure distribution. Leaving the default outputs, we obtain a plot of the cost per unit time and the printed results. This example is based on the reliaSoft article.

```python
from reliability.Repairable_systems import optimal_replacement_time
import matplotlib.pyplot as plt

optimal_replacement_time(cost_PM=1, cost_CM=5, weibull_alpha=1000, weibull_beta=2.5, q=0)
plt.show()

Cost model assuming as good as new replacement (q=0):
The minimum cost per unit time is 0.0035
The optimal replacement time is 493.19

Minimum cost per unit time is 0.0035
Optimal replacement time is 493.19
```
ROCOF is the Rate of Occurrence of Failures. It is used to model the trend (constant, increasing, decreasing) in the failure interarrival times. For a repairable system, we want the ROCOF to be improving (failure interarrival times to be increasing). As failure times can often appear quite random, it is necessary to conduct a statistical test to determine if there is a statistically significant trend, and if there is a trend we can then model that trend using a Power Law NHPP. The test for statistical significance is the Laplace test which compares the Laplace test statistic (U) with the z value (z_crit) from the standard normal distribution. If there is a statistically significant trend, the parameters of the model (Lambda_hat and Beta_hat) are calculated. By default the results are printed and a plot of the failure interarrival times and MTBF is plotted.

Inputs:
- times_between_failures - these are the failure interarrival times.
- failure_times - these are the actual failure times. Note 1: You can specify either times_between_failures OR failure_times but not both. Both options are provided for convenience so the conversion between the two is done internally. failure_times should be the same as np.cumsum(times_between_failures). Note 2: The repair time is assumed to be negligible. If the repair times are not negligibly small then you will need to manually adjust your input to factor in the repair times.
- test_end - use this to specify the end of the test if the test did not end at the time of the last failure.
- CI - the confidence interval for the Laplace test. Default is 0.95 for 95% CI.
- show_plot - True/False. Default is True. Plotting keywords are also accepted (eg. color, linestyle).
- print_results - True/False. Default is True

Outputs:
- U - The Laplace test statistic
- z_crit - (lower,upper) bound on z value. This is based on the CI.
- trend - ‘improving’,’worsening’,’constant’. This is based on the comparison of U with z_crit
- Beta_hat - the Beta parameter for the NHPP Power Law model. Only calculated if the trend is not constant.
- Lambda_hat - the Lambda parameter for the NHPP Power Law model. Only calculated if the trend is not constant.
- ROCOF - the Rate of Occurrence Of Failures. Only calculated if the trend is constant. If trend is not constant then ROCOF changes over time in accordance with Beta_hat and Lambda_hat.

- printed results. Only printed if print_results is True.

- plotted results. Only plotted if plot_results is True. Use plt.show() to display it.

In the example below, we provide the failure interarrival times. The function will run the Laplace test using the default 95% confidence interval and then, when a trend is found, it will plot the MTBF based on the calculated NHPP Power Law model. MTBF = 1/ROCOF.

```python
from reliability.Repairable_systems import ROCOF
import matplotlib.pyplot as plt

t = [104, 131, 1597, 59, 4, 503, 157, 6, 118, 173, 114, 62, 101, 216, 106, 140, 1, 102, 3, 393, 96, 232, 89, 61, 37, 293, 7, 165, 87, 99]
ROCOF(times_between_failures=t)
plt.show()

Laplace test results: U = 2.409, z_crit = (-1.96,+1.96)
At 95 % confidence level the ROCOF is WORSENING. Assume NHPP.
ROCOF assuming NHPP has parameters: Beta_hat = 1.588, Lambda_hat = 3.703e-05
```

![Failure interarrival times vs failure number](image)

At 95% confidence level the ROCOF is worsening
RELIABILITY
A Python library for reliability engineering
The Mean Cumulative Function (MCF) is a cumulative history function that shows the cumulative number of recurrences of an event, such as repairs over time. In the context of repairs over time, the value of the MCF can be thought of as the average number of repairs that each system will have undergone after a certain time. It is only applicable to repairable systems and assumes that each event (repair) is identical. For the non-parametric MCF it does not assume that each system’s MCF is identical, but this assumption is made for the parametric MCF.

The shape of the MCF is a key indicator that shows whether the systems are improving, worsening, or staying the same over time. If the MCF is concave down (appearing to level out) then the system is improving. A straight line (constant increase) indicates it is staying the same. Concave up (getting steeper) shows the system is worsening as repairs are required more frequently as time progresses.

Obtaining the MCF from failure times is an inherently non-parametric process (similar to Kaplan-Meier), but once the values are obtained, a model can be fitted to obtain the parametric estimate of the MCF. Each of these two approaches is described below as they are performed by separate functions within reliability.Repairable_systems.

Note that in some textbooks and academic papers the Mean Cumulative Function is also referred to as the Cumulative Intensity Function (CIF). These are two names for the same thing. If the shape of your MCF is more of an S than a single smooth curve, you may have a change in operating condition or in the repair effectiveness factor. This can be dealt with by splitting the MCF into segments, however, such models are more complex and are generally only found in academic literature.

### 21.1 Non-parametric MCF

The non-parametric estimate of the MCF provides both the estimate of the MCF and the confidence bounds at a particular time. The procedure to obtain the non-parametric MCF is outlined here. The confidence bounds are the one-sided bounds as this was chosen to align with the method used by Reliasoft.

**Inputs:**

- data - the repair times for each system. Format this as a list of lists. eg. data=[[4,7,9],[3,8,12]] would be the data for 2 systems. The largest time for each system is assumed to be the retirement time and is treated as a right censored value. If the system was retired immediately after the last repair then you must include a repeated value at the end as this will be used to indicate a right censored value. eg. A system that had repairs at 4, 7, and
9 then was retired after the last repair would be entered as data = [4,7,9,9] since the last value is treated as a right censored value. If you only have data from 1 system you may enter the data in a single list as data = [3,7,12] and it will be nested within another list automatically.

- print_results - prints the table of MCF results (state, time, MCF_lower, MCF, MCF_upper, variance)
- CI - Confidence interval. Default is 0.95 for 95% CI (one sided).
- show_plot - if True the plot will be shown. Default is True. Use plt.show() to show it.
- plot_CI - the plot will include the confidence intervals. Default is True.

Outputs:

- If print_results is True, a table of the results will be printed showing state, time, MCF_lower, MCF, MCF_upper, variance. In this table state is F for failure or C for right censored (retirement).
- If show_plot is True, the MCF plot will be shown.
- results - this is a dataframe of the results that are printed. It includes the blank lines for censored values
- time - this is the time column from results. Blank lines for censored values are removed
- MCF - this is the MCF column from results. Blank lines for censored values are removed
- variance - this is the Variance column from results. Blank lines for censored values are removed
- lower - this is the MCF_lower column from results. Blank lines for censored values are removed
- upper - this is the MCF_upper column from results. Blank lines for censored values are removed

The following example is taken from an example provided by Reliasoft. The failure times and retirement times (retirement time is indicated by +) of 5 systems are:

<table>
<thead>
<tr>
<th>Equipment ID</th>
<th>Months</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5, 10, 15, 17+</td>
</tr>
<tr>
<td>2</td>
<td>6, 13, 17, 19+</td>
</tr>
<tr>
<td>3</td>
<td>12, 20, 25, 26+</td>
</tr>
<tr>
<td>4</td>
<td>13, 15, 24+</td>
</tr>
<tr>
<td>5</td>
<td>16, 22, 25, 28+</td>
</tr>
</tbody>
</table>

```python
from reliability.Repairable_systems import MCF_nonparametric
import matplotlib.pyplot as plt
times = [[5, 10, 15, 17], [6, 13, 17, 19], [12, 20, 25, 26], [13, 15, 24], [16, 22, 25, 28]]
MCF_nonparametric(data=times)
plt.show()
```

```
Mean Cumulative Function results (95.0% CI)

<table>
<thead>
<tr>
<th>state</th>
<th>time</th>
<th>MCF_lower</th>
<th>MCF</th>
<th>MCF_upper</th>
<th>variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>5</td>
<td>0.0459299</td>
<td>0.2</td>
<td>0.870893</td>
<td>0.032</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>0.14134</td>
<td>0.4</td>
<td>1.13202</td>
<td>0.064</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>0.256603</td>
<td>0.6</td>
<td>1.40294</td>
<td>0.096</td>
</tr>
<tr>
<td>F</td>
<td>12</td>
<td>0.383374</td>
<td>0.8</td>
<td>1.66939</td>
<td>0.128</td>
</tr>
<tr>
<td>F</td>
<td>13</td>
<td>0.517916</td>
<td>1</td>
<td>1.93081</td>
<td>0.16</td>
</tr>
<tr>
<td>F</td>
<td>13</td>
<td>0.658169</td>
<td>1.2</td>
<td>2.18789</td>
<td>0.192</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>0.802848</td>
<td>1.4</td>
<td>2.44131</td>
<td>0.224</td>
</tr>
</tbody>
</table>
```

(continues on next page)
21.2 Parametric MCF

The estimates of the parametric MCF are obtained using MCF_nonparametric as this is the procedure required to obtain the points for the plot. To these points a Non-Homogeneous Poisson Process (NHPP) parametric model is fitted of the form:

\[ MCF(t) = \left( \frac{t}{\alpha} \right)^\beta \]

You may notice that this looks identical to the Weibull CHF, but despite this similarity, they are entirely different.
functions and the alpha and beta parameters from the MCF cannot be applied to a Weibull distribution for fitting the repair times or repair interarrival times.

The purpose of fitting a parametric model is to obtain the shape parameter ($\beta$) which indicates the long term health of the system/s. If the MCF is concave down ($\beta<1$) then the system is improving. A straight line ($\beta=1$) indicates it is staying the same. Concave up ($\beta>1$) shows the system is worsening as repairs are required more frequently as time progresses.

Many methods exist for fitting the model to the data. Within reliability, scipy.optimize.curve_fit is used which returns the covariance matrix and allows for the confidence intervals to be calculated using the appropriate formulas.

Inputs:

- data - the repair times for each system. Format this as a list of lists. e.g., data=[[4, 7, 9, 12], [3, 8, 12]] would be the data for 2 systems. The largest time for each system is assumed to be the retirement time and is treated as a right censored value. If the system was retired immediately after the last repair then you must include a repeated value at the end as this will be used to indicate a right censored value. e.g. A system that had repairs at 4, 7, and 9 then was retired after the last repair would be entered as data = [4, 7, 9, 9] since the last value is treated as a right censored value. If you only have data from 1 system you may enter the data in a single list as data = [3, 7, 12] and it will be nested within another list automatically.
- CI - the confidence interval. Default is 0.95 for 95% CI.
- print_results - prints the fitted parameters (alpha and beta) of the parametric MCF model.
- show_plot - if True the plot will be shown. Default is True. Use plt.show() to show it.
- plot_CI - True/False. Plots the confidence intervals. Default is True.

Outputs:

- If print_results is True, the model parameters will be printed along with a brief diagnosis of the long term health of the system based on the beta parameter.
- times - this is the times (x values) from the scatter plot. This value is calculated using MCF_nonparametric.
- MCF - this is the MCF (y values) from the scatter plot. This value is calculated using MCF_nonparametric.
- alpha - the calculated alpha parameter
- beta - the calculated beta parameter
- alpha_SE - the standard error in the alpha parameter
- beta_SE - the standard error in the beta parameter
- cov_alpha_beta - the covariance between the parameters
- alpha_upper - the upper CI estimate of the parameter
- alpha_lower - the lower CI estimate of the parameter
- beta_upper - the upper CI estimate of the parameter
- beta_lower - the lower CI estimate of the parameter
- results - a dataframe of the results (point estimate, standard error, Lower CI and Upper CI for each parameter)

The following example uses the same data as the MCF_nonparametric example provided above. From the output we can clearly see that the system is degrading over time as repairs are needed more frequently.

```python
from reliability.Repairable_systems import MCF_parametric
import matplotlib.pyplot as plt

times = [[5, 10, 15, 17], [6, 13, 17, 19], [12, 20, 25, 26], [13, 15, 24], [16, 22, 25, 28]]
```
reliability Documentation, Release latest

```python
from reliability.Repairable_systems import MCF_parametric
from reliability.Datasets import MCF_2
import matplotlib.pyplot as plt

times = MCF_2().times
MCF_parametric(data=times)
plt.show()

Mean Cumulative Function Parametric Model (95% CI):
$MCF = \left(\frac{t}{\alpha}\right)^\beta$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>11.980590</td>
<td>0.401372</td>
<td>11.219187</td>
<td>12.793666</td>
</tr>
<tr>
<td>Beta</td>
<td>1.673622</td>
<td>0.094654</td>
<td>1.498017</td>
<td>1.869813</td>
</tr>
</tbody>
</table>

Since Beta is greater than 1, the system repair rate is WORSENING over time.
```

![Parametric estimate of the Mean Cumulative Function](image.png)

$MCF = \left(\frac{t}{\alpha}\right)^\beta$ with $\alpha=11.9806$, $\beta=1.6736$

The parametric model that is fitted to the MCF is not always an appropriate model. The example below shows data from a collection of systems, some of which are improving and some are worsening. The net effect is an S-shaped MCF. The power model used by MCF_parametric is not able to accurately follow an S-shaped dataset. In this case, the MCF_nonparametric model is more appropriate, though there are some other parametric models (discussed in the first paragraph) which may be useful to model this dataset.
MCF_parametric(data=times, print_results=False)
plt.show()

Parametric estimate of the Mean Cumulative Function

\[ MCF = \left(\frac{t}{\alpha}\right)^\beta \] with \( \alpha=1533.5951 \), \( \beta=1.8627 \)

- **RELIABILITY**
- A Python library for reliability engineering
This function will plot the stress vs number of cycles (S-N) diagram when supplied with data from a series of fatigue tests. An S-N diagram is a common procedure used to model the fatigue life of metals which are subject to known cyclic loads. Typically, the plot is done using a semilog scale where the number of cycles is scaled logarithmically. This has the effect of linearizing the plot and making the accuracy of the model much easier to visualize. For steels, titanium alloys, and some other metals, there exists an endurance limit. This limit is the minimum stress required to propagate fatigue cracks, and all stresses below this endurance limit do not contribute to fatigue growth. The plot can be adjusted to use an endurance limit using the optional inputs, however, there must be runout data (equivalent to right censored data) supplied in order for the program to determine where to set the endurance limit.

Inputs:

- stress - an array or list of stress values at failure
- cycles - an array or list of cycles values at failure
- stress_runout - an array or list of stress values that did not result in failure Optional
- cycles_runout - an array or list of cycles values that did not result in failure. Optional
- xscale - ‘log’ or ‘linear’. Default is ‘log’. Adjusts the x-scale.
- stress_trace - an array or list of stress values to be traced across to cycles values.
- cycles_trace - an array or list of cycles values to be traced across to stress values.
- show_endurance_limit - This will adjust all lines of best fit to be greater than or equal to the average stress_runout. Defaults to False if stress_runout is not specified. Defaults to True if stress_runout is specified.
- method_for_bounds - ‘statistical’, ‘residual’, or None. Defaults to ‘statistical’. If set to ‘statistical’ the CI value is used, otherwise it is not used for the ‘residual’ method. Residual uses the maximum residual datapoint for symmetric bounds. Setting the method for bounds to None will turn off the confidence bounds.
- CI - Must be between 0 and 1. Default is 0.95 for 95% confidence interval. Only used if method_for_bounds = ‘statistical’
- Other plotting keywords (eg. color, linestyle, etc) are accepted for the line of best fit.

Outputs:
The plot is the only output. All calculated values are shown on the plot.

In this first example, we use the data for stress and cycles to produce an S-N diagram. We will not provide any runout data here so the endurance limit will not be calculated.

```python
from reliability.PoF import SN_diagram
import matplotlib.pyplot as plt
cycles = [15000, 24000, 36000, 80000, 177000, 162000, 301000, 290000, 361000, 881000, 1300000, 2500000]
SN_diagram(stress=stress, cycles=cycles)
plt.show()
```

In this second example, we will use the same data as above, but also supply runout data so that the endurance limit will be calculated. We will also adjust the method_for_bounds to be 'residual'. We are also going to find the life (in cycles) at a stress of 260 by using stress_trace, and the stress required to achieve a life of 5x10^5 cycles using cycles_trace.

```python
from reliability.PoF import SN_diagram
import matplotlib.pyplot as plt
cycles = [15000, 24000, 36000, 80000, 177000, 162000, 301000, 290000, 361000, 881000, 1300000, 2500000]
stress_runout = [210, 210, 205, 205, 205]
cycles_runout = [10 ** 7, 10 ** 7, 10 ** 7, 10 ** 7, 10 ** 7]
SN_diagram(stress=stress, cycles=cycles, stress_runout=stress_runout, cycles_runout=cycles_runout, method_for_bounds='residual', cycles_trace=[5 * 10 ** 5], stress_trace=[260])
```

(continues on next page)
plt.show()

References:
In the strain-life method of fatigue analysis, the elastic and plastic deformation of the material is used to determine how many cycles the material will last before failure. In this context, failure is defined as the formation of a small crack (typically 1mm) so the geometry of the material does not need to be considered provided the material properties have been accurately measured using a stress-strain test. This section of the documentation describes three functions which are useful in strain-life analysis. The first of these is useful to fit the stress-strain and strain-life models to available data, thereby providing the material properties. The second function is a diagram of the relationship between stress and strain during cyclic fatigue which shows the hysteresis loop and finds the min and max stress and strain. The third function produces the strain-life diagram and the equations for this diagram are used for calculating the number of cycles to failure. Further detail is available below for each of the respective functions.

The equations used for stress-strain and strain life are:

Ramberg-Osgood equation:

$$\varepsilon_{tot} = \varepsilon_{\text{elastic}} + (\frac{\sigma}{K})^{\frac{1}{n}}$$

Hysteresis curve equation:

$$\Delta \varepsilon = \Delta \varepsilon_{\text{elastic}} + 2 \left(\frac{\Delta \sigma}{2K}\right)^\frac{1}{n}$$

Coffin-Manson equation:

$$\varepsilon_{tot} = \varepsilon_{f} (2N_f)^b + \varepsilon_f (2N_f)^c$$

Morrow Mean Stress Correction:

$$\varepsilon_{tot} = \varepsilon_{f} (2N_f)^b + \varepsilon_f (2N_f)^c$$

Modified Morrow Mean Stress Correction:

$$\varepsilon_{tot} = \varepsilon_{f} (2N_f)^b + \varepsilon_f \left(\frac{\sigma_f - \sigma_m}{\sigma_f}\right) (2N_f)^c$$
Smith-Watson-Topper Mean Stress Correction: \[ \varepsilon_{\text{tot}} = \frac{\sigma_f^2}{\sigma_{\text{max}} E} (2N_f)^2 + \frac{\sigma_f \varepsilon_f}{\sigma_{\text{max}}} (2N_f)^{b+c} \]

### 23.1 Stress-Strain and Strain-Life parameter estimation

The function `stress_strain_life_parameters_from_data` will use stress and strain data to calculate the stress-strain parameters (K, n) from the Ramberg-Osgood relationship. If cycles is provided it will also produce the strain-life parameters (sigma_f, epsilon_f, b, c) from the Coffin-Manson equation. You cannot find the strain-life parameters without stress as we must use stress to find elastic strain. Note that if you already have the parameters K, n, sigma_f, epsilon_f, b, c, then you can use the functions `stress_strain_diagram` or `strain_life_diagram` as described below.

**Inputs:**
- strain - an array or list of strain
- stress - an array or list of stress
- E - The modulus of elasticity. Ensure this is in the same units as stress (typically MPa)
- cycles - the number of cycles to failure. Optional input. Required if you want to obtain the parameters sigma_f, epsilon_f, b, c
- print_results - True/False. Default is True.
- show_plot - True/False. Default is True.

**Outputs:**
- The stress-strain plot will be generated if show_plot is True. If cycles is provided then the strain-life plot will also be generated. Use `plt.show()` to show any plots.
- The results will be printed in the console if print_results is True.
- K - the cyclic strength coefficient
- n - the cyclic strain hardening exponent
- sigma_f - the fatigue strength coefficient. Only generated if cycles is provided.
- epsilon_f - the fatigue strain coefficient. Only generated if cycles is provided.
- b - the elastic strain exponent. Only generated if cycles is provided.
- c - the plastic strain exponent. Only generated if cycles is provided.

Note that the parameters generated are stored to an object, so you may find it useful to use this function with print_results=False and show_plot=False, and then access the calculated parameters later. This is done in the first example for the section on stress-strain diagram.

In the example below, we provide data from a fatigue test including stress, strain, and cycles to failure. We must also provide the modulus of elasticity (E) for the material. All other options are left as default values. The plots shown below are provided and the results are printed to the console.

```python
from reliability.PoF import stress_strain_life_parameters_from_data
import matplotlib.pyplot as plt
strain_data = [0.02, 0.015, 0.01, 0.006, 0.0035, 0.002]
stress_data = [650, 625, 555, 480, 395, 330]
cycles_data = [200, 350, 1100, 4600, 26000, 560000]
params = stress_strain_life_parameters_from_data(stress=stress_data, strain=strain_data, cycles=cycles_data, E=216000)
```

(continues on next page)
plt.show()

K (cyclic strength coefficient): 1462.4649152172044
n (strain hardening exponent): 0.19810419512368083
sigma_f (fatigue strength coefficient): 1097.405402055844
epsilon_f (fatigue strain coefficient): 0.23664541556833998
b (elastic strain exponent): -0.08598339316495743
\( \sigma \) (plastic strain exponent): -0.4501077996416115

\[
\epsilon_{\text{tot}} = \frac{\sigma}{216000} + \left( \frac{\sigma}{1462.4649} \right)^{0.1981}
\]

Stress-Strain diagram

23.1. Stress-Strain and Strain-Life parameter estimation
23.2 Stress-Strain diagram

The function `stress_strain_diagram` is used to visualize how the stress and strain vary with successive load cycles as described by the hysteresis curve equation. Due to residual tensile and compressive stresses, the stress and strain in the material does not unload in the same way that it loads. This results in a hysteresis loop being formed and this is the basis for crack propagation in the material leading to fatigue failure. The size of the hysteresis loop increases for higher strains. Fatigue tests are typically strain controlled; that is they are subjected to a specified amount of strain throughout the test, typically in a sinusoidal pattern. Fatigue tests may also be stress controlled, whereby the material is subjected to a specified amount of stress. This function accepts either input (max_stress or max_strain) and will find the corresponding stress or strain as required. If you do not specify min_stress or min_strain then it is assumed to be negative of the maximum value.

The cyclic loading sequence defaults to begin with tension, but can be changed using `initial_load_direction='compression'`. If your test begins with compression it is important to specify this as the residual stresses in the material from the initial loading will affect the results for the first reversal. This difference is caused by the Bauschinger effect. Only the initial loading and the first two reversals are plotted. For most materials the shape of the hysteresis loop will change over many hundreds of cycles as a result of fatigue hardening (also known as work-hardening) or fatigue-softening. More on this process is available in the eFatigue training documents.

Note that if you do not have the parameters K, n, but you do have stress and strain data then you can use the function `stress_strain_life_parameters_from_data`. This will be shown in the first example below.

Inputs:
• K - cyclic strength coefficient
• n - strain hardening exponent
• E - The modulus of elasticity. Ensure this is in the same units for which K and n were obtained (typically MPa)
• max_strain - the maximum strain to use for cyclic loading when plotting the hysteresis loop.
• max_stress - the maximum stress to use for cyclic loading when plotting the hysteresis loop. When specifying min and max stress or strain, do not specify both stress and strain as the corresponding value will be automatically calculated.
• min_strain - if this is not -max_strain then specify it here. Optional input.
• min_stress - if this is not -max_stress then specify it here. Optional input.
• initial_load_direction - ‘tension’ or ‘compression’. Default is ‘tension’.

Outputs:
• The stress-strain plot will always be generated. Use plt.show() to show it.
• If print_results is True, the calculated parameters below will be printed.
  • max_stress
  • max_strain
  • min_stress
  • min_strain

In the example below, we are using the same data from the first example, but this time, we will store the calculated parameters in an object named ‘params’. Then we can specify the calculated parameters to the stress_strain_diagram function. The hysteresis loop generated is for a strain-controlled fatigue test where the strain goes from -0.006 to +0.006.

```python
from reliability.PoF import stress_strain_life_parameters_from_data, stress_strain_diagram
import matplotlib.pyplot as plt
strain_data = [0.02, 0.015, 0.01, 0.006, 0.0035, 0.002]
stress_data = [650, 625, 555, 480, 395, 330]
cycles_data = [200, 350, 1100, 4600, 26000, 560000]
params = stress_strain_life_parameters_from_data(stress=stress_data, strain=strain_data, cycles=cycles_data, E=216000, show_plot=False, print_results=False)
stress_strain_diagram(E = 216000, n = params.n, K = params.K, max_strain=0.006)
plt.show()

Max stress: 483.85816239406745
Min stress: -483.8581623940621
Max strain: 0.006
Min strain: -0.006
```
In this second example, we will use the `stress_strain_diagram` to visualise the effects of residual stresses for a material subjected to non-zero mean stress. The material parameters (K and n) are already known so we do not need to obtain them from any data. We specify the max_stress is 378 MPa and the min_stress is -321 MPa. We will do this for two scenarios: initial tensile load, and initial compressive load. Upon inspection of the results we see for the initial tensile load, the min_stress in the material is actually -328.893 MPa which exceeds the min_stress we specified in our test. When we have an initial compressive load, the max_stress is 385.893 MPa which exceeds the max_stress we specified in our test. These results are not an error and are caused by the residual stresses in the material that were formed during the first loading cycle. In the case of an initial tensile load, when the material was pulled apart in tension by an external force, the material pulls back but due to plastic deformation, these internal forces in the material are not entirely removed, such that when the first compressive load peaks, the material’s internal stresses add to the external compressive forces. This phenomenon is important in load sequence effects for variable amplitude fatigue.

```python
from reliability.PoF import stress_strain_diagram
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(121)
print('Tension first:')
stress_strain_diagram(E=210000, K = 1200, n = 0.2, max_stress=378,min_stress=-321,
                       initial_load_direction='tension')
plt.title('Cyclic loading - tension first')
plt.subplot(122)
print('Compression first:')
stress_strain_diagram(E=210000, K = 1200, n = 0.2, max_stress=378,min_stress=-321,
                       initial_load_direction='compression')
plt.title('Cyclic loading - compression first')
```

(continues on next page)
23.3 Strain-Life diagram

The function `strain_life_diagram` provides a visual representation of the Coffin-Manson relationship between strain and life. In this equation, strain is split into elastic strain and plastic strain which are shown on the plot as straight lines (on a log-log scale), and life is represented by reversals (with 2 reversals per cycle). The total strain amplitude is used to determine the fatigue life by solving the Coffin-Manson equation. When a min_stress or min_strain is specified that results in a non-zero mean stress, there are several mean stress correction methods that are available. These are ‘morrow’, ‘modified_morrow’ (also known as Manson-Halford), and ‘SWT’ (Smith-Watson-Topper). The default method is ‘SWT’ but can be changed using the options described below. The equation used is displayed in the legend of the plot. Also shown on the plot is the life of the material at the specified strain amplitude, and the transition life (2Nt) for which the material failure transitions from being dominated by plastic strain to elastic strain.

Note that if you do not have the parameters sigma_f, epsilon_f, b, c, but you do have stress, strain, and cycles data then you can use the function ‘stress_strain_life_parameters_from_data’.
The residual stress in a material subjected to non-zero mean stress (as shown in the previous example) are not considered in this analysis, and the specified max and min values for stress or strain are taken as the true values to which the material is subjected.

Inputs:

- E - The modulus of elasticity. Ensure this is in the same units for which K and n were obtained (typically MPa)
- sigma_f - fatigue strength coefficient
- epsilon_f - fatigue strain coefficient
- b - elastic strain exponent
- c - plastic strain exponent
- K - cyclic strength coefficient. Optional input. Only required if you specify max_stress or max_strain.
- n - strain hardening exponent. Optional input. Only required if you specify max_stress or max_strain.
- max_stress - specify the max_stress if you want cycles to failure. If specified, you will also need to specify K and n.
- max_strain - specify the max_strain if you want cycles to failure.
- min_stress - if this is not -max_stress then specify it here. Optional input.
- min_strain - if this is not -max_strain then specify it here. Optional input. When specifying min and max stress or strain, do not specify both stress and strain as the corresponding value will be automatically calculated. Only specify the min if it is not -max
- mean_stress_correction_method - must be either ‘morrow’, ‘modified_morrow’, or ‘SWT’. Default is ‘SWT’. This is only used if mean_stress is found to be non-zero.
- print_results - True/False. Defaults to True. If use_level_stress or use_level_strain is specified then the printed results will be the cycles_to_failure
- show_plot - True/False. Default is True

Outputs:

- The strain-life plot will be generated if show_plot = True. Use plt.show() to show it.
- cycles_to_failure - only calculated if max_stress OR max_strain is specified. This will be printed if print_results = True.

```python
from reliability.PoF import strain_life_diagram
import matplotlib.pyplot as plt
strain_life_diagram(E=210000, sigma_f=1000, epsilon_f=1.1, b=-0.1, c=-0.6, K=1200, n=0.2, max_strain=0.0049, min_strain=-0.0029)
plt.show()
```

Failure will occur in 13771.39 cycles (27542.78 reversals).
References:

Fracture mechanics is an approach to fatigue analysis that involves calculating the number of cycles until failure of a component that is undergoing cyclic loading. We are generally interested in two values; the number of cycles needed to initiate a crack, and the number of cycles to grow the crack to a certain size (usually the size for brittle fracture to occur). The fracture mechanics functions described below are useful for solving typical fatigue problems given to students. Unfortunately, the limitation of these functions is that they are only applicable to thin plates with through-thickness cracks. As soon as you encounter a component that is not a thin plate, then the formulas required for analysis will be different from those used below. This is part of what makes fracture mechanics such a complex topic that is better handled by purpose-built fatigue analysis software such as Altair Hyperlife. Even in the relatively simple thin flat plate geometry, there are many complications that make fracture mechanics a challenging subject. These include variable amplitude loading, surface roughness, frequency effect, environmental effects (temperature, corrosion) and other miscellaneous factors. Solving fracture mechanics problems for flat plates can still provide engineers with an appreciation for how fatigue operates and the factors affecting fatigue so that they can incorporate these lessons learned into their work.

Most textbooks (including Probabilistic Physics of Failure Approach to Reliability (2017) which was used to design the functions below) apply a few simplifications for solving crack growth problems. These simplifications involve an assumption that the stress in the component is constant, that the geometry factor is constant, and that the crack length to cause failure (which has the geometry factor in its formula) is constant. These simplifications are necessary for hand calculations, but in reality we know that they all must change as the crack length grows which necessitates an iterative calculation. Both the simplified and iterative methods are included in the crack growth function. Also included in both functions is the ability to solve these problems for notched components by providing the appropriate correction factors for the notched geometry.

### 24.1 Crack initiation

The function `fracture_mechanics_crack_initiation` uses the material properties, the local cross-sectional area, and force applied to the component to determine how many cycles until crack initiation (of a 1 mm crack). Units should always be in MPa (and mm^2 for area). This function may be used for an un-notched or notched component. If the component is un-notched, the parameters q and Kt may be left as their default values of 1.

While there are formulas to find the parameters q and Kt, these formulas have not been included here so that the
function is reasonably generic to different materials and geometries. Resources for finding some of these parameters if they are not given to you:

\[ q = \frac{1}{1 + a/r} \]

Where \( r \) is the notch radius of curvature (in mm), and \( a \) is \( 0.025 \times (2070/Su) \). \( Su \) is the ultimate strength in MPa. This only applies to high strength steels where \( Su > 550 \text{MPa} \).

\( K_t \) can be found from the eFatigue website which has an online calculator that will provide you with the appropriate \( K_t \) for your notched geometry.

Inputs:

- \( P \) - Force applied on the component [units of MPa]
- \( A \) - Cross sectional area of the component (at the point of crack initiation) [units of mm\(^2\)]
- \( Sy \) - Yield strength of the material [units of MPa]
- \( E \) - Elastic modulus (Young’s modulus) [units of MPa]
- \( K \) - Strength coefficient of the material
- \( n \) - Strain hardening exponent of the material
- \( b \) - Elastic strain exponent of the material
- \( c \) - Plastic strain exponent of the material
- \( \sigma_f \) - Fatigue strength coefficient of the material
- \( \epsilon_f \) - Fatigue strain coefficient of the material
- \( q \) - Notch sensitivity factor. (default is 1 for no notch)
- \( K_t \) - Stress concentration factor. (default is 1 for no notch)
- \( \text{mean_stress_correction_method} \) - must be either ’morrow’, ’modified_morrow’, or ’SWT’. Default is ’modified_morrow’ as this is the same as the uncorrected Coffin-Manson relationship when mean stress is zero.

Outputs:

- The results will be printed to the console if print_results is True
  - \( \sigma_{\text{max}} \)
  - \( \sigma_{\text{min}} \)
  - \( \sigma_{\text{mean}} \)
  - \( \epsilon_{\text{max}} \)
  - \( \epsilon_{\text{min}} \)
  - \( \epsilon_{\text{mean}} \)
  - \( \text{cycles_to_failure} \)

In the following example we will provide the function with the appropriate inputs for our problem (taken from Example 2.8 in Probabilistic Physics of Failure Approach to Reliability (2017)). The \( \text{mean_stress_correction_method} \) is changed to ’SWT’ and the results will be printed to the console.

```python
from reliability.PoF import fracture_mechanics_crack_initiation
fracture_mechanics_crack_initiation(P=0.15, A=5*80, Kt=2.41, q=0.9857, Sy=690,
E=210000, K=1060, n=0.14, b=-0.081, c=-0.65, sigma_f=1160, epsilon_f=1.1,
mean_stress_correction_method=’SWT’)
```

A crack of 1 mm will be formed after: 2919.91 cycles (5839.82 reversals)
24.2 Crack growth

The function `fracture_mechanics_crack_growth` uses the principles of fracture mechanics to find the number of cycles required to grow a crack from an initial length until a final length. The final length (a_final) may be specified, but if not specified then a_final will be set as the critical crack length (a_crit) which causes failure due to rapid fracture. This function performs the same calculation using two methods: simplified and iterative. The simplified method assumes that the geometry factor (f(g)), the stress (\(S_{\text{net}}\)), and the critical crack length (a_crit) are constant. This method is the way most textbooks show these problems solved as they can be done by hand in a few steps. The iterative method does not make those assumptions and as a result, the parameters f(g), S_net and a_crit must be recalculated based on the current crack length at every cycle.

This function is applicable only to thin plates with a through thickness edge crack or a centre crack (which is to be specified using the parameter crack_type). You may also use this function for notched components (edge notches only, not centre holes) by specifying the parameters Kt and D which are based on the geometry of the notch. For any notched components, this method assumes the notched component has a “shallow notch” where the notch depth (D) is much less than the plate width (W). The value of Kt for notched components may be found on the eFatigue website. In the case of notched components, the local stress concentration from the notch will often cause slower crack growth. In these cases, the crack length is calculated in two parts (stage 1 and stage 2) which can clearly be seen on the plot using the iterative method (as shown in the example below).

Inputs:

- \(K_c\) - fracture toughness
- \(K_t\) - stress concentration factor (default is 1 for no notch).
- \(D\) - depth of the notch (mm) (default is None for no notch). A notched component is assumed to be doubly-notched (symmetric notches on both sides so that no bending occurs)
- \(C\) - material constant (sometimes referred to as A)
- \(m\) - material constant (sometimes referred to as n). This value must not be 2.
- \(P\) - external load on the material (MPa)
- \(t\) - plate thickness (mm)
- \(W\) - plate width (mm)
- \(a_{\text{initial}}\) - initial crack length (mm) (default is 1 mm)
- \(a_{\text{final}}\) - final crack length (mm) - default is None in which case a_final is assumed to be a_crit (length at failure). It is useful to be able to enter a_final in cases where there are different loading regimes over time.
- crack_type - must be either ‘edge’ or ‘center’. Default is ‘edge’. The geometry factor used for each of these in the simplified method is \(f(g) = 1.12\) for edge and \(f(g) = 1.0\) for center. The iterative method calculates these values exactly using \(a_{\text{initial}}\) and \(W\) (plate width).
- print_results - True/False. Default is True
- show_plot - True/False. Default is True.

Outputs:
• If print_results is True, all outputs will be printed with a description of the process.
• If show_plot is True, the crack growth plot will be shown for the iterative method.
• Nf_stage_1_simplified (in the case of single stage calculations this will be zero)
• Nf_stage_2_simplified
• Nf_total_simplified
• final_crack_length_simplified
• transition_length_simplified
• Nf_stage_1_iterative (in the case of single stage calculations this will be zero)
• Nf_stage_2_iterative
• Nf_total_iterative
• final_crack_length_iterative
• transition_length_iterative

In the following example, a crack of 1mm is grown to failure. The function determines that the notch (described by Kt and D) causes a local stress concentration which initially slows the propogation of the crack until the crack reaches the transition length. Once past the transition length, the crack grows much faster and results in brittle fracture of the material. This change in crack growth rate is evident on the plot from the iterative method. The reason for the different transition lengths between the simplified and iterative methods is that the simplified method uses 1.12 for the geometry factor whereas the iterative method finds the geometry factor using the local geometry (using W and D).

```python
from reliability.PoF import fracture_mechanics_crack_growth
import matplotlib.pyplot as plt
fracture_mechanics_crack_growth(Kc=66, C=6.91*10**-12, m=3, P=0.15, W=100, t=5, Kt=2.41, D=10)
plt.show()
```

SIMPLIFIED METHOD (keeping f(g), S_max, and a_crit as constant):
Crack growth was found in two stages since the transition length (2.08 mm) due to the notch, was greater than the initial crack length (1 mm).
Stage 1 (a_initial to transition length): 6802 cycles
Stage 2 (transition length to a_final): 1133 cycles
Total cycles to failure: 7935 cycles.
Critical crack length to cause failure was found to be: 7.86 mm.

ITERATIVE METHOD (recalculating f(g), S_max, and a_crit for each cycle):
Crack growth was found in two stages since the transition length (2.45 mm) due to the notch, was greater than the initial crack length (1 mm).
Stage 1 (a_initial to transition length): 7576 cycles
Stage 2 (transition length to a_final): 671 cycles
Total cycles to failure: 8247 cycles.
Critical crack length to cause failure was found to be: 6.39 mm.

144 Chapter 24. Fracture mechanics
24.2. Crack growth

References:

Creep is the progressive accumulation of plastic strain in a component under stress at an elevated temperature over a period of time. All creep modelling requires data that is unique to the material undergoing creep since all materials behave differently. This data may be stress, temperature, and time to failure data, or it may be material constants which are derived from the former. This section of reliability contains two functions to determine time to failure due to creep. These functions are `creep_rupture_curves` and `creep_failure_time`. Creep is generally modelled using the Larson-Miller relation or the Manson-Haferd relation.

The function `creep_rupture_curves` plots the creep rupture curves for a given set of creep data. The function also fits the lines of best fit to each temperature. The time to failure for a given temperature can be found by specifying `stress_trace` and `temp_trace`.

**Inputs:**
- `temp_array` - an array or list of temperatures
- `stress_array` - an array or list of stresses
- `TTF_array` - an array or list of times to failure at the given temperatures and stresses
- `stress_trace` - The stress to use for finding the time to failure (only 1 value is accepted)
- `temp_trace` - The temperature to use for finding the time to failure (only 1 value is accepted)

**Outputs:**
- The plot is the only output. Use `plt.show()` to show it.

In the following example (taken from example 2.16 of Probabilistic Physics of Failure Approach to Reliability (2017)), we provide creep data in the form of temperatures, stresses, and times to failure in order to obtain the creep rupture curves. We also are interested in the time to failure of a component at a stress of 70 and a temperature of 1100.

```python
from reliability.PoF import creep_rupture_curves
import matplotlib.pyplot as plt

TEMP = [900, 900, 900, 900, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1100, 1200, 1200, 1200, 1200, 1350, 1350, 1350]
STRESS = [90, 82, 78, 70, 75, 68, 60, 56, 49, 43, 38, 60.5, 50, 40, 29, 22, 40, 30, 25, 20, 15, 10]
TTF = [37, 975, 3581, 9878, 7, 17, 213, 1493, 2491, 5108, 7390, 10447, 18, 167, 615, 2220, 6637, 19, 102, 125, 331, 3, 7, 8, 9, 31, 8]

creep_rupture_curves(temp_array=TEMP, stress_array=STRESS, TTF=TTF, stress_trace=70, temp_trace=1100)
plt.show()
```

(continues on next page)
The function `creep_failure_time` uses the Larson-Miller relation to find the time to failure due to creep. The method uses a known failure time (`time_low`) at a lower failure temperature (`temp_low`) to find the unknown failure time at the higher temperature (`temp_high`). This relation requires the input temperatures in Fahrenheit. To convert Celsius to Fahrenheit use $F = C \times (9/5) + 32$. Also note that the conversion between Fahrenheit and Rankine used in this calculation is $R = F + 459.67$.

Inputs:

- `temp_low` - temperature (in degrees Fahrenheit) where the `time_low` is known
- `temp_high` - temperature (in degrees Fahrenheit) which `time_high` is unknown and will be found by this function
- `time_low` - time to failure at `temp_low`
- `C` - creep constant (default is 20). Typically 20-22 for metals
- `print_results` - True/False

Outputs:

- The time to failure at the higher temperature.
- If `print_results` is True, the output will also be printed to the console.
In the following example (which follows on from the previous example), we will use the Larson-Miller relation to find the time to failure due to creep at 1100°F for a component which we know fails at 9878 hours when subjected to the same stress at 900°F.

```python
from reliability.PoF import creep_failure_time
creep_failure_time(temp_low=900, temp_high=1100, time_low=9878)

,...
The time to failure at a temperature of 1100 °F is 8.27520045913433
The Larson-Miller parameter was found to be 32624.83162890552

,...
```

References:

The function `palmgren_miner_linear_damage` uses the Palmgren-Miner linear damage hypothesis to find the outputs listed below.

**Inputs:**
- `rated_life` - an array or list of how long the component will last at a given stress level
- `time_at_stress` - an array or list of how long the component is subjected to the stress that gives the `rated_life`
- `stress` - what stress the component is subjected to. Not used in the calculation but is required for printing the output.

Ensure that the `time_at_stress` and `rated_life` are in the same units. The answer will also be in those units. The number of items in each input must be the same.

**Outputs:**
- Fraction of life consumed per load cycle
- Service life of the component
- Fraction of damage caused at each stress level

In the following example, we consider a scenario in which ball bearings fail after 50000 hrs, 6500 hrs, and 1000 hrs, after being subjected to a stress of 1kN, 2kN, and 4kN respectively. If each load cycle involves 40 mins at 1kN, 15 mins at 2kN, and 5 mins at 4kN, how long will the ball bearings last?

```python
from reliability.PoF import palmgren_miner_linear_damage
palmgren_miner_linear_damage(rated_life=[50000, 6500, 1000], time_at_stress=[40/60, 15/60, 5/60], stress=[1, 2, 4])
```

```
Palmgren-Miner Linear Damage Model results:
Each load cycle uses 0.01351 % of the components life.
The service life of the component is 7400.37951 load cycles.
The amount of damage caused at each stress level is:
Stress = 1 , Damage fraction = 9.86717 %.
```

(continues on next page)
Stress = 2, Damage fraction = 28.463 %.
Stress = 4, Damage fraction = 61.66983 %.

References:

The Arrhenius model for Acceleration factor due to higher temperature is \( AF = e^{\frac{E_a}{k_B} \left( \frac{1}{T_{use}} - \frac{1}{T_{acc}} \right)} \). This function accepts \( T_{use} \) as a mandatory input and you may specify any two of the three other variables, and the third variable will be found.

Inputs:
- \( T_{use} \) - Temp of usage in Celsius
- \( T_{acc} \) - Temp of acceleration in Celsius (optional input)
- \( E_a \) - Activation energy in eV (optional input)
- \( AF \) - Acceleration factor (optional input)
- \( \text{print\_results} \) - True/False. Default is True

Outputs:
- Results will be printed to console if \( \text{print\_results} \) is True
- \( AF \) - Acceleration Factor
- \( T_{acc} \) - Accelerated temperature
- \( T_{use} \) - Use temperature
- \( E_a \) - Activation energy (in eV)

In the example below, the acceleration factor is found for an accelerated test at 100°C for a component that is normally run at 60°C and has an activation energy of 1.2 eV.

```python
from reliability.PoF import acceleration_factor
acceleration_factor(T_use=60, T_acc=100, Ea=1.2)
```

```
Acceleration Factor: 88.29574588463338
Use Temperature: 60 °C
Accelerated Temperature: 100 °C
```
Solving simultaneous equations with sympy

This document is a tutorial for how to use the Python module sympy to solve simultaneous equations. Since sympy does this so well, there is no need to implement it within reliability, but users may find this tutorial helpful as problems involving physics of failure will often require the solution of simultaneous equations. Sympy is not installed by default when you install reliability so users following this tutorial will need to ensure sympy is installed on their machine. The following three examples should be sufficient to illustrate how to use sympy for solving simultaneous equations. Further examples are available in the sympy documentation.

Example 1
Eqn 1: \( x + y = 5 \)
Eqn 2: \( x^2 + y^2 = 17 \)
Solving with sympy:

```python
import sympy as sym
x, y = sym.symbols('x,y')
eq1 = sym.Eq(x + y, 5)
eq2 = sym.Eq(x**2 + y**2, 17)
result = sym.solve([eq1, eq2], (x, y))
print(result)
```
```python
[(1, 4), (4, 1)] #these are the solutions for x,y. There are 2 solutions because the equations represent a line passing through a circle.
```

Example 2
Eqn 1: \( 1000000a + 100b = 119.54907 \)
Eqn 2: \( 1000a + b = 405 \)
Solving with sympy:
import sympy as sym

a, b = sym.symbols('a,b')
eq1 = sym.Eq(a*1000000**b, 119.54907)
eq2 = sym.Eq(a*1000**b, 405)
result = sym.solve([eq1, eq2], (a, b))
print(result)

...  
[(1372.03074854535, -0.176636273742481)]  # these are the solutions for a, b
...  

Example 3

Eqn 1: \(2x^2 + y + z = 1\)
Eqn 2: \(x + 2y + z = c_1\)
Eqn 3: \(-2x + y = -z\)

The actual solution to the above set of equations is:

\[
x = -\frac{1}{2} + \frac{\sqrt{3}}{2}
y = c_1 - \frac{3\sqrt{3}}{2} + \frac{3}{2}
z = -c_1 - \frac{5}{2} + 5\sqrt{3}
\]

and a second solution:

\[
x = -\frac{1}{2} - \frac{\sqrt{3}}{2}
y = c_1 + \frac{3\sqrt{3}}{2} + \frac{3}{2}
z = -c_1 - \frac{5}{2} - 5\sqrt{3}
\]

Solving with sympy:

import sympy as sym

x, y, z = sym.symbols('x,y,z')
c1 = sym.Symbol('c1')
eq1 = sym.Eq(2*x**2 + y + z, 1)
eq2 = sym.Eq(x + 2*y + z, c1)
eq3 = sym.Eq(-2*x + y, -z)
result = sym.solve([eq1, eq2, eq3], (x, y, z))
print(result)

...  
[(-1/2 + sqrt(3)/2, c1 - 3*sqrt(3)/2 + 3/2, -c1 - 5/2 + 5*sqrt(3)/2), (-sqrt(3)/2 - 1/2, -2, c1 + 3/2 + 3*sqrt(3)/2, -c1 - 5*sqrt(3)/2 - 5/2)]
...  

Note: If you are using an iPython notebook, the display abilities are much better than the command line interface, so you can simply add sym.init_printing() after the import line and your equations should be displayed nicely.

A special thanks to Brigham Young University for offering this tutorial.
RELIABILITY
A Python library for reliability engineering
Before reading this section, you should be familiar with what a probability plot is and how to use it. For a detailed explanation, please see the section on probability plots.

The module `reliability.ALT_probability_plotting` contains four ALT probability plotting functions. These functions are:

- `ALT_probability_plot_Weibull`
- `ALT_probability_plot_Lognormal`
- `ALT_probability_plot_Normal`
- `ALT_probability_plot_Exponential`

An ALT probability plot produces a multi-dataset probability plot which includes the probability plots for the data and the fitted distribution at each stress level, as well as a refitted distribution assuming a common shape parameter at each stress level. All of these functions perform in a similar way, with the main difference being the distribution that is fitted. The probability plots provided do not include the Beta distribution (because there are two shape parameters), the Gamma distribution (because changing the scale parameter will also change the slope of the line even when the shape parameter is constant) or any of the location shifted distributions (because these are not typically used for ALT probability plotting). The `ALT_probability_plot_Exponential` is a special case of the `ALT_probability_plot_Weibull` in which the common shape parameter is forced to be 1. An example of this is shown below.

When producing the ALT probability plot, the function automates the following process: fit a distribution to the data for each unique stress level, find the common shape parameter (several methods are provided), refit the distribution to the data for each unique stress level whilst forcing the shape parameter to be equal to the common shape parameter, plot the data along with the original and new fitted distributions, calculate the change in the common shape parameter from the original shape parameter to see if the model is applicable to this dataset. Each of the ALT plotting functions listed above has the following inputs and outputs.

**Inputs:**

- `failures` - an array or list of all the failure times
- `failure_stress` - an array or list of the corresponding stresses (such as temperature) at which each failure occurred. This must match the length of failures as each failure is tied to a failure stress.
- `right_censored` - an array or list of all the right censored failure times
right_censored_stress - an array or list of the corresponding stresses (such as temperature) at which each right_censored datapoint was obtained. This must match the length of right_censored as each right_censored value is tied to a right_censored stress.

print_results - True/False. Default is True

show_plot - True/False. Default is True

common_shape_method - ‘BIC’, ‘weighted_average’, ‘average’. Default is ‘BIC’. This is the method used to obtain the common_shape (beta) parameter. ‘BIC’ will find the common_shape that gives lowest total BIC (equivalent to the best overall fit), ‘weighted_average’ will perform a weighted average based on the amount of data (failures and right censored) for each stress, ‘average’ is simply the average. Note for the Lognormal and Normal plots, this variable is named common_sigma_method as we are forcing sigma to be a common value.

Outputs:

The plot will be produced if show_plot is True

A dataframe of the fitted distributions parameters will be printed if print_results is True

results - a dataframe of the fitted distributions parameters and change in the shape parameter

common_shape - the common shape parameter. This is Beta for Weibull, Sigma for Lognormal and Normal, and 1 for Exponential.

BIC_sum - the sum of the BIC for each of the distributions when fitted using the common_shape

AICc_sum - the sum of the AICc for each of the distributions when fitted using the common_shape

The time to run the function will be a few seconds if you have a large amount of data and the common_shape_method is set to ‘BIC’. This is because the distributions need to be refitted for each iteration of the optimizer (which is usually around 20 to 30 iterations). With 100 datapoints this should take less than 5 seconds for the ‘BIC’ method, and less than 1 second for the ‘average’ and ‘weighted_average’ methods. The more data you have, the longer it will take, so please be patient as a lot of computation is required.

In the following example we will use a dataset from reliability.Datasets which contains failures and right_censored data for three stress levels. We will analyse this dataset using the Weibull and Lognormal ALT probability plots to determine which model is a better fit for the data. All other inputs are left to their default values which gives us the plot and the results dataframe. From the printed results we can see how well the model fits our data. The AICc and BIC values suggest that the Lognormal model is a slightly better fit overall for this dataset, but both models would be suitable. The fitted distributions with a common shape parameter still agree well with the majority of our data (except for the lower tail of the 40 degree data), and the amount of change to the shape parameter was within the acceptable limits. See the section below for more details on what we are looking to get out of these plots.

from reliability.ALT_probability_plotting import ALT_probability_plot_Weibull, ALT_probability_plot_Lognormal
from reliability.Datasets import ALT_temperature
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(121)
ALT_probability_plot_Weibull(failures=ALT_temperature().failures,failure_stress=ALT_temperature().failure_stresses,right_censored=ALT_temperature().right_censored,right_censored_stress=ALT_temperature().right_censored_stresses)
plt.subplot(122)
ALT_probability_plot_Lognormal(failures=ALT_temperature().failures,failure_stress=ALT_temperature().failure_stresses,right_censored=ALT_temperature().right_censored,right_censored_stress=ALT_temperature().right_censored_stresses)
plt.gcf().set_size_inches(15,7)
plt.show()
In this second example, we examine the difference between `ALT_probability_plot_Weibull` and `ALT_probability_plot_Exponential`. A dataset is generated from several Exponential distributions. Ideally, we want to fit a distribution to this data which does not overfit, such that it should have as few parameters as necessary. Both the Weibull and Exponential distributions could be used here, but we know the Exponential is a more appropriate distribution since it was the source of the data. Upon examination of the results, we see very little difference between the common shape (from Exponential) and common beta (from Weibull) and very little difference in the plots, but the AICc and BIC are both slightly lower for the Exponential model indicating that the Exponential distribution should be used preferentially to the Weibull distribution (this result may change if the seed is changed to produce different data). Conveniently, the function `ALT_probability_plot_Exponential` also provides the AICc and BIC results from Weibull and will print a warning if it finds Weibull to be a more appropriate fit than Exponential based on the BIC.

```python
from reliability.ALT_probability_plotting import ALT_probability_plot_Weibull, ALT_probability_plot_Exponential
import matplotlib.pyplot as plt
import numpy as np
from reliability.Distributions import Exponential_Distribution

# create the data using an Exponential distribution
```
data1 = Exponential_Distribution(Lambda=1 / 100).random_samples(20, seed=2)
data2 = Exponential_Distribution(Lambda=1 / 500).random_samples(20, seed=3)
data3 = Exponential_Distribution(Lambda=1 / 3000).random_samples(20, seed=4)
f = np.hstack([data1, data2, data3])
f_stress = np.hstack([np.ones_like(data1) * 50, np.ones_like(data1) * 40, np.ones_like(data1) * 30])

# plot the data
plt.subplot(121)
ALT_probability_plot_Exponential(failures=f, failure_stress=f_stress)
plt.subplot(122)
ALT_probability_plot_Weibull(failures=f, failure_stress=f_stress, common_shape_method='average')
plt.gcf().set_size_inches((11,7))
plt.show()

```
ALT Exponential probability plot results:
<table>
<thead>
<tr>
<th>stress</th>
<th>weibull alpha</th>
<th>weibull beta</th>
<th>new 1/Lambda</th>
<th>common shape change</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>3950.302619</td>
<td>0.775461</td>
<td>4154.505068</td>
<td>1.0 +28.96%</td>
</tr>
<tr>
<td>40.0</td>
<td>366.204477</td>
<td>1.066262</td>
<td>357.669530</td>
<td>1.0 -6.21%</td>
</tr>
<tr>
<td>50.0</td>
<td>73.371485</td>
<td>1.254340</td>
<td>68.352088</td>
<td>1.0 -20.28%</td>
</tr>
</tbody>
</table>
Total AICc: 864.1158725562174
Total BIC: 866.4364027102126
Total AICc (weibull): 864.3730503138722
Total BIC (weibull): 866.6935804678675

ALT Weibull probability plot results:
<table>
<thead>
<tr>
<th>stress</th>
<th>original alpha</th>
<th>original beta</th>
<th>new alpha</th>
<th>common beta</th>
<th>beta change</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>3950.302619</td>
<td>0.775461</td>
<td>3950.303608</td>
<td>1.032021</td>
<td>+33.08%</td>
</tr>
<tr>
<td>40.0</td>
<td>366.204477</td>
<td>1.066262</td>
<td>361.817836</td>
<td>1.032021</td>
<td>-3.21%</td>
</tr>
<tr>
<td>50.0</td>
<td>73.371485</td>
<td>1.254340</td>
<td>68.996497</td>
<td>1.032021</td>
<td>-17.72%</td>
</tr>
</tbody>
</table>
Total AICc: 864.3730503138722
Total BIC: 866.6935804678675
```
29.1 Getting your input data in the right format

Because the ALT probability plots need failures and right censored data from many stress levels, it was not practical to make an input for each stress level. Instead, the failure times are combined in a single input and the failure_stress input provides a list of the corresponding stresses at which each failure occurred. The same is true of the right_censored and right_censored_stress inputs.

To get your data in the correct format, ensure you have combined all your failure times into a single list or numpy array and there is a corresponding list or array of the same length that provides all of the stresses. The following example illustrates one method to do this if you do not have the list already imported from Excel or another source. This is done for failures only but if you have right_censored data then you would do the same thing, but keeping it separate to the failure data. There is no need to sort the data in any particular order as this is all done automatically. The only requirement is that the length of failures matches the length of the failure_stress, and that there are no new stresses in right_censored_stress that are not present in failure_stress.

```python
import numpy as np

# create the data
failure_times_at_stress_1 = [800, 850, 910, 940]
failure_stress_1 = [40, 40, 40, 40]
failure_times_at_stress_2 = [650, 670, 715, 740]
failure_stress_2 = [50, 50, 50, 50]
failure_times_at_stress_3 = [300, 320, 350, 380]
failure_stress_3 = [60, 60, 60, 60]

# combine the data
failures = np.hstack([failure_times_at_stress_1, failure_times_at_stress_2, failure_times_at_stress_3])
failure_stresses = np.hstack([failure_stress_1, failure_stress_2, failure_stress_3])
```

(continues on next page)
#print for inspection
print(failures)
print(failure_stresses)

[800 850 910 940 650 670 715 740 300 320 350 380]
[40 40 40 40 50 50 50 50 60 60 60 60]

29.2 What does an ALT probability plot show me?

An ALT probability plot shows us how well our dataset can be modeled by the chosen distribution. This is more than just a goodness of fit at each stress level, because the distribution needs to be a good fit at all stress levels and be able to fit well with a common shape parameter. If you find the shape parameter changes significantly as the stress increases then it is likely that your accelerated life test is experiencing a different failure mode at higher stresses. When examining an ALT probability plot, the main things we are looking for are:

- Does the model appear to fit the data well at all stress levels (i.e. the dashed lines pass reasonably well through all the data points)
- Examine the AICc and BIC values when comparing multiple models. A lower value suggests a better fit.
- Is the amount of change to the shape parameter within the acceptable limits (generally less than 50% for each distribution).

The image provided above shows two distributions that fit well. If we apply the same data to the function ALT_probability_plot_Normal as shown in the example below, we get the image shown below. From this image we can see that the model does not fit well at the higher stress (80 degrees) and the amount of change to the shape parameter was up to 93%. Also note that the total AIC and total BIC for the Normal_2P model is higher (worse) than for the Weibull_2P and Lognormal_2P models shown in the first example. Based on these results, we would reject the Normal_2P model and try another model. If you find that none of the models work without large changes to the shape parameter at the higher stresses, then you can conclude that there must be a change in the failure mode for higher stresses and you may need to look at changing your accelerated test to keep the failure mode consistent across tests.

```python
from reliability.ALT_probability_plotting import ALT_probability_plot_Normal
from reliability.Datasets import ALT_temperature
import matplotlib.pyplot as plt
ALT_probability_plot_Normal(failures=ALT_temperature().failures,failure_stress=ALT_temperature().failure_stresses)
plt.show()

ALT Normal probability plot results:

<table>
<thead>
<tr>
<th>stress</th>
<th>original mu</th>
<th>original sigma</th>
<th>new mu</th>
<th>common sigma</th>
<th>sigma change</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>9098.952677</td>
<td>3203.855879</td>
<td>7764.809302</td>
<td>2258.04215</td>
<td>-29.52%</td>
</tr>
<tr>
<td>60</td>
<td>5174.454788</td>
<td>3021.349445</td>
<td>4756.980035</td>
<td>2258.04215</td>
<td>-25.26%</td>
</tr>
<tr>
<td>80</td>
<td>1600.177190</td>
<td>1169.695509</td>
<td>1638.730675</td>
<td>2258.04215</td>
<td>+93.05%</td>
</tr>
</tbody>
</table>

Total AICc: 709.5115334757447
Total BIC: 713.4159440639235
```

Chapter 29. ALT probability plots
29.2. What does an ALT probability plot show me?

References:
Chapter 29. ALT probability plots
CHAPTER 30

Fitting a model to ALT data

Before reading this section, you should be familiar with ALT probability plots, and Fitting distributions to non-ALT datasets.

The module `reliability.ALT_fitters` contains fitting function for 20 different ALT life-stress models. Each model is a combination of the life model with the scale or location parameter replaced with a life-stress model. For example, the Weibull-Exponential model is found by replacing the $\alpha$ parameter with the equation for the exponential life-stress model as follows:

Weibull PDF:

$$f(t) = \frac{\beta t^{\beta-1}}{\alpha^\beta} \cdot e^{x} \left( -\left( \frac{t}{\alpha} \right)^{\beta} \right)$$

Exponential Life-Stress model:

$$L(T) = b \cdot e^{x} \left( \alpha \right)$$

Replacing $\alpha$ with $L(T)$ gives the PDF of the Weibull-Exponential model:

Weibull-Exponential:

$$f(t, T) = \frac{\beta t^{\beta-1}}{\left( b \cdot e^{x} \left( \alpha \right) \right)^{\beta}} \cdot e^{x} \left( -\left( \frac{t}{\left( b \cdot e^{x} \left( \alpha \right) \right)} \right)^{\beta} \right)$$

The correct substitutions for each type of model are:

- Weibull: $\alpha = L(T)$
- Normal: $\mu = L(T)$
- Lognormal: $\mu = ln(L(T))$
- Exponential: $\lambda = \frac{1}{L(T)}$

The life models available are:

- Weibull_2P
- Normal_2P
- Lognormal_2P
- Expon_1P

The life-stress models available are:

Exponential (also used for Arrhenius equation):

$$L(T) = b \cdot e^{x} \left( \frac{t}{T} \right)$$
Eyring:

\[ L(T) = \frac{1}{T}.exp\left(-\left(c - \frac{a}{T}\right)\right) \]

Power (also known as inverse power):

\[ L(S) = a.S^n \]

Dual-Exponential (also known as Temperature-Humidity):

\[ L(T, H) = c.exp\left(\frac{a}{T} + \frac{b}{H}\right) \]

Power-Exponential (also known as Thermal-Non-Thermal):

\[ L(T, S) = c.S^n.exp\left(\frac{a}{T}\right) \]

When choosing a model, it is important to consider the physics involved in the life-stress model rather than just trying everything to see what fits best. For example, the Power-Exponential model is most appropriate for a dataset that was obtained from an ALT reliability test with a thermal and a non-thermal stress (such as temperature and voltage). It would be inappropriate to model the data from a Temperature-Humidity test using a Power-Exponential model as the physics suggests that a Temperature-Humidity test should be modelled using the Dual-Exponential model.

Each of the fitting functions works in a very similar way so the documentation below can be applied to all of the models with minor modifications to the parameter names of the outputs. The following documentation is for the Weibull-Power model.

Inputs:

- failures - an array or list of the failure times.
- failure_stress - an array or list of the corresponding stresses (such as temperature) at which each failure occurred. This must match the length of failures as each failure is tied to a failure stress.
- right_censored - an array or list of all the right censored failure times
- right_censored_stress - an array or list of the corresponding stresses (such as temperature) at which each right_censored data point was obtained. This must match the length of right_censored as each right_censored value is tied to a right_censored stress.
- use_level_stress - The use level stress at which you want to know the mean life. Optional input.
- print_results - True/False. Default is True
- show_plot - True/False. Default is True
- CI - confidence interval for estimating confidence limits on parameters. Must be between 0 and 1. Default is 0.95 for 95% CI.
- initial_guess - starting values for \([a,n]\). Default is calculated using a curvefit to failure data. Optional input. If fitting fails, you will be prompted to try a better initial guess and you can use this input to do it.

Outputs:

- a - fitted parameter from the Power model
- n - fitted parameter from the Power model
- beta - the fitted Weibull_2P beta
- loglik2 - LogLikelihood*-2 (as used on JMP Pro)
- loglik - Log Likelihood (as used in Minitab and Reliasoft)
- AICc - Akaike Information Criterion
- BIC - Bayesian Information Criterion
- a_SE - the standard error (sqrt(variance)) of the parameter
- n_SE - the standard error (sqrt(variance)) of the parameter
- beta_SE - the standard error (sqrt(variance)) of the parameter
- a_upper - the upper CI estimate of the parameter
- a_lower - the lower CI estimate of the parameter

Chapter 30. Fitting a model to ALT data
• n_upper - the upper CI estimate of the parameter
• n_lower - the lower CI estimate of the parameter
• beta_upper - the upper CI estimate of the parameter
• beta_lower - the lower CI estimate of the parameter
• results - a dataframe of the results (point estimate, standard error, Lower CI and Upper CI for each parameter)
• mean_life - the mean life at the use_level_stress. Only calculated if use_level_stress is specified

In the following example, we will fit the Weibull-Power model to an ALT dataset obtained from a fatigue test. This dataset can be found in reliability.Datasets. We want to know the mean life at the use level of 60 so the parameter use_level_stress is specified. All other values are left as defaults and the results and plot are shown.

```python
from reliability.ALT_fitters import Fit_Weibull_Power
from reliability.Datasets import ALT_load2
import matplotlib.pyplot as plt
data = ALT_load2()
Fit_Weibull_Power(failures=data.failures, failure_stress=data.failure_stresses, right_censored=data.right_censored, right_censored_stress=data.right_censored_stresses, use_level_stress=60)
plt.show()

Results from Fit_Weibull_Power (95% CI):
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>398816.331485</td>
<td>519397.960450</td>
<td>-619184.964641</td>
<td>1.416818e+06</td>
</tr>
<tr>
<td>n</td>
<td>-1.417306</td>
<td>0.243944</td>
<td>-1.895428</td>
<td>-9.391834e-01</td>
</tr>
<tr>
<td>beta</td>
<td>3.017297</td>
<td>0.716426</td>
<td>1.894563</td>
<td>4.805374e+00</td>
</tr>
</tbody>
</table>

At the use level stress of 60, the mean life is 1075.32845
```
In this second example, we will fit a dual stress model to a dual stress data set. The data set contains temperature and voltage data so it is most appropriate to model this dataset using a Power-Exponential model. A few differences to note with the dual stress models is that each stress requires a separate input, so if you also have censored data then this will require 6 inputs. If using the Power Exponential model it is essential that the thermal and non-thermal stresses go in their named inputs or the model will likely fail to fit the data. In this example we want to know the life at a use level stress of 325K and 0.5V which the output tells us is 4673 hours.

```python
from reliability.ALT_fitters import Fit_Weibull_Power_Exponential
from reliability.Datasets import ALT_temperature_voltage
import matplotlib.pyplot as plt

data = ALT_temperature_voltage()
Fit_Weibull_Power_Exponential(failures=data.failures, failure_stress_thermal=data.failure_stress_temp, failure_stress_nonthermal=data.failure_stress_voltage, use_level_stress=[325, 0.5])
```

(continues on next page)
Results from Fit_Weibull_Power_Exponential (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3404.485691</td>
<td>627.674716</td>
<td>2174.265854</td>
<td>4634.705528</td>
</tr>
<tr>
<td>c</td>
<td>0.087610</td>
<td>0.141217</td>
<td>-0.189170</td>
<td>0.364391</td>
</tr>
<tr>
<td>n</td>
<td>-0.713424</td>
<td>0.277561</td>
<td>-1.257434</td>
<td>-0.169414</td>
</tr>
<tr>
<td>beta</td>
<td>4.997527</td>
<td>1.173998</td>
<td>3.153512</td>
<td>7.919828</td>
</tr>
</tbody>
</table>

At the use level stresses of 325 and 0.5, the mean life is 4673.15347

• Accelerated Life Testing Data Analysis Reference - ReliaWiki, Reliawiki.com, 2019. [Online].
The function `similar_distributions` is a tool for finding the probability distributions that are most similar to an input distribution. It samples the CDF of an input distribution and then fits all other distributions to those samples to determine the best fitting and therefore most similar distributions.

**Inputs**
- `distribution` - a distribution object created using the `reliability.Distributions` module
- `include_location_shifted` - True/False. Default is True. When set to True it will include `Weibull_3P`, `Lognormal_3P`, `Gamma_3P`, `Expon_2P`
- `show_plot` - True/False. Default is True
- `print_results` - True/False. Default is True
- `number_of_distributions_to_show` - the number of similar distributions to show. Default is 3. If the number specified exceeds the number available (typically 8), then the number specified will automatically be reduced.

**Outputs**
- If `show_plot` is True then the plot of PDF and CDF will automatically be shown.
- If `print_results` is True then the parameters of the most similar distributions will be printed.
- `results` - an array of distributions objects ranked in order of best fit.
- `most_similar_distribution` - a distribution object. This is the first item from `results`.

In the example below, we create a Weibull Distribution object using the `reliability.Distributions` module. We then provide the Weibull Distribution as input to `similar_distributions` and the output reveals the top 3 similar distributions. The optional input of `include_location_shifted` has been set to False.

```python
from reliability.Distributions import Weibull_Distribution
from reliability.Other_functions import similar_distributions
dist = Weibull_Distribution(alpha=50, beta=3.3)
similar_distributions(distribution=dist, include_location_shifted=False)
```

(continues on next page)
The input distribution was:
Weibull Distribution (\(\alpha=50, \beta=3.3\))

The top 3 most similar distributions are:
Normal Distribution (\(\mu=44.8471, \sigma=14.9226\))
Gamma Distribution (\(\alpha=5.7607, \beta=7.7849\))
Loglogistic Distribution (\(\alpha=43.465, \beta=4.7564\))
CHAPTER 32

Convert dataframe to grouped lists

This function was written because many of our datasets are in the form of columns with one column indicating the label (e.g., success, failure) and the other column providing the value. Since many functions in reliability are written to accept grouped lists (lists where all the values belong to the same group such as right_censored failure times), it is beneficial to have a fast way to perform this conversion. This function will split the dataframe into as many grouped lists as there are unique values in the labels (left) column.

Inputs:
- input_dataframe. This must be a dataframe containing 2 columns, where the label column is the left column and the value column is the right column. The column titles are not important.

Outputs:
- lists, names - lists is a list of the grouped lists, and names is the identifying labels used to group the lists from the first column.

In the example below, we will create some data in a pandas dataframe, print it to see what it looks like, and then split it up using this function and print the results.

```python
from reliability.Other_functions import convert_dataframe_to_grouped_lists
import pandas as pd

# create some data in a dataframe
data = {'outcome': ['Failed', 'Censored', 'Failed', 'Failed', 'Censored'],
        'cycles': [1253, 1500, 1342, 1489, 1500]}
df = pd.DataFrame(data, columns=['outcome', 'cycles'])
print(df, '

# usage of the function
lists, names = convert_dataframe_to_grouped_lists(df)
print(names[0], lists[0])
print(names[1], lists[1])

... outcome cycles
0 Failed  1253
1 Censored  1500
```

(continues on next page)
2  Failed  1342
3  Failed  1489
4  Censored  1500

Censored [1500, 1500]
Failed [1253, 1342, 1489]

RELIABILITY
A Python library for reliability engineering
Crosshairs

This function provides interactive crosshairs on matplotlib plots. The crosshairs will follow the users’ mouse cursor when they are near lines or points and will snap to these lines and points. Upon a mouse click the crosshairs will add an annotation to the plot. This annotation can be dragged to a new position. To delete the annotation, right click on it. To temporarily hide all annotations, toggle ‘h’ on your keyboard.

Note that crosshairs should be called after everything is added to the plot (but before plt.show()) so that the objects in the plot are identified for the ‘snap to’ feature. If something is added to the plot after calling crosshairs then you will not be able to move the crosshairs onto it.

If your interactive development environment does not generate the plot in its own window then your plot is not interactive and this will not work. For iPython notebook users, the interactive window should be available by typing “%matplotlib qt” after importing matplotlib as described here.

There are some customisable attributes of the crosshairs and annotations using the following inputs:

- xlabel - the x-label for the annotation. Default is x.
- ylabel - the y-label for the annotation. Default is y.
- decimals - the number of decimals to use when rounding values in the crosshairs and in the annotation. Default is 2.
- plotting kwargs are also accepted. eg. color, linestyle, etc.

In the following example, we see the crosshairs being used to display the value of the Weibull CDF. The dynamic nature of this feature is shown in the video at the bottom of this page.

```python
from reliability.Other_functions import crosshairs
from reliability.Distributions import Weibull_Distribution
import matplotlib.pyplot as plt

Weibull_Distribution(alpha=50,beta=2).CDF()
crosshairs(xlabel='t',ylabel='F') #it is important to call this last
plt.show()
```
A special thanks goes to Antony Lee, the author of mplcursors. The crosshairs function works using mplcursors to enable the ‘snap to’ feature and the annotations. Antony was very helpful in getting this to work.
The distribution explorer is a simple way to explore the shape of each distribution based on its parameters. To achieve this, an interactive window is shown with the 5 characteristic functions (PDF, CDF, SF, HF, CHF) of each probability distribution. Parameters can be changed using slider widgets. Distributions can be changed using the radio button widget.

There are no inputs or outputs. Everything is done within the interactive matplotlib window. Please see the video for an example of the interactive features of the distribution explorer.

To open the distribution explorer, use the following code:

```python
from reliability.Other_functions import distribution_explorer
distribution_explorer()
```
Chapter 34. Distribution explorer
CHAPTER 35

Histogram

This function plots a histogram using the matplotlib histogram (plt.hist()), but adds some additional features. Default formatting is improved, the number of bins is optimized by default, and there is an option to shade the bins white above a chosen threshold. If you would like to specify the number of bins rather than having the optimal number calculated, then the bins argument allows this.

Inputs:

- data - the data to plot. Array or list.
- white_above - bins above this value will be shaded white
- bins - array of bin edges or string in ['auto','fd','doane','scott','stone','rice','sturges','sqrt']. Default is 'auto'. See numpy for more information on bin edges.
- density - True/False. Default is True. Always use True if plotting with a probability distribution.
- cumulative - True/False. Default is False. Use False for PDF and True for CDF.
- kwargs - plotting kwargs for the histogram (color, alpha, etc.)

The following example shows the difference between the appearance of the default histogram in matplotlib, and the histogram in reliability.

```python
from reliability.Distributions import Gamma_Distribution
from reliability.Fitters import Fit_Gamma_2P
from reliability.Other_functions import make_right_censored_data, histogram
import matplotlib.pyplot as plt

a = 30
b = 4
threshold = 180  # this is used when right censoring the data
dist = Gamma_Distribution(alpha=30, beta=4)
raw_data = dist.random_samples(500, seed=2)  # create some data. Seeded for repeatability
data = make_right_censored_data(raw_data, threshold=threshold)  # right censor the data
gf = Fit_Gamma_2P(failures=data.failures, right_censored=data.right_censored, show_probability_plot=False, print_results=False)
```

(continues on next page)
plt.subplot(121)
g.f.distribution.PDF()
plt.hist(raw_data, density=True)  # default histogram from matplotlib
plt.title('matplotlib histogram')

plt.subplot(122)
g.f.distribution.PDF()
histogram(raw_data, white_above=threshold)  # histogram from reliability - better,
   formatting, optimal bin width by default, white_above option
plt.title('reliability histogram')

plt.subplots_adjust(right=0.95, wspace=0.38)
plt.show()
This function is a tool to convert complete data to complete and right censored data. Two methods are available which enable the production of either singly-censored or multiply-censored data. This function is often used in testing of the Fitters or Nonparametric functions when some right censored data is needed.

Inputs:
- `data` - list or array of data
- `threshold` - number or None. Default is None. If number this is the point to right censor (right censoring is done if data > threshold). This is known as “singly censored data” as everything is censored at a single point.
- `fraction_censored` - number between 0 and 1. Defautl is 0.5. Censoring is done randomly. This is known as “multiply censored data” as there are multiple times at which censoring occurs. If both threshold and fraction_censored are None, fraction_censored will default to 0.5 to produce multiply censored data. If both threshold and fraction_censored are specified, an error will be raised since these methods conflict.
- `seed` - sets the random seed. This is used for multiply censored data (i.e. when threshold is None). The data is shuffled to remove censoring bias that may be caused by any pre-sorting. Specifying the seed ensures a repeatable random shuffle.

Outputs:
- `failures` - array of failure data
- `right_censored` - array of right_censored data

In this first example we will look at the production of singly censored data. That is data which is all censored at the same value (defined by threshold).

```python
from reliability.Other_functions import make_right_censored_data
output = make_right_censored_data(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], threshold=6)
print('Failures:', output.failures)
print('Right Censored:', output.right_censored)
```

```
Failures: [1 2 3 4 5 6]
(continues on next page)```
In this second example we will look at the production of multiply censored data. That is data which is censored at different values. The amount of data to be censored is governed by `fraction_censored`. If unspecified it will default to 0.5 resulting in 50% of the data being right censored. Note that there is some randomness to the censoring. For repeatability set the seed.

```python
from reliability.Other_functions import make_right_censored_data
output = make_right_censored_data(data=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], fraction_censored=0.5, seed=1)
print('Failures:', output.failures)
print('Right Censored:', output.right_censored)

```

```
Failures: [4 1 5 2 3] # half of the data has not been censored. It has been shuffled - so its order will be different from the order of the input data.
Right Censored: [5.89006504 8.71327034 4.27673283 3.11056676 2.728583] # half of the data has been censored at some value between 0 and the original value
```
Datasets

There are a few datasets that have been included with reliability that users may find useful for testing and experimenting. While this list is currently small, expect it to increase significantly over time. Within reliability, the following datasets are available:

**Standard datasets**

- automotive - 10 failures, 21 right censored. It is used in this example
- defective_sample - 1350 failures, 12296 right censored. It exhibits the behavior of a defective sample (also known as Limited failure population or Defective subpopulation).
- electronics - 10 failures, 4072 right censored. It is used in this example.
- mileage - 100 failures with no right censoring. It is used in the examples for KStest and chi2test.

**ALT Datasets**

- ALT_temperature - conducted at 3 temperatures. 35 failures, 102 right censored. For example usage of many of the ALT Datasets see the examples here.
- ALT_temperature2 - conducted at 4 temperatures. 40 failures, 20 right censored.
- ALT_temperature3 - conducted at 3 temperatures. 30 failures, 0 right censored.
- ALT_load - conducted at 3 loads. 20 failures, 0 censored.
- ALT_load2 - conducted at 3 loads. 13 failures, 5 right censored.
- ALT_temperature_voltage - conducted at 2 different temperatures and 2 different voltages. 12 failures, 0 right censored.
- ALT_temperature_voltage2 - conducted at 3 different temperatures and 2 different voltages. 18 failures, 8 right censored.
- ALT_temperature_humidity - conducted at 2 different temperatures and 2 different humidities. 12 failures, 0 right censored.

**MCF Datasets**
• MCF_1 - this dataset contains failure and retirement times for 5 repairable systems. Exhibits a worsening repair rate.

• MCF_2 - this dataset contains failure and retirement times for 56 repairable systems. Exhibits a worsening then improving repair rate. Difficult to fit this dataset.

All datasets are functions which create objects and every dataset object has several attributes.

For the standard datasets, these attributes are:

• info - a dataframe of statistics about the dataset
• failures - a list of the failure data
• right_censored - a list of the right_censored data
• right_censored_stress - a list of the right_censored stresses (ALT datasets only)

For the ALT datasets, these attributes are similar to the above standard attributes, just with some variation for the specific dataset. These include things like:

• failure_stress_humidity
• right_censored_stress_voltage
• failure_stress_temp
• other similarly named attributes based on the dataset

For the MCF datasets these attributes are:

• times
• number_of_systems

If you would like more information on a dataset, you can type the name of the dataset in the help function (after importing it).

```python
from reliability.Datasets import automotive
print(help(automotive))
```

If you would like the statistics about a dataset you can access the info dataframe as shown below.

```python
from reliability.Datasets import defective_sample
print(defective_sample().info)
```

```
<table>
<thead>
<tr>
<th>Stat</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>defective_sample</td>
</tr>
<tr>
<td>Total Values</td>
<td>13645</td>
</tr>
<tr>
<td>Failures</td>
<td>1350 (9.89%)</td>
</tr>
<tr>
<td>Right Censored</td>
<td>12295 (90.11%)</td>
</tr>
</tbody>
</table>
```

The following example shows how to import a dataset and use it. Note that we must use () before accessing the failures and right_censored values.

```python
from reliability.Datasets import automotive
from reliability.Fitters import Fit_Weibull_2P
Fit_Weibull_2P(failures=automotive().failures,right_censored=automotive().right_censored,show_probability_plot=False)
```

(continues on next page)
Results from Fit_Weibull_2P (95% CI):

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Point Estimate</th>
<th>Standard Error</th>
<th>Lower CI</th>
<th>Upper CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>140882.303527</td>
<td>49299.609699</td>
<td>70956.382925</td>
<td>279718.647273</td>
</tr>
<tr>
<td>Beta</td>
<td>1.132769</td>
<td>0.301468</td>
<td>0.672370</td>
<td>1.908422</td>
</tr>
</tbody>
</table>

Log-Likelihood: -128.98350896528038

If you have an interesting dataset, please email me (alpha.reliability@gmail.com) and I may include it in this database.

RELIABILITY
A Python library for reliability engineering
One sample proportion

This function calculates the upper and lower bounds of reliability for a given number of trials and successes. It is most applicable to analysis of test results in which there are only success/failure results and the analyst wants to know the reliability of the batch given those sample results.

Inputs:
- trials - the number of trials which were conducted
- successes - the number of trials which were successful
- CI - the desired confidence interval. Defaults to 0.95 for 95% CI.

Outputs:
- lower, upper - Confidence interval limits. Note that this will return nan for lower or upper if the one sided CI is calculated (ie. when successes=0 or successes=trials).

In this example, consider a scenario in which we have a large batch of items that we need to test for their reliability. The batch is large and testing is expensive so we will conduct the test on 30 samples. From those 30 samples, 29 passed the test. If the batch needs at least 85% reliability with a 95% confidence, then should we accept or reject the batch?

```python
from reliability.Reliability_testing import one_sample_proportion
result = one_sample_proportion(trials=30, successes=29)
print(result)
```

```
(0.8278305443665873, 0.9991564290733695)
```

The lower bound (with 95% confidence interval) on the reliability was 82.78%. Since this is below our requirement of 85%, then we should reject the batch.
This function determines if there is a statistically significant difference in the results from two different tests. Similar to the One_sample_proportion, we are interested in using results from a success/failure test, but we are now interested in whether the difference in results is significant when comparing results between two tests.

Inputs:
- sample_1_trials - number of trials in the first sample
- sample_1_successes - number of successes in the first sample
- sample_2_trials - number of trials in the second sample
- sample_2_successes - number of successes in the second sample
- CI - desired confidence interval. Defaults to 0.95 for 95% CI.

Outputs:
- lower, upper, result - lower and upper are bounds on the difference. If the bounds include 0 then it is a statistically non-significant difference.

In this example, consider that sample 1 and sample 2 are batches of items that two suppliers sent you as part of their contract bidding process. You test everything each supplier sent you and need to know whether the reliability difference between suppliers is significant. At first glance, the reliability for sample 1 is 490/500 = 98%, and for sample 2 is 770/800 = 96.25%. Without considering the confidence intervals, we might be inclined to think that sample 1 is almost 2% better than sample 2. Let's run the two proportion test with the 95% confidence interval.

```python
from reliability.Reliability_testing import two_proportion_test
result = two_proportion_test(sample_1_trials=500, sample_1_successes=490, sample_2_trials=800, sample_2_successes=770)
print(result)
```

```python
('-0.004972498915250083, 0.03549724989152493, 'non-significant')
```

Because the lower and upper bounds on the confidence interval includes 0, we can say with 95% confidence that there is no statistically significant difference between the suppliers based on the results from the batches supplied.
Sample size required for no failures

The function `sample_size_no_failures` is used to determine the minimum sample size required for a test in which no failures are expected, and the desired outcome is the lower bound on the reliability based on the sample size and desired confidence interval.

**Inputs:**

- `reliability` - lower bound on product reliability (between 0 and 1)
- `CI` - confidence interval of result (between 0.5 and 1). Defaults to 0.95 for 95% CI.
- `lifetimes` - if testing the product for multiple lifetimes then more failures are expected so a smaller sample size will be required to demonstrate the desired reliability (assuming no failures). Conversely, if testing for less than one full lifetime then a larger sample size will be required. Default is 1.
- `weibull_shape` - if the weibull shape (beta) of the failure mode is known, specify it here. Otherwise leave the default of 1 for the exponential distribution.

**Outputs:**

- number of items required in the test. This will always be an integer (rounded up).

As an example, consider a scenario in which we want to be sure that a batch of LEDs meets the reliability target for on/off cycles. Testing is for the planned lifetime (1 million cycles) and tested items will have most or all of their lifetime used up during testing so we can’t test everything. How many items from the batch do we need to test to ensure we achieve 99.9% reliability with a 95% confidence interval?

```python
from reliability.Reliability_testing import sample_size_no_failures
result = sample_size_no_failures(reliability=0.999)
print(result)
```

```python
'2995'
```

Based on this result, we need to test 2995 items from the batch and not have a single failure in order to be 95% confident that the reliability of the batch meets or exceeds 99.9%. If we tested for more on/off cycles (lets say 3
million which is 3 lifetimes), then the number of successful results would only need to be 999. In this way, we can design our qualification test based on the desired reliability, CI, and lifetimes that are tested to.

In the event that we suffer a single failure during this test, then we will need to adjust the testing method, either by finishing the testing and calculating the lower bound on reliability using the `one_sample_proportion` test, or by using a `sequential_sampling_chart`.
A sequential sampling chart provides decision boundaries so that a success/failure test may be stopped as soon as there have been enough successes or enough failures to exceed the decision boundary. The decision boundary is calculated based on four parameters. These are the producer’s quality, consumer’s quality, producer’s risk, and consumer’s risk. Consider the producer’s and consumer’s quality to be the desired reliability of the sample, and the producer’s and consumer’s risk to be 1-confidence interval.

Inputs:

- `p1` - producer_quality. The acceptable failure rate for the producer (typical around 0.01)
- `p2` - consumer_quality. The acceptable failure rate for the consumer (typical around 0.1)
- `alpha` - producer_risk. Producer’s CI = 1-alpha (typically 0.05)
- `beta` - consumer_risk. Consumer’s CI = 1-beta (typically 0.1)
- `test_results` - array or list of binary test results. eg. [0,0,0,1] for 3 successes and 1 failure. Default=None
- `show_plot` - True/False. Defaults to True.
- `print_results` - True/False. Defaults to True.
- `max_samples` - the x_lim of the plot. Default=100.

Outputs:

- The sequential sampling chart - A plot of sequential sampling chart with decision boundaries. test_results are only plotted on the chart if provided as an input.
- results - a dataframe of tabulated decision results. These will be printed if print_results=True

In the example below, we use the inputs p1=0.01, p2=0.10, alpha=0.05, beta=0.10. The resulting decision boundaries are plotted, and the test results that we have supplied are also plotted as a stepped line. The plot shows that after our 3rd failure, the test should be stopped as the batch can be rejected. The dataframe of results is also printed by default. In this dataframe, the value of x is used to replace impossible numbers, i.e. we cannot reject 2 failures if we have only conducted 1 inspection. This example is based on an example in the Engineering statistics handbook published online by NIST.
```python
from reliability.Reliability_testing import sequential_samling_chart

test_results = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1]

sequential_samling_chart(p1=0.01,p2=0.10,alpha=0.05,beta=0.10,test_results=test_results)

...  

<table>
<thead>
<tr>
<th>Samples</th>
<th>Failures to accept</th>
<th>Failures to reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>71</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>75</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>76</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>77</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>78</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>79</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>81</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>82</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>83</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>84</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>85</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>86</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
```
(continues on next page)
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Sequential sampling decision boundaries

- **test results**
- **Reject sample**
- **Keep Testing**
- **Accept Sample**

Number of failures from samples tested vs. Number of samples tested.
CHAPTER 42

Reliability test planner

A solver to determine the parameters of a reliability test when given 3 out of the 4 unknowns (lower confidence bound on MTBF, test duration, number of failures, confidence interval).

The underlying assumption is that the failures follow an exponential distribution (ie. failures occur randomly and the hazard rate does not change with age). Using this assumption, the Chi-squared distribution is used to find the lower confidence bound on MTBF for a given test duration, number of failures, and specified confidence interval:

$$MTBF = \frac{2T}{\chi^2\left(\frac{1-CI}{n}, 2F+p\right)}$$

Where:

- MTBF = Mean time between failures (same as mean time to failure (MTTF) when the hazard rate is constant as it is here). Note that this is the lower confidence interval on MTBF. If you want the point estimate then specify CI=0.5 and two_sided=False.
- T = Test duration (this is the total time on test across all units being tested)
- CI = Confidence interval (the confidence interval to be used for the lower bound on the MTBF)
- F = number of failures during the test
- n = adjustment for one sided (n=1) or two sided (n=2) test
- p = adjustment for time terminated (p=2) or failure terminated (p=0) test

The above formula can be rearranged, or solved iteratively to determine any of these parameters when given the other 3. The user must specify any 3 out of the 4 variables (not including one_sided, print_results, or time_terminated) and the remaining variable will be calculated. Note the difference between the one-sided and two-sided confidence intervals which are specified using the input one_sided=True/False described below. A description of the difference between one-sided and two-sided confidence intervals is provided at the end of this page. The formula used defaults to a time_terminated test (where the test was stopped at a particular time which was not related to the number of failures). If the test was stopped after a particular number of failures (such as all items failing) then you must specify time_terminated=False to ensure the correct formula is used.

A similar calculator is available in the reliability analytics toolkit.

Inputs:
• MTBF - mean time between failures. This is the lower confidence bound on the MTBF. Units given in same units as the test_duration.
• number_of_failures - the number of failures recorded (or allowed) to achieve the MTBF. Must be an integer.
• test_duration - the amount of time on test required (or performed) to achieve the MTBF. May also be distance, rounds, fires, cycles, etc. Units given in same units as MTBF.
• CI - the confidence interval at which the lower confidence bound on the MTBF is given. Must be between 0.5 and 1. For example, specify 0.95 for 95% confidence interval.
• print_results - True/False. Default is True.
• one_sided - True/False. Default is True. If set to False, the two sided confidence interval will be returned.
• time_terminated - True/False. Default is True. If set to False, the formula for the failure-terminated test will be used.

Outputs:
• If print_results is True, all the variables will be printed.
• An output object is also returned with the same values as the inputs and the remaining value also calculated. This allows for any of the outputs to be called by name.

In the example below, we have done a time-terminated reliability test for 19520 hours (units are not important here as it may be days, cycles, rounds, etc.). During the test there were 7 failures. We want to know the MTBF that was achieved during the test within an 80% confidence (two-sided). The second example shows how the output may be supressed and the results accessed by name.

```python
#example 1
from reliability.Reliability_testing import reliability_test_planner
reliability_test_planner(test_duration=19520, CI=0.8, number_of_failures=7, one_sided=False)
```

```
Reliability Test Planner results for time-terminated test
Solving for MTBF
Test duration: 19520
MTBF (lower confidence bound): 1658.3248534993454
Number of failures: 7
Confidence interval (2 sided): 0.8
```

```python
#example 2
from reliability.Reliability_testing import reliability_test_planner
output = reliability_test_planner(number_of_failures=6, test_duration=10000, CI=0.8, ...
  print_results=False)
print(output.MTBF)
```

```
1101.8815940201118
```

### 42.1 One-sided vs two-sided confidence interval

The below image illustrates the difference between one-sided and two-sided confidence interval. You can use either the one-sided or two-sided interval when you are seeking only the lower bound, but it is essential to understand that they will give very different results for the same CI. They will give equivalent results if the CI is set appropriately (eg.
90% one-sided is the same as 80% two-sided. If you are unsure which to use, the more conservative approach is to use the two-sided interval. If you want the point estimate, use the one-sided interval with a CI=0.5.

One-sided confidence interval

Two-sided confidence interval

In both cases, the confidence interval is 50% and the total area outside of the confidence interval is also 50%.

The lower bound of the two-sided confidence interval is more conservative (lower) than that of the one-sided confidence interval.

RELIABILITY
A Python library for reliability engineering
This function is an extension of the `reliability_test_planner` which allows users to calculate the required duration for a reliability test to achieve the specified producers and consumers risks. This is done based on the specified MTBF (mean time between failure) required and MTBF design.

This type of determination must be made when organisations looking to test an item are uncertain of how much testing is required, but they know the amount of risk they are willing to accept as well as the MTBF required and the MTBF to which the item has been designed.

**Inputs:**

- `MTBF_required` - the required MTBF that the equipment must demonstrate during the test.
- `MTBF_design` - the design target for the MTBF that the producer aims to achieve.
- `consumer_risk` - the risk the consumer is accepting. This is the probability that a bad product will be accepted as a good product by the consumer.
- `producer_risk` - the risk the producer is accepting. This is the probability that a good product will be rejected as a bad product by the consumer.
- `one_sided` - default is True. The risk is analogous to the confidence interval, and the confidence interval can be one sided or two sided.
- `time_terminated` - default is True. whether the test is time terminated or failure terminated. Typically it will be time terminated if the required test duration is sought.
- `show_plot` - True/False. Default is True. This will create a plot of the risk vs test duration. Use `plt.show()` to show it.
- `print_results` - True/False. Default is True. This will print the results to the console.

**Outputs:**

- `test duration`
  - If `print_results` is True, all the variables will be printed to the console.
  - If `show_plot` is True a plot of producer’s and consumer’s risk Vs test duration will be generated. Use `plt.show()` to display it.
In the example below the consumer requires a vehicle to achieve an MTBF of 2500km and is willing to accept 20% risk that they accept a bad item when they should have rejected it. The producer has designed the vehicle to have an MTBF of 3000km and they are willing to accept 20% risk that the consumer rejects a good item when they should have accepted it. How many kilometres should the reliability test be? Using the function we find the test needs to be 231616 km.

```python
from reliability.Reliability_testing import reliability_test_duration
import matplotlib.pyplot as plt

reliability_test_duration(MTBF_required=2500, MTBF_design=3000, consumer_risk=0.2, producer_risk=0.2)
plt.show()
```

```
Reliability Test Duration Solver for time-terminated test
Required test duration: 231615.79491309822 # Note that this duration is the total time on test and may be split across several vehicles.
Specified consumer's risk: 0.2
Specified producer's risk: 0.2
Specified MTBF required by the consumer: 2500
Specified MTBF designed to by the producer: 3000
```

How does the algorithm work?
The underlying method is as follows:
Step 1) Begin with failures = 1. This will be iterated later.

Step 2) Using the function `Reliability_testing.reliability_test_planner`, we set CI = 1-consumer_risk, MTBF = MTBF_required to solve for the test_duration that is achieved by this test. This is the test duration required if there was 1 failure which would give the specified MTBF required and specified consumer’s risk.

Step 3) We again use the function `Reliability_testing.reliability_test_planner` but this time we set MTBF = MTBF_design and use the test_duration as the output from step 2. Still keeping failures = 1 we are solving for the CI achieved. This is effectively the producer’s risk for the given test_duration and number of failures.

Step 4) The answer is higher than the specified producer’s risk, so we now repeat steps 2 and 3 by increasing the number of failures by 1 each iteration. This is continued until the producer’s risk is below what was specified. We then go back 1 failure since is it standard that the producer’s risk can’t be below what was specified (or the consumer may think the producer is cheating by lowering their risk).

We now have a value for test_duration that will give our required outputs in both equations. We also happen to arrive at the number of failures, though this is not particularly relevant since it is just part of the solution process and the actual number of failures will be determined based on the conduct of the reliability test.

The plot that is produced by `Reliability_testing.reliability_test_duration` displays a scatter plot at each failure. Since the number of failures must be an integer, we get results for reliability test durations that go in steps. The result returned corresponds to the test_duration at the last failure before the producer’s risk dropped below what was specified. Also note that if the consumer’s risk is different from the producer’s risk, the solution for test_duration will not occur near the point on the graph where producer’s risk and consumer’s risk are equal.
The Chi-squared test is a statistical test for goodness of fit to determine whether we can accept or reject the hypothesis that the data is from the specified distribution at the specified level of significance. This method is not a means of comparing distributions (which can be done with AICc, BIC, and AD), but instead allows us to accept or reject a hypothesis that data come from a distribution. Note that the result is sensitive to the bins. For this reason, it is recommended to leave bins as the default value.

The procedure for the test involves comparing the fitted CDF (from a hypothesised distribution) against the empirical CDF (from a cumulative histogram of the data). As with all histograms, the exact shape of the histogram depends on the bins. The difference between the fitted CDF and the empirical CDF is used to find the Chi-squared statistic. The specified level of significance (analogous to confidence level), the number of parameters in the hypothesised distribution, and the number of data points is used to obtain the Chi-squared critical value from the chi-squared distribution. By comparing the chi-squared statistic with the chi-squared critical value, we can determine whether the hypothesis (that the data are from the specified distribution) should be rejected or accepted. The acceptance criteria is when the the chi-squared statistic is below the critical value.

**Inputs:**
- `distribution` - a distribution object created using the reliability.Distributions module
- `data` - an array or list of data that are hypothesised to come from the distribution
- `significance` - This is the complement of confidence. 0.05 significance is the same as 95% confidence. Must be between 0 and 0.5. Default is 0.05.
- `bins` - an array or list of the bin edges from which to group the data OR a string for the bin edge method from numpy. String options are ‘auto’, ‘fd’, ‘doane’, ‘scott’, ‘stone’, ‘rice’, ‘sturges’, or ‘sqrt’. For more information see the numpy documentation on numpy.histogram_bin_edges. Default is ‘auto’.
- `print_results` - if True the results will be printed. Default is True.
- `show_plot` - if True a plot of the distribution and histogram will be shown. Default is True.

**Outputs:**
- `chi2_statistic` - the chi-squared statistic
- `chi2_critical_value` - the chi-squared critical value
• hypothesis - ‘ACCEPT’ or ‘REJECT’. If chisquared_statistic < chisquared_critical_value then we can accept the hypothesis that the data is from the specified distribution

• bin_edges - the bin edges used. If bins is a list or array then bin_edges = bins. If bins is a string then you can find the bin_edges that were calculated using this output.

In the example below we import a dataset called mileage which contains 100 values that appear to be normally distributed. Using the function chi2test we can determine whether we should accept the hypothesis that the data are from a Normal distribution with parameters \( \mu=30011 \) and \( \sigma=10472 \). This example is based on Example 2.31 (page 63) of Reliability Engineering and Risk Analysis (listed in recommended resources).

```python
from reliability.Datasets import mileage
from reliability.Distributions import Normal_Distribution
from reliability.Reliability_testing import chi2test
import numpy as np

data = mileage().failures
dist = Normal_Distribution(mu=30011, sigma=10472)

bins = [0, 13417, 18104, 22791, 27478, 32165, 36852, 41539, 46226, np.inf] #it is not necessary to specify the bins and leaving them unspecified is usually best

chi2test(distribution=dist, data=data, bins=bins)

```

Chi-squared statistic: 3.1294947845652
Chi-squared critical value: 12.591587243743977
At the 0.05 significance level, we can ACCEPT the hypothesis that the data comes from a Normal Distribution \((\mu=30011, \sigma=10472)\)
Chi-squared test

Hypothesised distribution CDF vs cumulative histogram of data

![Graph showing comparison between hypothesised distribution CDF and cumulative histogram of data.]

- **Normal Distribution**: $\mu=30011, \sigma=10472$
- **Cumulative Histogram**: 

**RELIABILITY**

A Python library for reliability engineering
Kolmogorov-Smirnov test

The Kolmogorov-Smirnov test is a statistical test for goodness of fit to determine whether we can accept or reject the hypothesis that the data is from the specified distribution at the specified level of significance. This method is not a means of comparing distributions (which can be done with AICc, BIC, and AD), but instead allows us to accept or reject a hypothesis that data come from a distribution. Unlike the chi-squared test, the Kolmogorov-Smirnov test does not depend on the bins of a histogram, therefore making it a more consistent goodness of fit.

The procedure for the test involves comparing the fitted CDF (from a hypothesised distribution) against the empirical CDF (calculated using a rank order of the data of the form i/n). The difference between the fitted CDF and the empirical CDF is used to find the Kolmogorov-Smirnov statistic. The specified level of significance (analogous to confidence level) and the number of data points is used to obtain the Kolmogorov-Smirnov critical value from the Kolmogorov-Smirnov distribution. By comparing the Kolmogorov-Smirnov statistic with the Kolmogorov-Smirnov critical value, we can determine whether the hypothesis (that the data are from the specified distribution) should be rejected or accepted. The acceptance criteria is when the the Kolmogorov-Smirnov statistic is below the critical value.

Inputs:
- distribution - a distribution object created using the reliability.Distributions module
- data - an array or list of data that are hypothesised to come from the distribution
- significance - This is the complement of confidence. 0.05 significance is the same as 95% confidence. Must be between 0 and 0.5. Default is 0.05.
- print_results - if True the results will be printed. Default is True
- show_plot - if True a plot of the distribution CDF and empirical CDF will be shown. Default is True.

Outputs:
- KS_statistic - the Kolmogorov-Smirnov statistic
- KS_critical_value - the Kolmogorov-Smirnov critical value
- hypothesis - ‘ACCEPT’ or ‘REJECT’. If KS_statistic < KS_critical_value then we can accept the hypothesis that the data is from the specified distribution.
In the example below we import a dataset called mileage which contains 100 values that appear to be normally distributed. Using the function KStest we can determine whether we should accept the hypothesis that the data are from a Normal distribution with parameters mu=30011 and sigma=10472.

```python
from reliability.Datasets import mileage
from reliability.Distributions import Normal_Distribution
from reliability.Reliability_testing import KStest

data = mileage().failures
dist = Normal_Distribution(mu=30011, sigma=10472)
KStest(distribution=dist, data=data)
```

```
Kolmogorov-Smirnov statistic: 0.07162465859560846
Kolmogorov-Smirnov critical value: 0.13402791648569978
At the 0.05 significance level, we can ACCEPT the hypothesis that the data comes from a Normal Distribution (\(\mu=30011, \sigma=10472\))
```

Kolmogorov-Smirnov test
Hypothesised distribution CDF vs empirical CDF of data
46.1 Version: 0.5.5 — Currently unreleased — due for release around mid-February

New features

- All of the standard fitters have been significantly improved with the following features:
  - Least Squares estimation is now available. Previously the fit was solely achieved using MLE. MLE remains the default.
  - For the least squares estimation, users may select RRX, RRY, LS. RRX and RRY are rank regression on X and rank regression on Y respectively. LS will perform both RRX and RRY and use the one with the best log-likelihood.
  - There are 3 optimisers to choose from for all of the standard fitters. These are L-BFGS-B, TNC, powell. Previously there was only an option for some of the fitters and the optimiser was not standardized.
  - Removal of scipy as the method to obtain the initial guess for MLE. With the inclusion of least squares estimation, the MLE method is much faster since it is not reliant on scipy to provide an initial guess (which failed to account for right censored data and often gave a poor guess).

- Addition of a new module for converting data between different formats. The module reliability.Convert_data allows for conversion between FR (failures, right censored), FNRN (failures, number of failures, right censored, number of right censored), and XCN (event time, censoring code, number of events). It also provides a streamlined process for importing data from xlsx files, for exporting data to xlsx files, and for printing the dataset in a dataframe for easy visualisation.

API Changes

- All of the standard fitters now include method and optimizer arguments.
- Fitters.Fit_Weibull_2P, Fitters.Fit_Weibull_3P, Fitters.Fit_Weibull_2P_grouped have had some changes to their input arguments so that they all include method and optimizer. The initial_guess_method option is gone as it has been replaced by least squares estimation.
The entire Stress_strength module has been deprecated. This is because there were (and likely only ever would be) two functions in this module which is not enough to justify a separate module. The two functions have been moved into Other_functions and renamed. Full deprecation will occur in March 2021 (in version 0.5.6), and until then a DeprecationWarning will be printed and the old functions will still work. The renaming is as follows:


Bug Fixes

- fixed a bug in Reliability_testing.reliability_test_duration in which certain inputs resulted in 1 failure and the plot limits caused a crash when left=right limit.

Other

- Utils has 2 new functions (linear_regression and least_squares). These are now used by Fitters to obtain the least squares estimates.
- The format of all the printed fitters outputs has been improved. More detail is provided and the formatting is better.
- Dataframes from fitters are formatted better to retain the index but not display it.
- Text output for sample_size_no_failures
- Text output for one_sample_proportion
- Text output for two_proportion_test
- one_sample_proportion will now return 0 or 1 for the lower and upper reliability estimates instead of NaN in cases when there are all failures or all successes.

46.2 Version: 0.5.4 — Released: 7 November 2020

New features

- Confidence intervals added for Normal, Lognormal, Loglogistic, and Gumbel Distributions. Confidence intervals for the Gamma and Beta Distributions will be part of 0.5.5 in Feb 2021
- Added Gumbel_Distribution to Distributions
- Added Gumbel_Distribution to Other_functions.distribution_explorer
- Added Fit_Gumbel_2P to Fitters
- Added Gumbel_probability_plot to Probability_plotting
- Added Gumbel Distribution to Fitters.Fit_Everything
- Added Gumbel Distribution to Other_functions.similar_distributions
- Added Gumbel Distribution to Stress_strength.Probability_of_failure
- Added Gumbel Distribution to Reliability_testing.chi2test and Reliability_testing.KStest
- Added Loglogistic and Gumbel Distributions to PP_plot_parametric, QQ_plot_parametric, PP_plot_semiparametric, and QQ_plot_semiparametric. Loglogistic should have been added in version 0.5.3 but it was missed.
- Added Loglogistic and Gumbel Distributions to Mixture Model and Competing Risks Model. Loglogistic should have been added in version 0.5.3 but it was missed.
• Fit_Everything now plots everything in order of best fit for all 3 of the plots generated.

• Both the Competing Risks Model and Mixture Model now work for negative xvals when the mixture contains one or more Normal and/or Gumbel Distributions. Previously these were be truncated at 0 which could lead to inaccuracies if the model contained Normal Distributions (or Gumbel Distributions, though Gumbel was not available previously).

API Changes

• Confidence intervals were previously available for the Hazard functions of the Weibull and Exponential distributions. This capability has been removed as it was not useful (just as confidence intervals on the PDF are not useful). Any attempt to use confidence interval related keywords (such as CI and CI_type) on the HF of any distribution will generate an error.

• Fit_Everything now includes an option to exclude distributions.

• Fit_Expon_1P and Fit_Expon_2P are deprecated. These have been replaced by Fit_Exponential_1P and Fit_Exponential_2P. Using the old functions will still work and will issue a DeprecationWarning printed to the console. Full deprecation/removal will occur in March 2021 (in version 0.5.6). The reason for the change is to minimize the use of abbreviated terms. It was originally abbreviated because the word Exponential_Distribution seemed too long, but this is no longer valid with Loglogistic_Distribution being added. Also, scipy’s function for Exponential is “expon” so Fit_Expon_1P initially seemed like an appropriate abbreviation.

• percentiles have been added to all fitters (except Gamma and Beta). This will print a table of percentiles (with bounds on time) to the console. This is similar to the output that Minitab gives when fitting a distribution.

Bug Fixes

• Other_functions.distribution_explorer had a bug caused by a recent update to matplotlib. When a non-existent axis was deleted, the error matplotlib generated was a ValueError and that is now changed to AttributeError which was not being appropriately handled by distribution_explorer.

• All of the standard distributions expected a list or array for their 5 functions (PDF, CDF, SF, HF, CHF). A command like this “dist.SF(1)” would cause an error and should have been entered as dist.SF([1]). This is now fixed such that if the input is not in a list or array then it will no longer produce an error and the output type will be np.float64.

• Within Fit_Everything if only 3 points were entered some of the AIC values would be ‘Insufficient Data’. If the user also specified sort_by=’AIC’ then an error would be raised by pandas trying to sort by strings and numbers. In this case the sort_by method will automatically be changed to BIC.

• The Exponential confidence intervals were invisibe if there were only 2 failures for the fit. This was cause by the upper CI reaching 1 which is effectively infinity on a probability plot. 1’s are now filtered out so the CI will always appear.

Other

• Removed margins in the stress_strength plots so that the xaxis coincides with the plot window.

• Changed layout of Fitters.Fit_Everything probability plot and PP plot to be 4x3 without Beta fitted and 5x3 with Beta fitted. This was necessary to include the Gumbel Distribution in the space that Beta previously used.

• Formatting changes to Fitters.Fit_Everything PP plot so the red line extends to the edges of the plot.

• The histogram plot in Fitters.Fit_Everything now has its legend in the order of the the results, such that the best fitting distribution will appear first in the legend.

• Within Other_functions.similar_distributions there were cases when a 3P distribution was fitted and the optimal gamma was 0 (making it the same as its 2P distribution). A filter has been added so the 3P distribution will only be shown if the gamma parameter is non-zero.

• Improved plots for Stress_strength so the distribution xvals extend beyond the plot xlims. This is only noticable if the plot is moved.
• Adjusted scaling and line colors for all QQ and PP plots to improve the way they are displayed.
• PP_plot_parametric now has labels for quantile lines which are linked to the axes coords, so if the plot is moves / zoomed the labels will follow the plotting window.
• Improved the Mixture Model PDF and HF using the actual formula rather than taking the numerical derivatives of CDF and CHF respectively.
• Fit_Everything can now accept a minimum of 2 failures (previously the minimum was 3) and it will automatically exclude the 3P distributions
• All warnings throughout reliability are now printed in red.
• New Utils function colorprint. This provides a simple API for printing in color, bold, underline and italic.
• Improved input checking for all the fitters. This has been standardised in a Utils function so nothing is missed for each of the fitters.
• Probability_plotting.plot_points previously has a minimum of 2 failures required to plot the points. The minimum is now 1 failure required.

46.3 Version: 0.5.3 — Released: 29 September 2020

New features
• Added Loglogistic_Distribution to Distributions
• Added Fit_Loglogistic_2P and Fit_Loglogistic_3P to Fitters
• Added Loglogistic_probability_plot to Probability_plotting
• Added Fit_Loglogistic_2P and Fit_Loglogistic_3P to Fitters.Fit_Everything
• Added Loglogistic distribution to Other_functions.similar_distributions
• Added Loglogistic distribution to Stress_strength.probability_of_failure
• Added the function Reliability_testing.reliability_test_duration
• Added the function Other_functions.distribution_explorer
• Added Utils.probability_plot_xylims and Utils.probability_plot_xyticks which provide better axes limits and tick labels. These are now incorporated into all probability plots, ALT probability plots and ALT Fitters.
• Added Chi-squared and Kolmogorov-Smirnov goodness of fit tests to Reliability_testing
• Added Anderson-Darling goodness of fit test statistic into all Fitters (It is not approriate to use for ALT_fitters for the entire model). This now allows users to compare distributions goodness of fit using Log-likelihood, AICc. BIC, or AD. Note that the Anderson-Darling test statistic is the default goodness of fit test statistic in Minitab.
• Added Utils.anderson_darling to simplify the process of calculating the AD statistic. It’s a lot of formulas that are best packaged into a function that is called by each of the Fitters.
• Added Datasets.mileage which is a simple dataset with no right censored data.
• Added Nonparametric.RankAdjustment. This method is similar in results to Kaplan-Meier and Nelson-Aalen but very different in the method used.
• Other_functions.make_right_censored_data can now create either singly-censored or multiply-censored data. Previously it only created singly-censored data.

API Changes
• Reliability_testing.reliability_test_planner has an optional argument of two_sided which was set to True as default. This has been changed to one_sided=True, making the default calculation use the one-sided confidence interval and changing the argument name. The reason for this change was to align the function with the approach more commonly used in industry.

• All probability plots had h1 and h2 options for the plotting heuristics. These have been replaced by the argument “a” which is the same as what h1 was. h2 can be calculated from h1 and the length of the dataset so it was redundant. “a” was chosen to align with wikipedia.

• Thanks to the addition of the Nonparametric.RankAdjustment, the functions Probability_plotting.QQ_plot_semiparametric and Probability_plotting.PP_plot_semiparametric now allow for ‘RA’ as the option in their method. Previously the methods were limited to ‘KM’, and ‘NA’ for Kaplan-Meier and Nelson-Aalen.

• Other_functions.make_right_censored_data now has an additional argument of fraction_censored which controls the amount of data to right censor when producing multiply-censored data. There is also a random seed argument added for repeatability.

• All the ALT_fitters were missing loglik as an output. They had loglik2 which is the same as loglik*-2 but this is added for completeness and to match the outputs from Fitters.

Bug Fixes

• Fixed autoscale for cases where the HF is constant so it no longer lies along the yaxis upper limit

• Fit_Everything had a bug in the default xvals for the Beta_Distribution’s histogram which caused an error in some special cases. This is now resolved.

• All the quantile functions in each distribution didn’t accept np.float64 and raised an error. They now accept this data type.

• The AICc and BIC in all the ALT_fitters was slightly wrong due to a small coding error. This is now fixed.

Other

• Fixed the HF and CHF equations for Exponential_Distribution to be actual equations. The is preferred than using the HF = PDF/SF and CHF=-ln(SF) relationships which breakdown when SF=0 at high xvals. This has also been implemented for the loglogistic distribution. Can’t do it for Normal, Lognormal, Gamma, and Beta distributions as these do not have closed form solutions for HF and CHF which don’t involve the SF.

• Changed the Gamma_Distribution and Weibull_Distribution mode to be self.gamma when beta < 1. Previously it was “No mode exists when beta < 1” which is true from a formula perspective but it is clear that the mode is equal to gamma as that’s where the asymptote occurs. The only distribution with “no mode exists…” is the Beta distribution as it can have 2 modes for certain values of alpha and beta.

• Updated Utils.generate_X_array to use 200 points (rather than 100) and allocated more points to the right hand side of the plot (beyond b99). This was because plots were not displaying smoothly enough for distributions with high skewness.

• Changed default plotting upper limit to b9999. Previously it was slightly more and was not a round quantile. Done for simplicity and minimal change will be noticed.

• Changed the layout of the Probability plots and PP plots in Fit_Everything from a 5x2 grid to a 4x3 grid. This made more sense due to the addition of the Loglogistic Distribution which would have made the layout 6x2 which is too long.

• Plotting enhancements to increase the detail in plots using less points (by generating more points where the plots curve and less where the plots are flat). Using 200 instead of 1000 points will make the plots much faster, particularly when multiple distributions are layered. In version 0.5.2 this was just done for the Weibull Distribution but it has now been implemented for all 7 of the standard probability distributions.
• Plotting enhancements to the x and y scale such that the limits are based on the quantiles. This will ensure more relevant detail is shown, particularly for location shifted distributions. In version 0.5.2 this was just done for the done for Weibull Distribution but it has now been implemented for all 7 of the standard probability distributions.

• Within Stress_strength.Probability_of_failure, the integration method has been changed from quad to trapz based on this issue.

• Within Stress_strength the legend text for both plots no longer formats the probability of failure as a percentage and the format is changed to use scientific notation which is much more appropriate for very small failure probabilities.

• Within Stress_strength both functions will issue a warning if stress.mean > strength.mean to indicate that the user may have assigned the distributions in the wrong order.

• The version requirements for all dependancies have been updated to their most recent versions. This is most important for scipy which recently had an update that affects the covariance matrix results.

• Added __version__ to the __init__.py file so that the version number is recorded in the same way as other packages record it.

• Other_functions.histogram has an argument for bins. Previously this accepted the exact bins to be used and if left blank calculated them using the Freedman-Diaconis rule. In addition to accepting the exact bins to use, the bins argument now accepts strings just like matplotlib and numpy, and the default is now ‘auto’. See numpy for more detail on the strings available.

• KaplanMeier and NelsonAalen now consider previous xlim when plotting. This prevents plot limits from being overridden by the most recent plot.

46.4 Version: 0.5.2 — Released: 14 August 2020

New features

• New distributions
  – Mixture_Distribution
  – Competing_Risks_Distribution

• A new fitter for the Weibull competing risks model (Fit_Weibull_CR)

• The output of the Fit_Weibull_Mixture now includes a probability plot instead of a histogram of the PDF and CDF

• The output of the Fit_Weibull_Mixture now prints the confidence interval estimates of the parameters

• Added some datasets for use with the mean cumulative function (MCF_1 and MCF_2).

API Changes

• Within Fitters.Fit_Weibull_mixture the option show_plot has been changed to show_probability_plot to align with all the other fitters.

Bug Fixes

• Fixed the autoscale in Weibull and Exponential distributions that locked autoscaling when confidence intervals were plotted sequentially.

• Automatic removal of zeros for all fitters (except Normal_2P). Previously the zeros were left in the data and resulted in NaNs and crashes. Also added a dedicated error to report input with times below zero.

• Fixed the confidence interval bounds for Kaplan-Meier and Nelson-Aalen CHF plots. Some of the bounds were inf since the CHF = -ln(SF) which will be inf when SF=0.
• MCF_Nonparametric and MCF_Parametric had a bug which caused crashes when the dataset included a system with only one censored time. This has now been fixed.

Other

• Minor clean up of code. Removed unnecessary imports, removed unused variables, etc. Hopefully this will have no noticable effects.

• Within Fitters.Fit_Everything the histogram output has been improved with better formatting and it now uses the Freedman-Diaconis rule for obtaining optimal bin width.

• Fixed Weibull HF and CHF equations to use actual equations and not PDF/SF or -ln(SF) as these result in NaN when SF=0 (an issue at high xvals). These changes are currently only implemented for Weibull_Distribution.

• Improved creation of xvals for PDF,CDF, SF, HF, CHF within the Weibull Distribution. The changes now generate datapoints where there is more detail (between the 0.1% and 99.9% quantiles) such that only 100 datapoints are needed to show more detail than was previously achieved with 1000 datapoints. This is most noticable with Weibull distributions that have high beta values and are significantly location shifted. An example of this is shown in the plot below. These changes are only implemented for Weibull_Distribution but will be extended to all distributions in the very near future.

• Improved autoscaling for the Weibull Distribution plots. For location shifted distributions, this zooms in on the 0.1% to 99.9% quantiles allowing users to see more detail. The HF and CHF ylims are also limited based on the quantiles so that they do not obscure the detail if there is an asymptote to large values or infinity. An example of this is shown in the plot below. These changes are only implemented for Weibull_Distribution but will be extended to all distributions in the very near future.

New features

• More efficient method used within Other_functions.similar_distributions. Results are always consistent and more accurate now.

• Other_functions.histogram. This plots a histogram with optimal bin width, better default formatting, and an option to shade bins white above a threshold.

46.5 Version: 0.5.1 — Released: 08 July 2020
API Changes

- Some of the functions in reliability.Other_functions have been moved into reliability_Utils and reliability.Reliability_testing. The new layout is:
  - Utils ⇒ round_to_decimals, transform_spaced, axes_transforms
  - Other_functions ⇒ similar_distributions, convert_dataframe_to_grouped_lists, crosshairs, make_right_censored_data
  - Reliability_testing ⇒ one_sample_proportion, two_proportion_test, sample_size_no_failures, sequential_sampling_chart, reliability_test_planner

- Within Other_functions.similar_distributions the option ‘monte_carlo_trials’ has been removed as the distribution sampling method is no longer random.

Bug Fixes

- Fixed confidence interval color inheritance for Nonparametric.Kaplan_Meier and Nonparametric.Nelson_Aalen. Previously the color was only inherited if specified rather than left as default.

- The default axes labels for both Stress_strength.Probability_of_failure and Stress_strength.Probability_of_failure_normdist were reversed. They have now been switched to the correct labels.

Other

- Documentation updates to reflect the API changes in Version 0.5.1

46.6 Version: 0.5.0 — Released: 04 July 2020

New features

- Confidence intervals on fitted distributions ==> this has only been implemented for Weibull and Exponential. It is quite difficult and takes considerable time and testing. I will do Normal and Lognormal distributions next, then Gamma and Beta distributions. I hope to finish them all by September 2020.

- Confidence intervals have been disabled in in ALT_probability_plotting and ALT_fitters to avoid cluttering on the plot.

- The probability plot in Fit_Everything now uses the Exponential_probability_plot_Weibull_Scale instead of Exponential_probability_plot. It is much clearer to see the effectiveness of the fit using the Weibull scale.

- Added an option to seed the random_samples functions within the Distributions module. This allows for repeatable results.

- Improvements to rounding of all titles, labels, and stats in Distributions and Probability_plotting using a new function, round_to_decimals.

- Added Other_functions.round_to_decimals which keeps the specified number of decimals after leading zeros. This is useful as round would make very small values appear as 0.

- Minor improvements to color inheritance for probability_plotting.


- Within Stress_strength, the method of obtaining the solution has been changed from monte carlo to integration. Thanks to Thomas Enzinger for providing the formula for this method in response to an Issue that was raised. Using the integration method, accuracy is much higher (1e-11 error now vs 1e-3 error previously) and always consistent, and the speed is significantly improved over the monte carlo method. As noted below in API changes, there is no need to specify the number of monte_carlo_samples and no option to obtain the convergence plot.
• Within Stress_strength, the colors used for shading have been changed to improve the style.

• Probability_plotting.plot_points now includes the option to plot the points for the PDF and HF. These are not very useful as they appear messy due to the discontinuous nature of the function, but they are added for completeness.

• Added Other_functions.transform_spaced. This is similar to np.linspace and np.logspace but it creates an array that is ‘weibull spaced’, ‘normal spaced’, ‘exponential spaced’, ‘beta spaced’, or ‘gamma spaced’. It is used to get data points for the confidence intervals so they are as evenly spaced as possible, particularly on probability paper. This function is likely to be moved into utils.

• Other_functions.make_right_censored_data has been added. This function accepts uncensored data and a threshold, and returns failures and right_censored arrays.

• Added mplcursors to requirements in setup.py as it is needed for the crosshairs function.

• Added crosshairs function to Other_functions. This is a very useful feature that provides interactive crosshairs to the plot using snap-to feature and also adds annotations on click events. Thanks to Antony Lee (the author of mplcursors) for help with getting this to work using his library.

Bug fixes

• Within Stress_strength, there are improvements to the fill_between method as it had errors in some special cases.

• Fixed an Issue in Lognormal_Probability_Plot that occurred for very large numbers (above 1e20)

API Changes

• Within Stress_strength, the output format has changed from an object to a returned value of the probability of failure. This makes it much more simple to access the answer since the object had only one value.

• Within Stress_strength, the method of obtaining the solution has been changed from monte carlo to integration. As a result, there is now no need to specify the number of monte_carlo_samples and no option to obtain the convergence plot.

• Added the options initial_guess_method and optimizer to Fit_Weibull_2P and Fit_Weibull_3P. They were previously only in Fit_Weibull_2P_grouped. It is planned to add these options to all fitters.

• There is now the option CI_type for the Weibull and Exponential fitters. This allows users to chose between confidence bounds on reliability and time. This option will be added to all fitters as the confidence intervals for the other distributions are completed.

Other

• Added tests folder. This is planned to include automated tests.

• Created utils module. I plan to move some utilities into here that are currently inside other modules where users can access them, but users should never need to access them so they just create clutter in the dropdown lists of your IDE.

• Added Reliability_testing module. I plan to move everything related to reliability testing out of Other_functions as there is now enough functions to justify a new module dedicated to reliability testing.

• Documentation updates to reflect the changes in Version 0.5.0

46.7 Version: 0.4.9 — Released: 27 April 2020

New features

• Updates to reliability_test_planner to include option for failure terminated test

Other
• Addition of this Changelog to the documentation
Development roadmap

The following development roadmap is the current task list and implementation plan for the Python reliability library. I welcome the addition of new suggestions, both large and small, as well as help with writing the code if you feel that you have the ability. This roadmap is regularly changing and you may see some things remain on here for a while without progressing, while others may be prioritized at short notice. If you have a suggested feature or you find a bug, please raise an Issue on Github or email me (alpha.reliability@gmail.com) and I will endeavour to either add it rapidly (for simple tasks and bug fixes) or add it to the roadmap. The current release schedule is approximately every 4 to 6 weeks.

High priority (likely to be part of the next release)

- Confidence intervals for Gamma and Beta Distributions. Currently the confidence intervals have been completed for all of the other standard distributions.

- Writing more automated tests. This will speed up the code development processes and help prevent future changes having unidentified effects.

Medium priority (expected in early 2021)

- New Distributions along with their associated fitters and probability plots:
  - Defective_Subpopulation_Distribution. This is for when the CDF does not reach 1 due to a lot of right censored data.
  - Zero_Inflated_Distribution. This is for when the CDF starts above 0 due to a lot of ‘dead on arrival’ products in the dataset.

- Add least squares as a method to obtain the initial guess for all Fitters. Currently this has only been implemented in Weibull and Loglogistic fitters but all the other Fitters use scipy which is slower but more accurate for small datasets.

- Improvement to the optimisation routine within each of the fitters. Weibull has been optimised very well but the others need to same improvements such as updating the initial guess using the output until the BIC converges.

- Improvement to the online documentation for how some of these methods work, including the addition of more formulas, algorithms, and better referencing.
• Merge Fit_Weibull_2P_grouped functionality into Fit_Weibull_2P. Input format will be failures=[], right_censored=[], n_failures=[], n_right_censored=[]. Once this is done for Weibull it will be replicated for all Fitters so they are faster for large datasets with repeated values.

• Add converters between 3 data formats:
  – FR = failures, right_censored
  – FRNN = failures, right_censored, n_failures, n_right_censored
  – XCN = unit, censoring_code, number_of_events (where unit is time, km, cycles, etc.)

Low priority (more of a wish list at this point)

• Warranty Module. This will be a new module of many tools for warranty calculation.

• New reliability growth models. Currently there is only the Duane model. It is planned to include the Crow Extended and AMSAA PM2 models.

• Cox Proportional Hazards Model - This is available in Lifelines.

• Parameter Solver using GUI.

• Speed improvements to fitters by using JAX to replace Autograd. This will be done once the issue preventing JAX from being installed on Windows machines is resolved. It is also reliant on approriate functions for Beta and Gamma being written, which is why autograd-gamma is a dependancy in addition to autograd.
Citing reliability in your work

If reliability contributes to a project that leads to a scientific publication, please acknowledge this contribution by citing the DOI 10.5281/zenodo.3938000.

The following reference is using APA:


If you would like to use another referencing style, the details you may need are:

- **Author**: Matthew Reid
- **Year published**: 2020
- **Title**: Reliability – a Python library for reliability engineering
- **Version**: 0.5.1
- **Platform**: Python
- **Available from**: https://pypi.org/project/reliability/
- **DOI**: 10.5281/zenodo.3938000

Note that the version number is constantly changing so please check PyPI for the current version.

If you have used reliability in any published academic work, I would love to hear about it (alpha.reliability@gmail.com). Depending on the usage, I may provide a link to your work below.

**Links to articles and papers that have used the Python reliability library:**

- Probabilistic characterization of random variables - Phase II - by Javier Alfonso Ochoa Moreno. Note that this article is in Spanish.
- A tutorial for reliability engineers: going from scratch to building Weibull Analyses using Python - by Dr Sarah Lukens.
Chapter 48. Citing reliability in your work
CHAPTER 49

How to request or contribute a new feature

If you would like to see something added or an existing feature changed to make it better, please send me an email (alpha.reliability@gmail.com) and from there we can put together a plan on how to proceed. I greatly appreciate all help, even if it is just pointing out an error in the code or documentation. There are a large number of features currently identified for inclusion into this library so if you would like to contribute something but don’t know what to help with, please email me and we can discuss the functions that are currently planned for development.

If you are requesting something new, it is helpful to include as much detail as possible, such as:

- a detailed explanation of what the new feature is and why you think it would be useful
- example Python code
- any references such as academic papers or textbooks which describe the necessary steps
- screenshots from other programs (such as Minitab, JMP Pro, Reliasoft) which show the feature in use

Remember to upgrade reliability to the newest version and double check that the requested feature is not in there, as the codebase is constantly evolving.
CHAPTER 50

How to get help

Questions on statistics or mathematics
If you have a question about a statistical or mathematical process that reliability performs, please consult Google and Wikipedia to see if you can find an existing explanation. If you still need to ask someone then I recommend asking your question on Stack Exchange. If you still can’t get an answer on there, you’re welcome to email me directly (alpha.reliability@gmail.com) and I will try my best to help you.

Questions on using the Python reliability library
If you have a question about how to do something using reliability or how to use one of the features within reliability then you should firstly consult the documentation and the help files within Python. An example of how to access one of the help files is provided below. If the documentation and help files still do not answer your question then you’re welcome to email me directly (alpha.reliability@gmail.com) and I will work to improve the documentation if it is unclear.

from reliability import Fitters
print(help(Fitters))
How to donate to the project

The Python reliability library is free, open source, and it always will be. It aims to provide students and professionals alike with a set of tools that you would otherwise only find in commercial software. Most commercial software is prohibitively expensive, especially for individuals, so I aim to bring these features to you for free by continuing to develop this library.

Developing and maintaining this library is all done in my free time and can be quite a time consuming process. If you would like to donate as a way of showing your appreciation, you can send whatever amount you feel is appropriate using paypal.me. If you don’t want to say thanks with money that is fine too, as I always appreciate getting emails from people all over the world telling me that they find the library useful.
The Python reliability library was written by Matthew Reid. Matthew holds a Masters of Reliability Engineering from the University of Maryland, a Masters of Project Management from the University of New South Wales, and a Bachelor of Aeronautical Engineering from the University of New South Wales. Matthew lives in Melbourne, Australia and currently works as a reliability engineer on a variety of acquisition and sustainment projects for land materiel. If you would like to contact Matthew, you can send a message via LinkedIn.

The Python reliability library was written because there were no dedicated reliability engineering libraries for Python, and Matthew found himself needing to use scipy.stats, lifelines, Minitab, MATLAB, JMP Pro, and his own Python scripts, for a variety of common reliability engineering tasks that were not available in one place. This library is intended to make reliability engineering more accessible to the world, particularly to those individuals and small businesses who find the high cost of proprietary software to be a barrier to entry into conducting reliability engineering analysis.

This is Matthew’s first Python library on the Python Package Index and is currently in active development. In accordance with the LGPLv3 license, every effort has been made to ensure the software is free of errors, however, no guarantees or warranties are provided in any form. Feedback on the Python reliability library is most welcome. If you find an error, have a suggestion, would like to request something to be included, or would like to contribute something, please send it through by email (alpha.reliability@gmail.com).
RELIABILITY
A Python library for reliability engineering
During the process of writing reliability there have been many problems that I was unable to solve alone. I would like to thank the following people who provided help and feedback on problems with the code and with the reliability concepts:

- **Cameron Davidson-Pilon** for help with getting autograd to work to fit censored data and for writing autograd-gamma which makes it possible to fit the gamma and beta distributions. Also for providing help with obtaining the Fisher Information Matrix so that the confidence intervals for parameters could be estimated.

- **Dr. Vasiliy Krivtsov** for providing feedback on PP and QQ plots, for further explaining optimal replacement time equations, and for guidance in developing the Competing risk model. Dr. Krivtsov teaches “Collection and analysis of Reliability Data (ENRE640)” at the University of Maryland.

- **Dr. Mohammad Modarres** for help with PoF, ALT_fitters, and ALT_probability_plotting. Dr. Modarres teaches several reliability engineering subjects at the University of Maryland and has authored several of the textbooks listed under recommended resources.

- The Stack Overflow user ImportanceOfBeingErnest for this answer that was necessary to get the probability plotting functions working correctly for Gamma and Beta distributions.

- Antony Lee for help in adapting parts of his mplcursors library into the crosshairs function in reliability.Other_functions.crosshairs

- **Thomas Enzinger** for help in improving the method of finding the area in stress-strength interference between any two distributions. Previously this was done using a monte-carlo method, but Thomas’ method is much more accurate and always consistent. This is incorporated in Version 0.5.0.

- **Karthick Mani** for help implementing the Loglogistic and Gumbel Distributions including implementation of these distributions in Fitters and Probability_plotting.