HIGH THROUGHPUT NON-PARAMETRIC PROBABILITY DENSITY ESTIMATION VIA NOVEL MULTITHREADED STITCHING METHOD

by

Zach D. Merino

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Applied Physics

Charlotte

2019

Approved by:

_____
Dr. Donald Jacobs

_____
Dr. Michael Grabchak

_____
Dr. Rosario Porras-Aguilar

ABSTRACT

ZACH D. MERINO. High throughput non-parametric probability density estimation via novel multithreaded stitching method. (Under the direction of DR. DONALD JACOBS)

A method of univariant probability density function (pdf) estimation is developed for big data applications. The method employs the use of a non-parametric maximum entropy estimator (NMEM) for a data driven multithreaded probability density estimation algorithm, which has been termed the stitching estimator (SE). The NMEM has previously shown to be a robust pdf estimator for high throughput applications, which has made it the ideal choice for the underlying estimator in the SE's algorithm. This work divides the estimation problem into many smaller estimation problems; termed blocks. The sample is partitioned into blocks by an optimized branching tree algorithm which has been developed to maximize the uniformity for the density of the data in every block. The algorithm finds pdf estimates for blocks using the NMEM then the estimates per block are combined through a stitching procedure that uses a weighted average which utilizes the cumulative probability density functions (cdf) for each pair of adjacent blocks. Further improvements are obtained by implementing a sub-sampling approach that generates sub-samples from the original sample without replacement. The pdfs from each sub-sample are then averaged to give a final estimate. The SE has been extensively benchmarked against a large set of diverse distributions for sample sizes ranging from of 2^9 up to 2^{20} and 1000 trials per sample size. The quality of the estimates are quantified using scaled quantile residual (SQR) plots, which is a sample size invariant metric that is consistent with the Anderson-Darling test. The set of test distributions range from easy single mode distributions to extremely difficult exotic distributions. In all cases tested the SE yields excellent estimates with no need for a priori knowledge of the structure of the data.

ACKNOWLEDGEMENTS

There have been many who have earned my deep gratitude throughout my journey in graduate school. Most notably, my research advisor Dr. Donald Jacobs for all of the insight and guidance he has provided me with over my academic career and to Dr. Jenny Farmer for her invaluable help learning about the NMEM; for this I am truly grateful. I must also thank my research committee for the informative discussions that we've shared over my time as a research member in the BioMolecular Research Group. I have gleaned much from these discussions regarding both research and career related topics. The graduate student and researcher I am today has largely be thanks to the many engaging trials and tribulations myself and fellow graduate students have over come together and I will always relish the memories of these experiences, as well as, the people who I am proud to call my friends. Lastly, I would like the thank two of the most outstanding people I know; my parents. Without their support and unconditional love I would not be the person I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

N_{blocks}	number of data points in a block
block	subset of data points from the sample that fall within an upper and lower boundary
cdf	cumulative density function
KDE	kernel density estimation
LL	quasi-log-likelihood
ME	mean error
N	number of data points in sample
NMEM	non-parametric maximum entropy method
OBT	optimized branching tree
pdf	probability density function
QQ	quantile-quantile
SE	stitching estimator
SQR	scaled quantile residuals
SURD	sample uniform random data
SURD coverage	a value for the cdf of the LL function
target coverage	target SURD coverage for $\hat{f}(x)$ from NMEM

CHAPTER 1: INTRODUCTION

Being able to quickly and accurately estimate a pdf for univariate data is of fundamental importance to many areas both within and outside of physics. An example of this importance is in the field of Bioinformatics in which it is crucial to have methods for pdf estimation that can provide high throughput for large amounts of data. Another example which is outside the field of physics, such as in finance, it is desired to have an estimate for the pdf of a sample of univariant data that is accurate about the most likely to occur events to ensure for lower risk investments. Among the example applications given there are numerous other areas where pdf estimation is of great importance, such as, damage detection in engineering [2], isotope analysis in archaeology [3], and econometric data analysis in economics [4]. The applications for pdf estimators exemplified above assure that the use of a pdf estimator is certainly needed in many areas of analysis, but a continuing issue is that the method of estimation is left to the user. If the annalist has a wide range of experience, the form of the pdf estimate can be selected to obtain an accurate representation of the data. However, this would be an unreasonable approach for large numbers of datasets with no knowledge a priori of the general trends in the data. The annalist must still choose an estimator, but without objective criteria this can introduce subjective bias into the estimated pdfs. There are many methods for pdf estimation all of which contain their own advantages and disadvantages; some of which will be reviewed below.

1.1 Histograms

The first and one of the most widely used is the histogram method. This approach is the easiest to implement method for defining the probability distribution for a

given set of data. Simply: define a bin width, set a bin origin, partition the span of the data using the defined bin width, then generate a bar graph from the number of data points falling in a given bin [5]. Unfortunately, the choice of bin size and bin location can dramatically change the general shape of the probability distribution, which can lead to the loss of crucial information or the detection of erroneous features from random fluctuations in the data. Another downside to using histograms to generate probability distributions is that they do not provide a continuous model for the probability density function, which may be desired to accurately generate other statistical information. The estimate of a pdf, $\hat{f}(x)$, using the histogram method is formally defined by equation 1.1.

$$\hat{f}(x|\widetilde{\boldsymbol{x}}, \boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{w_i}I(x - \widetilde{x}_i, w_i) \tag{1.1}$$

Where $\widetilde{x}_i \equiv$ *bin center*, $w_i \equiv$ *bin width*, and $I(x - \widetilde{x}_i, w_i)$ is normally referred to as the indicator function and is defined by equation 1.2 [6].

$$I(x - \widetilde{x}_i, w_i) = \begin{cases} 1 & x \in [-\frac{w}{2}, \frac{w}{2}) \\ 0 & otherwise \end{cases} \tag{1.2}$$

The definition for $\hat{f}(x)$ from the histogram method in equation 1.1 has been left general. When w_i and $\widetilde{x}_{i+1} - \widetilde{x}_i$ is constant for all i then this is the usually referred to probability density estimate using the histogram method. On the other hand if w_i and $\widetilde{x}_{i+1} - \widetilde{x}_i$ are allowed to vary based on data driven criterion then the histogram method is referred to as an adaptive histogram method. Adaptive histogram methods improve $\hat{f}(x)$ over the usual histogram approach, however, information about the data's location within a bin is still lost, creating a discontinuous model.

1.2 Kernel density estimation

Another ubiquitous method for univariant data pdf estimation is that of kernel density estimation (KDE). The method consists of choosing a kernel basis function, $K(x)$, with a specific set of parameters. Once the kernel and parameters are selected a linear superposition of the N kernels is constructed to estimate the data. Similar to the bin width size from the histogram method there is a choice of bandwidth size for the kernel function. The most general form of KDE is shown in equation 1.3, where β is the shift parameter and h is the bandwidth parameter. Equation 1.3 describes adaptive bandwidth KDE when h is allowed to be determine from data driven criterion [7] or standard KDE when h is constant.

$$\hat{f}(x|\boldsymbol{\beta}, \boldsymbol{h}) = \frac{1}{N}\sum_{i=1}^{N} K(x \mid \beta_i, h_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{h_i} K\Big(\frac{x-\beta_i}{h_i}\Big) \tag{1.3}$$

As an example, a common choice of kernel is that of a Gaussian distribution with the parameters being the mean, μ, and standard deviation, σ. Therefore the pdf estimate, $\hat{f}(x)$, of the true pdf, $f(x)$, is calculated using equation 1.4.

$$\hat{f}(x|\boldsymbol{\sigma}, \boldsymbol{\mu}) = \frac{1}{N}\sum_{i=1}^{N} K(x \mid \sigma_i, \mu_i) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{\sigma_i\sqrt{2\pi}} e^{-\frac{(x-\mu_i)^2}{2{\sigma_i}^2}} \tag{1.4}$$

Using KDE over histograms still has its challenges that the annalist must face. The series of issues that must be considered when using KDE are: 1) Which kernel is the most suited for the dataset? 2) What size basis, N, is necessary to discriminate relevant trends in the data from superfluous noise? 3) What specific values for the parameters fit the data best? Both 1 and 2 are commonly left to the annalist's interpretation of the dataset, however, methods are available to determine the parameters through data driven processes; these methods are termed non-parametric KDE. An example of such a non-parametric method is that of the histogram and another is to

take $\hat{f}(x)$, as in equation 1.3 and minimize the mean square error (MSE), or L_2 error, for the parameters $\boldsymbol{\beta}, \boldsymbol{h}$ using a data driven bandwidth selector method defined in [8]; other L_p errors could be used.

1.3 Alternative pdf estimation methods

More sophisticated non-parametric probability estimation algorithms have been developed over the past few decades that have tried to go beyond the different variants of KDE and adaptive KDE. Many proposed estimators utilize physical concepts that are very familiar to physicist and mathematicians, such as, utilizing the equations that describe diffusion to aid in KDE [9, 10]. Other proposed estimators utilize methods that have been familiar to statisticians and computer scientist since the 1960s and have been developed under statistical learning theory. Examples of these approaches are the use of a support vector machine or an artificial neural network [11, 12]. These methods have shown improvement over KDE but have their respective relative drawbacks. Either, the methods have increased sophistication in the theory and algorithms or the need for a priori knowledge of the expected random variable's expected outcome i.e. training datasets to teach the algorithm. Therefore, it is advantageous to explore methods of probability density estimation that still can be robust, as well as, relatively simple.

1.4 Estimators developed in the BMPG group

As eluded to above, there are methods of pdf estimation that contain a set of parameters, basis size, that are used to construct a parameter space where with the aid of a specific metric the parameters are estimated. However, in many cases the optimal basis size is unknown or perhaps can never be known, therefore, it is advantageous to use a non-parametric estimation method that can adapt the basis size for a specific criterion. For this reason, the research in the conducted BMPG (BioMolecular Physics Group) utilized an estimator developed by Dr. Jenny Farmer

and Dr. Donald Jacobs for determining $\hat{f}(x)$. This approach has been termed the non-parametric maximum entropy method (NMEM), which uses a method of funnel diffusion defined in [13, 1] to estimate the parameters of $\hat{f}(x)$ while exploring and adapting the parameter space.

The NMEM was intended to be used for high data throughput. This method was found to be robust and computationally efficient for a large number of common distributions [1]. However, as is the case when attempting to create a general estimator for a broad range of datasets, the method has limitations. The NMEM estimator becomes computationally inefficient and/or less accurate with divergent and heavy tailed distributions. The samples from these difficult distributions are far from being uniform and the NMEM makes a decision to truncate a series expansion to save on computational cost. The truncation of the series expansion leads to the introduction of systematic errors in the pdf estimates. To improve the estimate of pdfs for these difficult cases Dr. Jacobs developed an estimation method termed the stitching estimator (SE), which utilizes the NMEM estimator to find $\hat{f}(x)$. The stitching estimator is given this name because the range of the sample is partitioned into blocks that the NMEM estimator is applied too. The reduced sample sizes of the blocks create lower variance in the density of the data per block, which allows NMEM to be both fast and accurate. The partitioning of the sample into blocks is akin to how a histogram partitions the data sample into bins. Once the pdf estimates per block are determined, they are then stitched together using a weighted average technique.

The stitching method for determining $\hat{f}(x)$ has empirically shown a significant improvement over the NMEM method for many divergent and heavy tailed distributions. Although noticeable improvement has been observed there are still characteristics of the estimator that must be refined. 1) As the sample size for a given distribution increase the tendency for over fitting arises. 2) For divergent distributions and heavy tailed distributions (ex. stable distribution) $\hat{f}(x)$ for the blocks about the tails or

divergent areas of the distribution retain a non-negligible probability to fail by either taking too long to estimate or being be poorly estimated. 3) The determination of the block lengths or number of data points per block can significantly vary the accuracy of $\hat{f}(x)$ to $f(x)$. Problem 2 turns out to be heavily dependent on problem 3 as will be discussed later and has motivated the focus of this thesis to solve problem 3.

CHAPTER 2: STITCH ESTIMATOR BASICS

2.1 Define blocks

The first procedure for the SE method is akin to a histogram approach, where a bin size must be defined to partition the sample of the random variable. This procedure for the SE method partitions the sample into what are defined as blocks. Once the total number of k blocks has been determined the NMEM estimator is utilized to come up with an estimate, $\hat{f}_k(x)$, for all blocks. The method of defining blocks is implemented to divide and concur difficult distribution types, which is a common place tactic when attempting to find solutions to difficult problems. Figure 2.1 shows an example of how the sample data is partitioned into blocks where estimates $\hat{f}_k(x)$ for each block are obtained and then stitched together. Figure 2.1 also displays the length and number of data points for the each respective block. It is advantageous to divide the pdf estimate problem into blocks, because this allows for easy implementation of a multithreaded algorithm. Being able to create a multithreaded algorithm with the NMEM estimator as the backbone of the script enables the algorithm to be just as efficient as NMEM estimator while significantly increasing the quality of the pdf estimates for a given sample size.

There are several approaches that can be implemented to partition the data, but the two broad categories are classified as a fixed or adaptive methods. The methods for determining the blocks sizes that were initially implemented in the algorithm are discussed in detail below.

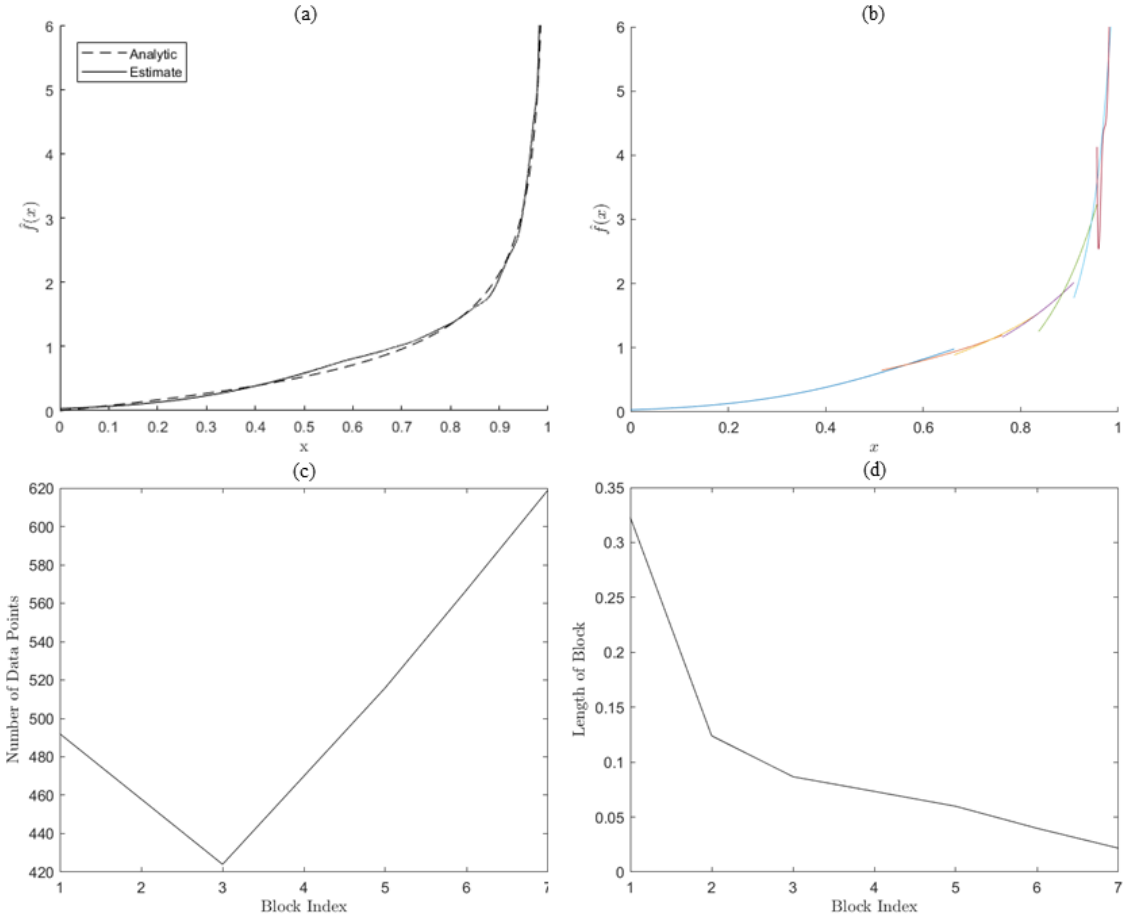

Figure 2.1: (a) Total $\hat{f}(x)$ for data generated from a Beta distribution ($a = 2$, $b = 0.5$) that has been stitched together from $\hat{f}_k(x)$ for each block. (b) Displaying each $\hat{f}_k(x)$ prior to stitching adjacent estimates to produce $\hat{f}(x)$. (c) Number of data points in each block. (d) Length of each block.

2.1.1 Fixed number of blocks

An easily implemented method for determining the block sizes was to require the total number of blocks to be fixed and this was achieved by requiring the number of sample points per block to be proportional to the sample size, as shown in equation 2.1, where c is the percentage coefficient. The number of sample points per block was fixed to be a specific percentage of the sample size. For this method the percentage of points to fall within each block is currently user specified with a common percentage

being 20% of the sample size.

$$N_{block} = \lceil cN \rceil \tag{2.1}$$

Figure 2.2 displays two methods for generating blocks with approximately the same number of data points per block. The first approach, show in figure 2.2(a), creates blocks of fixed size by picking a near central point in the samples range then creates blocks N_{blocks} in size. This processes of creating blocks of size N_{blocks} continues outward toward the edges of the sample's range until blocks N_{blocks} in size can no longer be created. This method leads to blocks with the same number of data points until

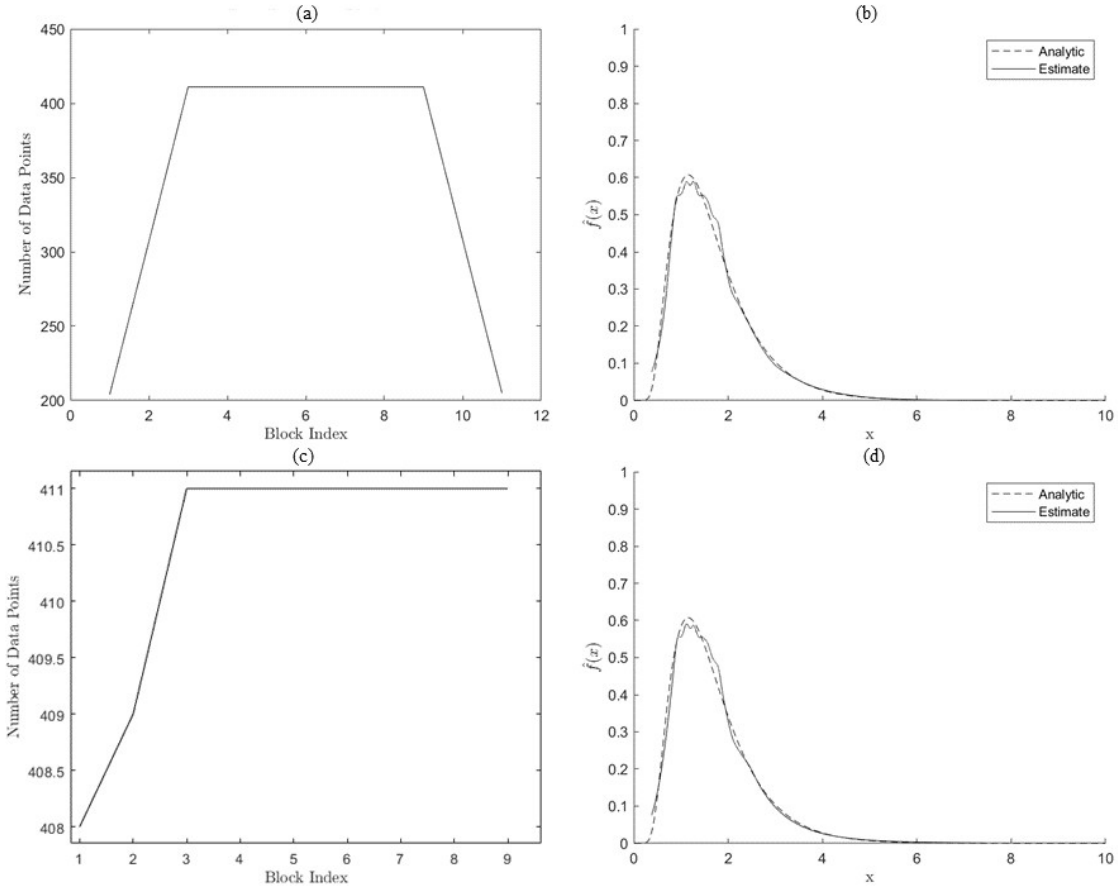

Figure 2.2: $\hat{f}(x)$ for Beta distribution ($\beta = 1.5$, $\gamma = 0.5$) (a) Blocks created from center outward. (b) $\hat{f}(x)$ for blocks created from center outward. (c) Blocks created from left to right. (d) $\hat{f}(x)$ for blocks created from left to right.

the very edges. The second approach, shown in figure 2.2(c), creates blocks N_{block} in size starting from the right most position in the sample's range. Similar to the previous method, this process continues until blocks N_{blocks} in size can no longer be created. The method from creating blocks of fixed size in 2.2(a) is a more elegant approach over the method in 2.2(c) due to the symmetric end block sizes, which eliminates any potential for a bias in $\hat{f}(x)$ due to block partitions. However, in application the choice of method for partitioning the sample into blocks has little effect on the overall estimate $\hat{f}(x)$ for the majority of distribution types.

2.1.2 M-slope

The magic slope (M-slope) method is an adaptive algorithm that was initially implemented to improve the SE which determines the block sizes by updating a given block's size based on a comparison of the M-slope parameter to the ratio of the length of the block to the smallest distance between adjacent points within the block. Equation 2.2 shows the relationship that is used for the comparison of $\alpha^{(k)}$ with M-slope for each block, where $\Delta x_i \equiv x_{i+1} - x_i$ is the difference between pairs of data points in the block and the set of Δx's for the kth block is defined as $\{\Delta x\}^{(k)}$.

$$\alpha^{(k)} \equiv \frac{X^{(k)}_{Right\ \ Block\ \ Boundary} - X^{(k)}_{Left\ \ Block\ \ Boundary}}{min\{\Delta x\}^{(k)}} \tag{2.2}$$

Before the minimum distance between data points in the kth block is computed a few checks are made to ensure that the data is continuous, that there will not be a difference between data points that would numerically result in a difference of zero, and that there are not random pairs of data points exceedingly close due to random sampling. If any of the checks fail a fuzzing procedure is carried out that adds random noise to the dataset. M-slope is optimized through iterative methods to make sure the block sizes are neither too large or small for the block sample under consideration. The conditions for determining whether or not a block is too large or small, as well

as, the initial starting M-slope value has been chosen through empirical investigation. The ratio $\alpha^{(k)}$ is related the expected slope variation of the estimate from a block's sample. Therefore M-slope is optimized to ensure estimates obtained for each block have minimal slope variation across the block. The benefit of this method is that the acceptable M-slope conditions already are well suited for general types of datasets and may be further improved if necessary to increase the scope of distribution types that the SE can readily handle.

2.2 Create secondary blocks

Once the initial bock sizes have been established a secondary set of blocks is created to further improve $\hat{f}(x)$ and may be interpreted as creating block "layers" very similar to how bricks are laid as seen in figure 2.3. The second level of blocks created overlaps the boundaries of the first block level with the second level of block's boundaries being located at the mean position for the first level of blocks. Establishing the second level of blocks helps ensure that a more accurate prediction of $\hat{f}(x)$ is constructed from the stitching of $\hat{f}_k(x)\ \forall\ k$ by reducing the chance for individual blocks from over fitting to random fluctuations within a block's sample.

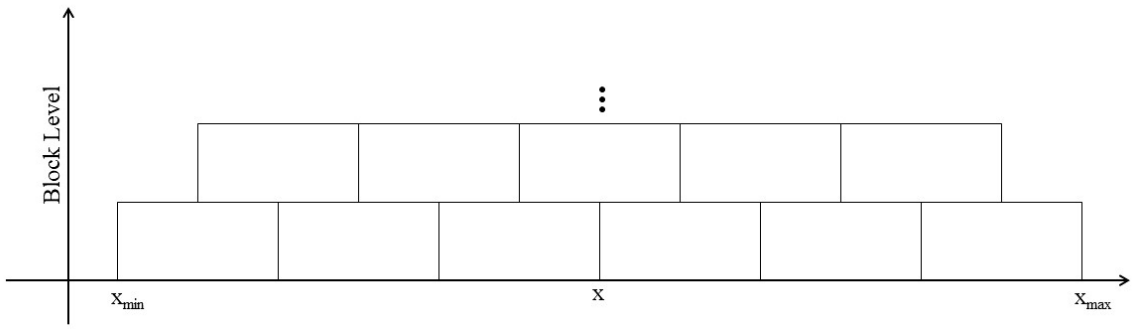

Figure 2.3: Visual interpretation for creating block layers for a sample of data.

2.3 Find $\hat{f}_k(x)$ per block via NMEM estimator

After all of the block sizes have been established the estimate $\hat{f}(x)$ is computed using the NMEM estimator created by Dr. Farmer and Dr. Jacobs. The C++

code that performs NMEM estimation on a set of data has been developed to have a number of useful features that a user may control. One which will be mentioned is the return of a total score that is related to the quality of the estimate from a scoring function defined in [1, 14]. This score has the potential to be used as an acceptance criterion to decide whether $\hat{f}_k(x)$ should be excepted, rejected, or recalculated. The NMEM C++ program is quite fast in its default form and may be further sped up by some degree from modification of user controlled input options. It is advantageous to have a powerful estimation method that is also fast to apply to each block, because the algorithm created in MATLAB for the SE is created to be multithreaded. Thus, the SE can be nearly as fast as the NMEM estimator while yielding consistent pdf estimates for common distributions and is able provide pdf estimates for exotic distributions that the NMEM estimator (any many other estimators) have difficulty with. Once the block sizes are defined the NMEM estimator will run for all of the blocks in parallel, which can significantly speed up the time of computation depending on the number of available processors.

2.3.1 Scoring function

The scoring function is a quasi-log-likelihood (LL) function for typical fluctuations in sample uniform random data (SURD) and is rigorously defined defined in [1, 14]. The reason for the use of the qualifier "quasi" is because there are correlations built into the function due sorting the sample.

maximizing the LL function leads to $\hat{f}(x)$ over fitting to the sample, therefore, a target score is set that is in the range of expected $\hat{f}(x)$ outcomes but will not yield an overfitted estimate. Figure 2.4 displays the pdf for the LL function, which shows the typical fluctuations expected in SURD. The quality of the $\hat{f}(x)$ is related to how well it exhibits SURD. Also, a point along the cdf of the LL function is defined as the SURD coverage and relates how much of $\hat{f}(x)$ exhibits SURD. For example, if $\hat{f}(x)$ received a score of -0.37 from the LL function, this would correspond to $\hat{f}(x)$

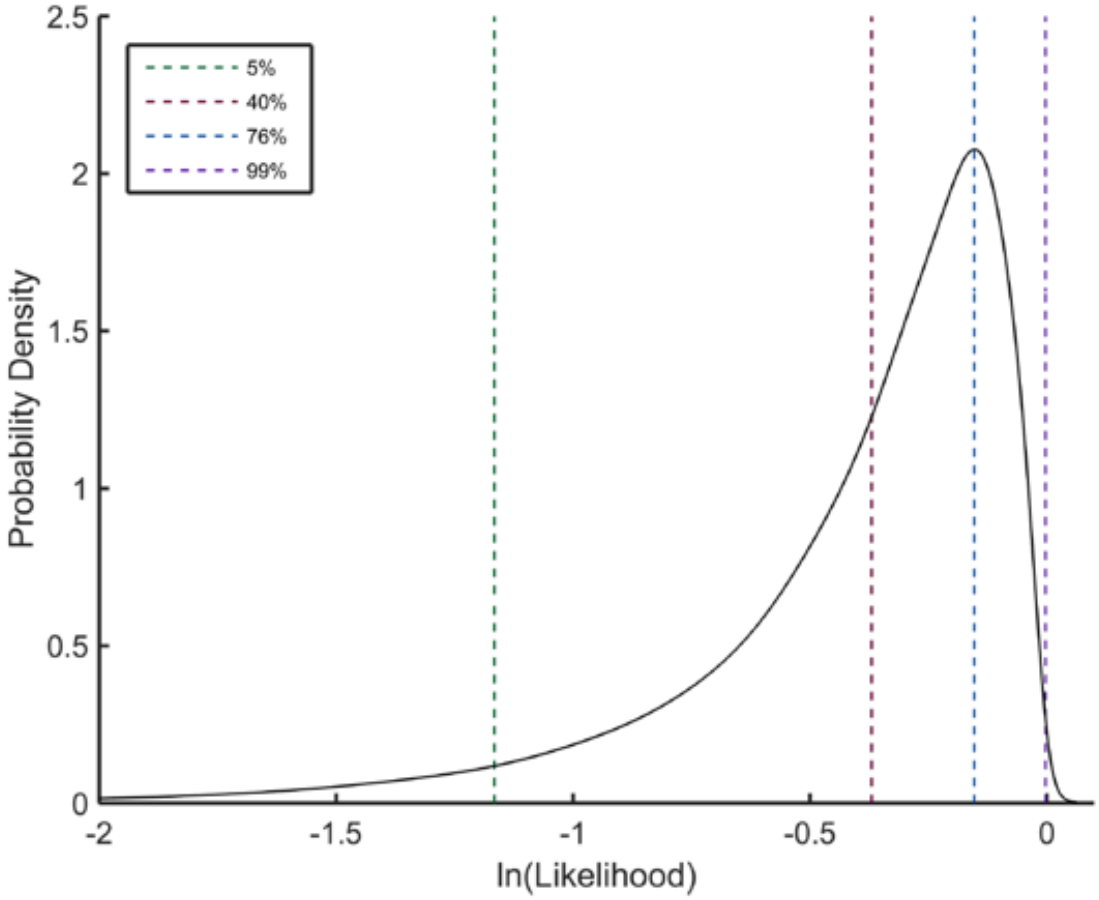

Figure 2.4: Pdf for Log-Likelihood function. This figure shows several threshold values that have a chance of occuring and has come from [1].

exhibiting approximately 40% SURD coverage.

2.3.2 Scaled quantile residual

For a given sample of size N the data points $x^{(i)}$ for $i \in [1, N]$ are sorted and transformed to $U^{(i)}$ using the cdf of $\hat{f}(x)$. Therefore, $U^{(i)}$ has a range of [0,1] and $U^{(i)} < U^{(i+1)} \; \forall \; i \; \in \; [1, N]$. From order statistics the pdf of finding $U^{(i)}$ at position u is,

$$p_i(u|N) = \frac{N!(1-u)^{N-i}u^{i-1}}{(N-i)!(i-1)!} \tag{2.3}$$

And from this the mean and standard deviation are given by,

$$\mu_i = \frac{i}{N+1} \qquad \sigma_i = \sqrt{\frac{\mu_i(1-\mu_i)}{N+2}} \tag{2.4}$$

Using μ the residuals for the transformed estimate $\hat{f}(x)$ are defined as $U^{(i)} - \mu$ and can be made sample size invariant by multiplying by $\sqrt{N+2}$. Thus, the scaled quantile residuals, Δ_i, is defined by,

$$\Delta_i = \sqrt{N+2}(U^{(i)} - \mu_i) \tag{2.5}$$

Using Δ_i, SQR plots can be generated to evaluate the quality of the estimates $\hat{f}(x)$. To extend the ability for SQR plots to represent the quality of $\hat{f}(x)$ a 99% confidence interval is plotted along with Δ_i using $\pm 3.4\sqrt{N+1}\sigma_i$.

Examples of quantile-quantile (QQ) plots compared to SQR plots as the sample size increases are displayed in figure 2.5. For small sample sizes both QQ and SQR plots exhibit reasonable statistical resolution, however, as the sample size increases from $N = 256$ in figure 2.5(a) up to $N = 32,768$ in figure 2.5(d) the resolution of the QQ plots diminishes, while the SQR plots maintain the same resolution. The reason for this occurrence is due to the estimated pdf improving as the sample size increases, therefore the residuals between the $\hat{f}(x)$ and $f(x)$ will in general decreases in size. The use of the phrase "in general" is to imply that a reasonable $\hat{f}(x)$ for $f(x)$ must be found, otherwise other features in the QQ plot will be present.

The SQR plots shown in figure 2.5 all are enveloped by a grey oval which represents the boundaries for a 99% confidence interval that the sample comes from $\hat{f}(x)$. If the SQR plotted line starts to fall outside of the 99% confidence interval then $\hat{f}(x)$ could still be a reasonable estimate for the sample, but if much of the SQR plotted line falls outside of the 99% confidence interval $\hat{f}(x)$ should rejected or taken with caution.

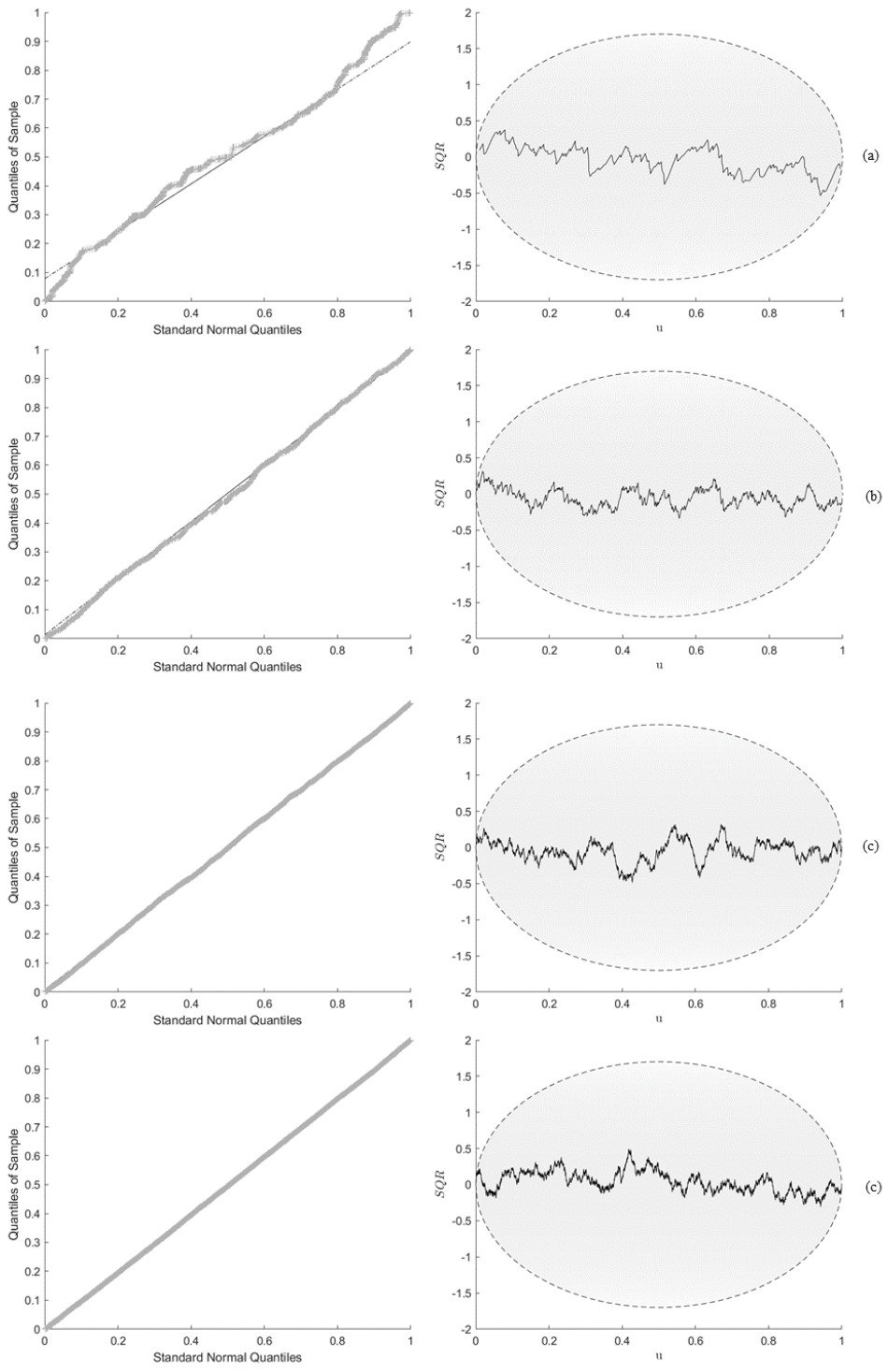

Figure 2.5: QQ and SQR plots for a standard normal distribution with sample sizes consisting of (a) 256 (b) 1,024 (c) 8,192 (d) 32,768.

2.4 Stitch together $\hat{f}_k(x)$ and $\hat{f}_{k+1}(x)$

The pdf estimates for each individual block are stitched together using a weighted average method defined by equations 2.6-2.12 and an example of the method is shown in section 2.6.4 figure 2.11. For each block the cumulative distribution function, $\hat{F}(x)$, is estimated and the pdf estimate of the overlap region, $\hat{f}_s(x)$, for the two blocks is calculated by using the cdf for each block to define weights for that block's pdfs. The variables u and v shown in equations 2.6 and 2.7 are created to ensure that resulting weights, w_k and w_{k+1}, will take values in the range [0,1]. The variables a and b defined in equations 2.8 and 2.9 are used to effect the rate of transition from the left to the right block's pdf estimate. The rate of transition maybe affected by changing the exponent to equations 2.8 and 2.9, however, an optimal exponent of 2 has been heuristically determined. The final weights, w_k and w_{k+1}, are then defined by equations 2.10 and 2.11 such that w_k goes from 1 to 0 and w_{k+1} goes from 0 to 1 as x increases.

$$u = \frac{\hat{F}_k(x) - min(\hat{F}_k(x))}{max(\hat{F}_k(x)) - min(\hat{F}_k(x)))} \tag{2.6}$$

$$v = \frac{\hat{F}_{k+1}(x) - min(\hat{F}_{k+1}(x))}{max(\hat{F}_{k+1}(x)) - min(\hat{F}_{k+1}(x)))} \tag{2.7}$$

$$a = (1-u)^2 \tag{2.8}$$

$$b = v^2 \tag{2.9}$$

$$w_{k+1} = \frac{b}{a+b} \tag{2.10}$$

$$w_k = \frac{a}{a+b} \tag{2.11}$$

$$\hat{f}_s(x) = w_k \hat{f}_k(x) + w_{k+1} \hat{f}_{k+1}(x) \tag{2.12}$$

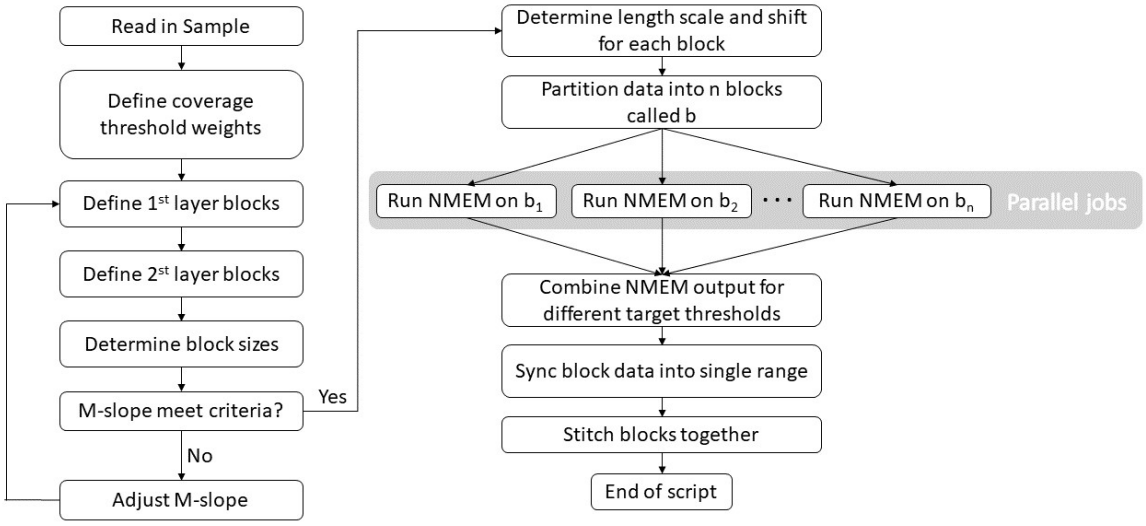

Figure 2.6: Algorithm flow chart for stitching method.

2.5 Stitching method algorithm

Features of the SE have been discussed in the previous sections and are part of the algorithm presented in the flow chart shown in figure 2.6. The pseudo code which accompanies this flowchart is described below.

1. Read in univariant sample.
2. Define the weights for different target coverage using the LL scoring function.
3. Define the upper and lower boundaries for the first layer of blocks using initial M-slope criteria.
4. Define the upper and lower boundaries for the second layer of blocks that are staggering the first layer. The second layer's block boundaries are the mean position for the respective upper and lower boundaries of the first block layer.
5. Determine block size for all layers of blocks.
6. Test if the mean block size and the number of total blocks meet acceptable criteria to ensure the blocks are not too small or too large. When the two criteria are met move on to step 7, otherwise, adjust M-slope and return to step 3.

7. Determine the length scale and shift for all blocks.
8. Partition the data according to the number of blocks. Center all the partitions about the origin and scale the data using the information determined in step 7.
9. Run the NMEM estimator for each partition of data and each target threshold in parallel.
10. Combine the output estimates for each partition using a weighted average of the different target coverage values with the weights defined in step 2. Apply scaling to the pdf estimates, as well as, position each pdf estimate to the block's original starting position.
11. Sync scaled and shifted partitioned estimates back into a single range.
12. Stitch the partitioned data in each block together using the weighted average defined in equation 2.12.

2.6 Preliminary results

2.6.1 Synthetic test dataset

To determine the efficiency and accuracy of the SE over other ubiquitous pdf estimators a standardized dataset of distributions for sample sizes from 2^9 to 2^{20} along with as many trials per sample as need has been created with the use of a MATLAB script. Having the standard dataset that utilizes MATLAB's random number generators will ensure a method for testing different estimators in an unbiased manor. Some examples of the distributions that will be under consideration are displayed in figure 2.7. This standardized test data will not only be useful for this research endeavor but will become a tool for any future work conducted in the BPMG's lab. The data is generated on call when needed through the MATLAB scripted in appendix C.2.

All random samples generated from mixture distributions were created using a binomial sampling method for each distribution in the mixture. The widely known binomial distribution for number of trials N and probability of success p is given by

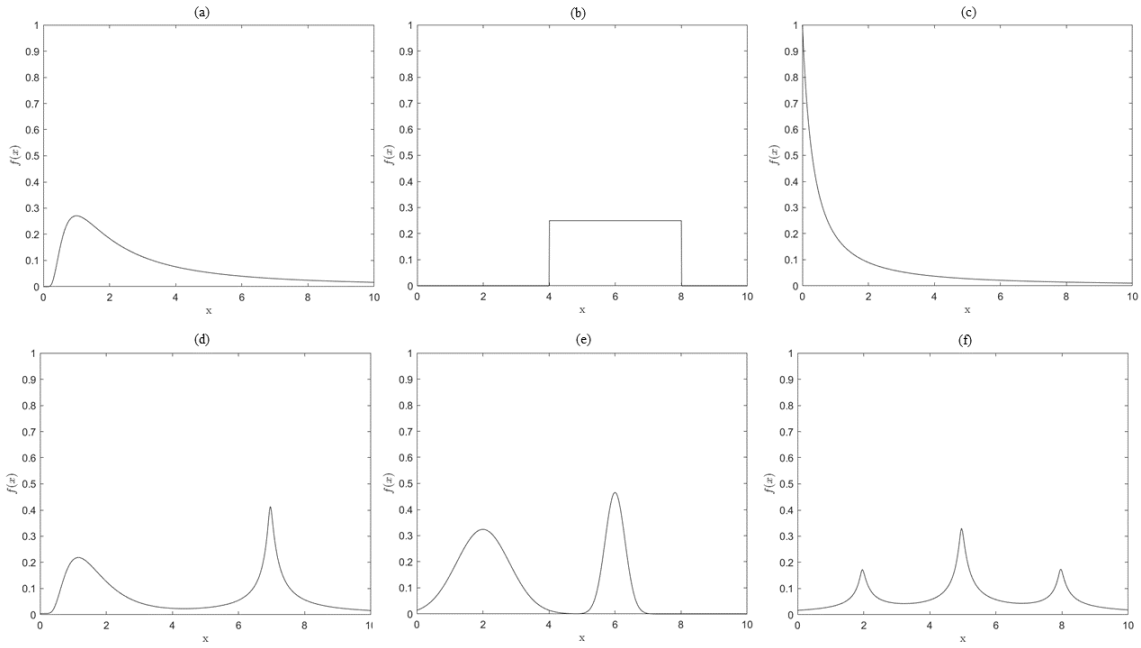

Figure 2.7: (a) Generalized extreme value distribution (b) Uniform distribution (c) Generalized Pareto distribution (d) Mixture of Birnbaum-Saunders and stable distributions (e) Mixture of two normal distributions (f) Mixture of three Stable distributions.

equation 2.13. A random sample from this distribution is defined by equation 2.14. Therefore, form distributions in a mixture distribution, the number of samples to be generated from each one is given by equations 2.15-2.17. These equations are used to create the number of data points from each distribution sequentially; starting with n_1 and ending with n_m.

$$B(N,p) = \binom{N}{x} p^x (1-p)^{(N-x)} \ ; \ x = 0, 1, 2, ..., N \tag{2.13}$$

$$B_s(N,p) \equiv random\ sample\ from\ B(N,p) \tag{2.14}$$

$$n_1 = B_s(N, p_1) \tag{2.15}$$

$$n_i = B_s\left(N - \sum_{j=1}^{i-1} n_j,\ p_i\Big(\sum_{j=i}^{m} p_j\Big)^{-1}\right) \tag{2.16}$$

$$n_m = N - \sum_{i=1}^{m-1} n_i \tag{2.17}$$

This ensures proper random sampling from each distribution in the mixture. Other methods using a uniform random sampling method were initially employed but were less computationally efficient compared to using a binomial random sampling approach.

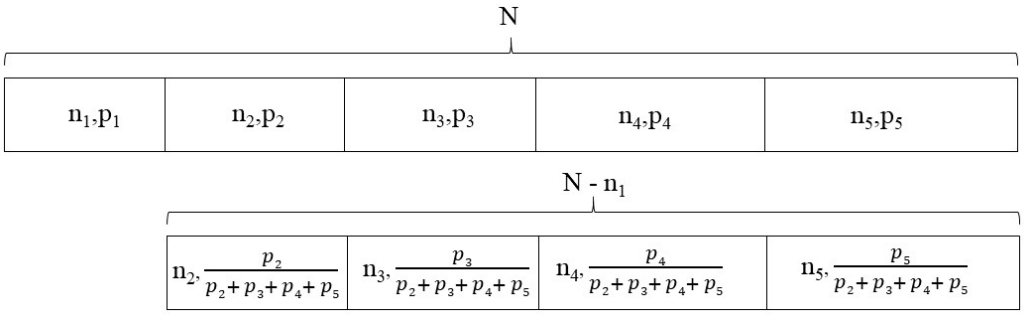

Figure 2.8: Visual representation of mixture sampling procedure for the first two distributions in a mixture of five distributions.

For example, using equations 2.15-2.17 to create a mixture distribution of five with probabilities $p_1,\ p_2,\ p_3,\ p_4$, and p_5 for each distribution type yields,

$$n_1 = B_s(N, p_1) \tag{2.18}$$

$$n_2 = B_s\Big(N - n_1,\ \frac{p_2}{p_2 + p_3 + p_4 + p_5}\Big) \tag{2.19}$$

$$n_3 = B_s\Big(N - n_1 - n_2,\ \frac{p_3}{p_3 + p_4 + p_5}\Big) \tag{2.20}$$

$$n_4 = B_s\Big(N - n_1 - n_2 - n_3,\ \frac{p_4}{p_4 + p_5}\Big) \tag{2.21}$$

$$n_5 = N - n_1 - n_2 - n_3 - n_4 \tag{2.22}$$

The algorithmic approach used in the example creates starts by calculating n_1 by randomly sampling the binomial distribution for N and p_1. Given n_1 data points from N are from distribution one, n_2 data points are generated by random sampling the binomial distribution for $N - n_1$ available data points and the new conditional probability $\frac{p_2}{p_2+p_3+p_4+p_5}$. This procedure is continued until n_5, where n_5 simply equals the number of data points not allocated to any of the other distributions in the mixture. Figure

2.8 shows a visual interpretation for what the mixture sampling procedure does for the first two distributions. The number of data points allocated for a distribution is removed from consideration and the probabilities for the other distributions are then equal to the conditional probability given the remaining distributions.

2.6.2 Fixed number of points per block

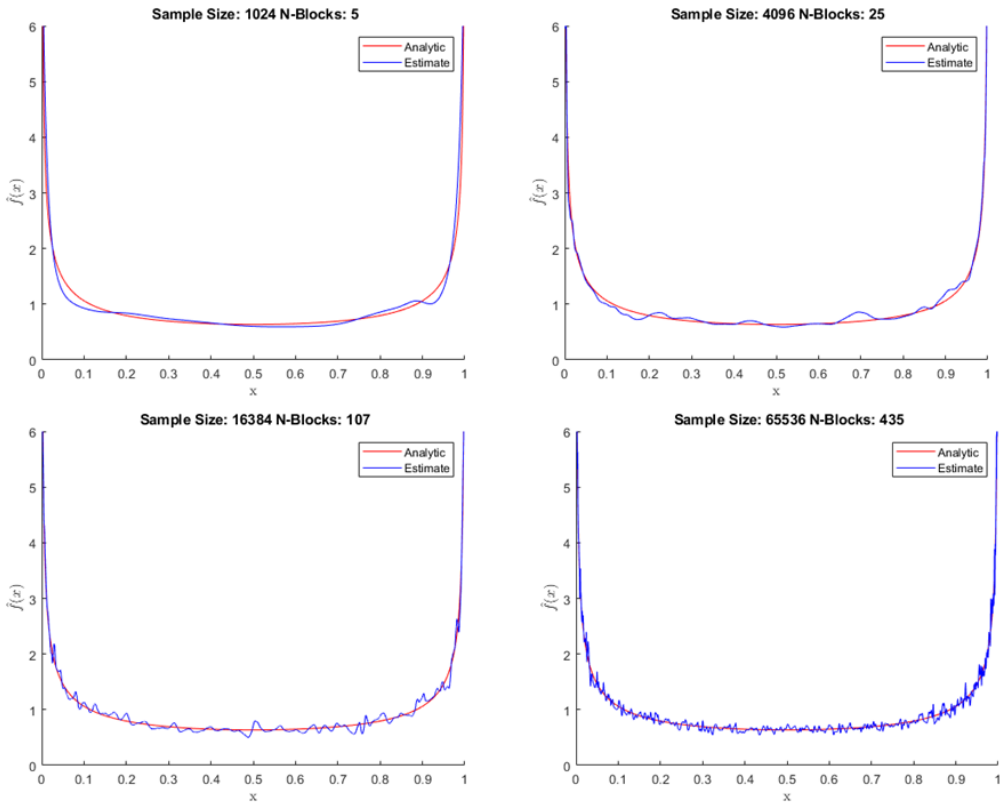

Figure 2.9: Displays the pdf estimates, $\hat{f}(x)$, for different sample sizes for a Beta distribution (a = 0.5, b = 0.5) and where the number of data points per block were fixed.

A method initially explored for defining the block size was to require each block to have the same number of data points, N_{block}. Doing so could ensure that the user assigned enough data points per block to gain a reasonable estimate. Fixing the number of data points per block required the number of blocks to vary depending on the sample size. As seen in Figure 2.9 this will led to a high variance in $\hat{f}(x)$ compared to $f(x)$, but an $\hat{f}(x)$ with little bias. Another disadvantage to fixing N_{block}

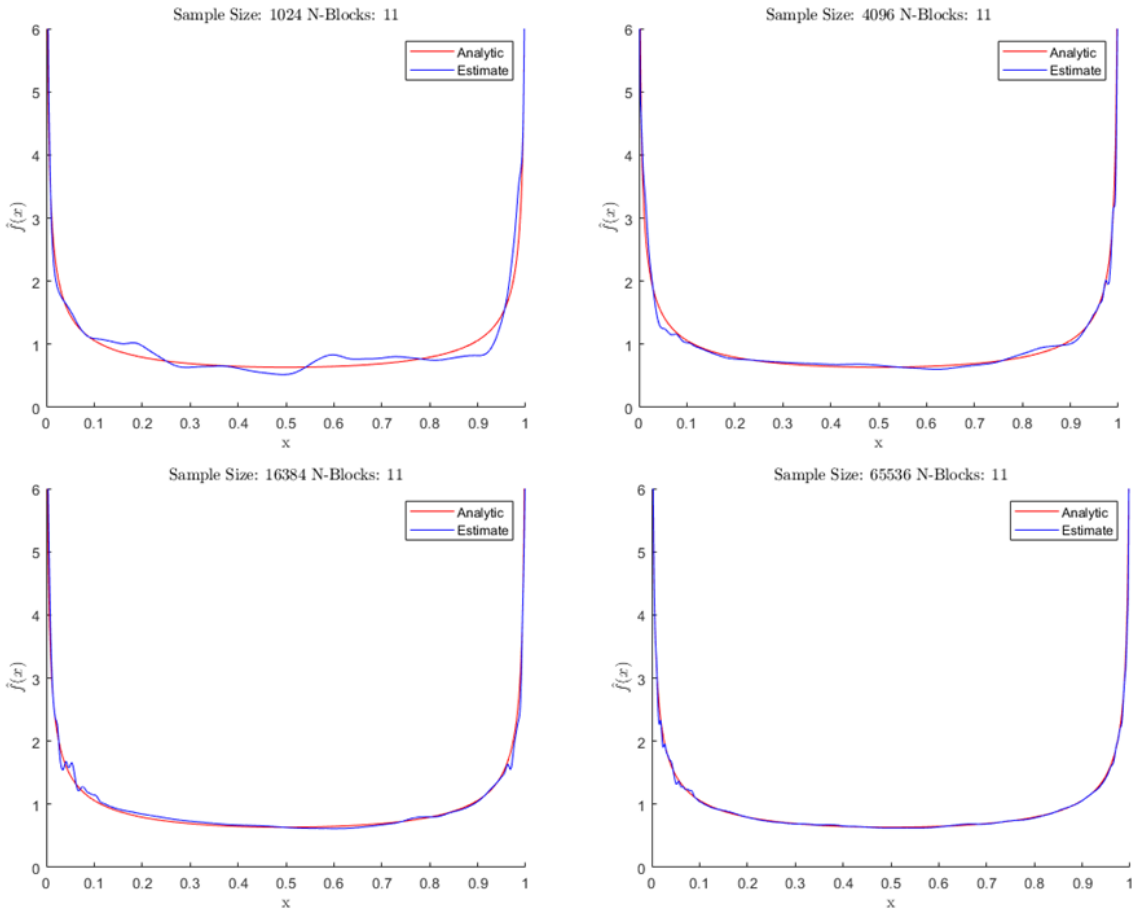

Figure 2.10: Displays the pdf estimates, $\hat{f}(x)$, for different sample sizes for a Beta distribution (a = 0.5, b = 0.5) and where the number of blocks were fixed.

is that for sets of data where the sample sizes vary greatly over the set an optimal N_{block} may only work well over a subset of the entire set. Some N_{block} may work for large samples, but will be incompatible for small samples, while the converse leads to an $\hat{f}(x)$ with high variance. For these reasons this method of block definition was eliminated.

2.6.3 Fixed number of blocks

Another approach to defining the block sizes was to require the number of blocks to be fixed opposed to the number of data points in each block. This approach allowed the user to define the total number of blocks by requiring that a certain percentage of the total sample falls into each block. Figure 2.10 shows $\hat{f}(x)$ for a beta distribution

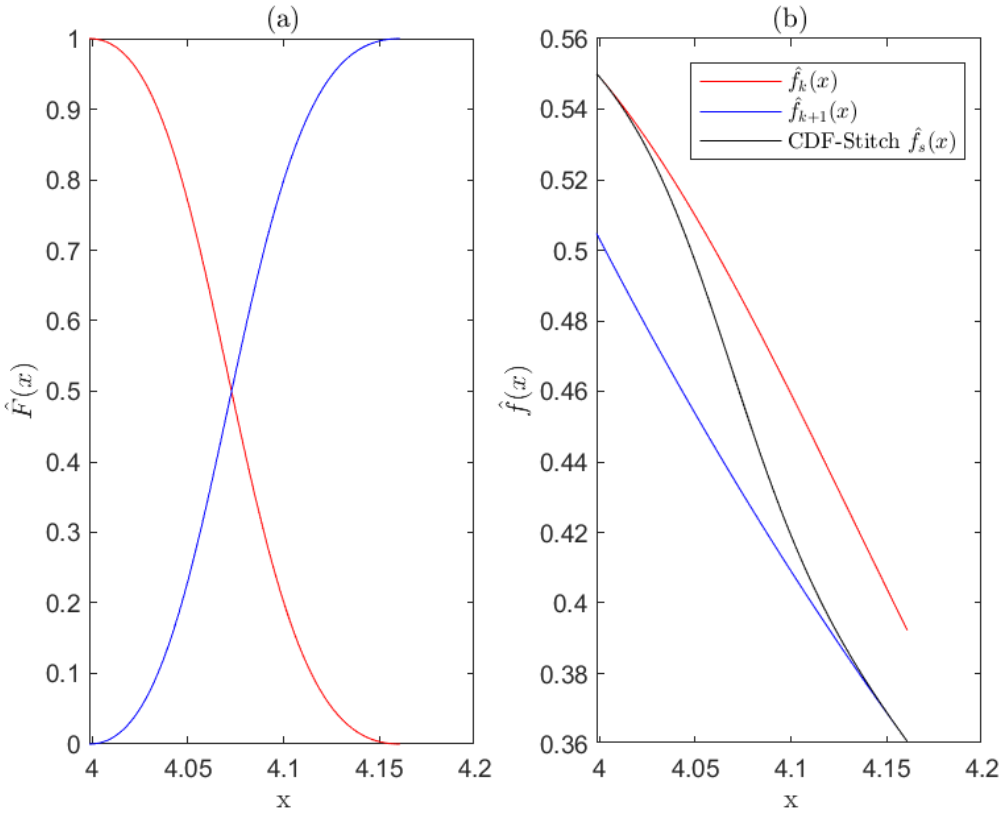

Figure 2.11: (a) Shows the cdf etimates, $\hat{F}(x)$, for the nth and n+1 block's overlap region. (b) Displays the pdf estimates, $\hat{f}(x)$, for the adjacent blocks, as well as, the weighted averaged estimate labeled stitched pdf.

for varying sample sizes. Fixing the number of blocks by scaling the number of data points reduces the variance of the $\hat{f}(x)$. This was to be expected, because as the sample size increases the NMEM estimator is able to glean what the important trends in the data are verse unimportant noise. However, this method of block definition has difficulties with samples that contain heavy tails or divergences. This is caused from the density of the blocks data being relatively sparse in the one region while dense in another, which can lead to the NMEM having difficulties computing the pdf estimate for the block. For this reason, the block definition method was eliminated.

2.6.4 Weighted average using CDF

The current stitching method uses a weighted average with the cdf, $\hat{F}(x)$, for the adjacent blocks being the weights. Figure 2.11 displays an example of $\hat{F}(x)$ along with $\hat{f}(x)$ for two adjacent blocks. Using the cdf as weights to generated the estimate, $\hat{f}_s(x)$, in the blocks overlap region ensures that the stitched curved will still exhibit the behavior for the underlying data.

Another method previously explored for generating $\hat{f}_s(x)$ was to us a 3rd-order polynomial fit between $\hat{f}_k(x)$ and $\hat{f}_{k+1}(x)$. This was achieved by taking the third order polynomial, $\hat{f}_s(x) = c_4x^3 + c_3x^2 + c_2x + c_1$ and requiring the following boundary conditions,

$$\hat{f}_k(x_{Right}) = \hat{f}_{k+1}(x_{Right}) \tag{2.23}$$

$$\hat{f}_k(x_{Left}) = \hat{f}_{k+1}(x_{Left}) \tag{2.24}$$

$$\frac{d\hat{f}_k(x_{Right})}{dx} = \frac{d\hat{f}_{k+1}(x_{Right})}{dx} \tag{2.25}$$

$$\frac{d\hat{f}_k(x_{Left})}{dx} = \frac{d\hat{f}_{k+1}(x_{Left})}{dx} \tag{2.26}$$

$$\tag{2.27}$$

to be used to solve for the coefficients c_4, c_3, c_2, c_1. Where x_{Left} and x_{Right} are the left and right most x-coordinates for the overlap region under consideration. Using the polynomial fit to stitch the adjacent pdf estimates together yielded similar results to the weighted average method previously discussed and for this reason was not considered further, although in remains a viable alternative.

2.6.5 Blacklist for $\hat{f}_k(x)$

For the class of distributions with heavy tails the NMEM has shown to have a high likelihood to fail for the exterior blocks on the heavy tail. For this reason, a blacklist of the failed NMEM estimates, $\hat{f}_k(x)$, is created. These blacklisted blocks are

initially removed. The pdf estimate is attempted for a sub sample for the blacklisted blocks. If successful more of the original blacklisted sample may be added and another pdf estimate attempt made. This is an iterative process that allows heavy tails of distributions to be better estimated. Figure 2.12(a) shows an example of how a failed block estimate leads to systematic error in the SQR plots and figure 2.12(b) shows how the blacklisting routine is able to remove the systematic error. However, after further research into another method of block definition, called R-ratio, the need to have the blacklisting routine was potentially unnecessary.

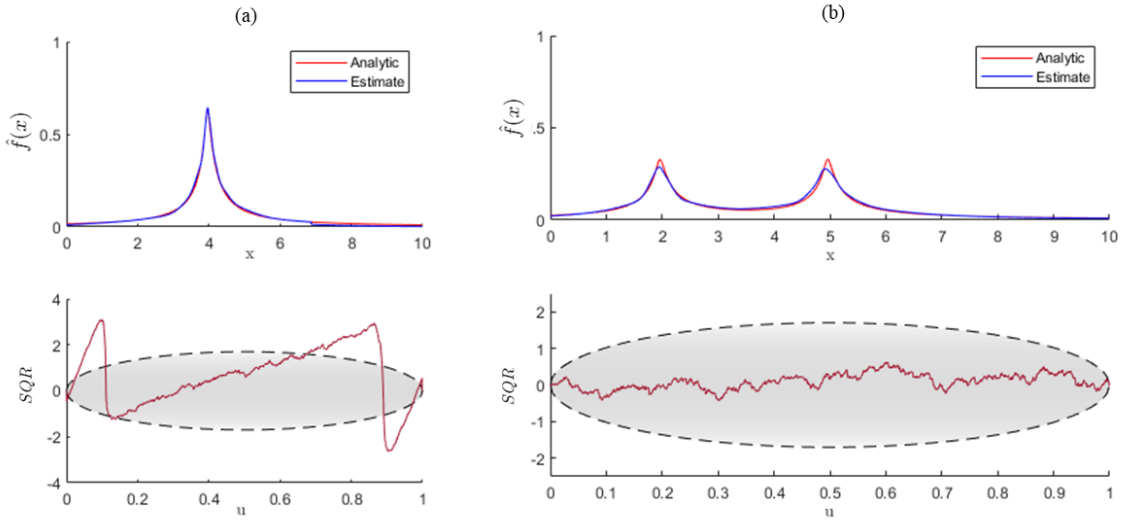

Figure 2.12: Figure (a) shows an example of a failed block estimate for a heavy tailed stable distribution. This leads to visible systematic error in the SQR plot (b) blacklisted routine is implemented for a mixture distribution that is more prone to yield failed block estimates than the single stable distribtuion. The SQR plot shows that the systematic error no longer exists, because there are no longer any failed block estimates.

CHAPTER 3: RESEARCH METHODOLOGY

3.1 Optimized branching tree

Another adaptive approach for determining the appropriate size of each block, as well as, the total number of blocks has been termed the optimized branching tree method. The adaptive algorithm initially evaluates the variation in the density of the sample's data points then recursively makes partition decisions based on a comparison of the parameter ξ with the sample size dependent threshold parameter $\Gamma(N)$ which creates new blocks. $\Gamma(N)$ and ξ are defined in equations 3.1 and 3.2, where c_0 is a scaling coefficient and p is an exponential coefficient that are heuristically determined to pick the appropriate $\Gamma(N)$ for general probability density estimation applications. B represents the number of data points per block and R, referred to as R-ratio, is a parameter that represents the variation in the density of the data points in the block.

$$\Gamma(N) = c_0 N^{p_0} \tag{3.1}$$

$$\xi = \frac{B^{p_1} * R^{p_2}}{N^{p_2}} \tag{3.2}$$

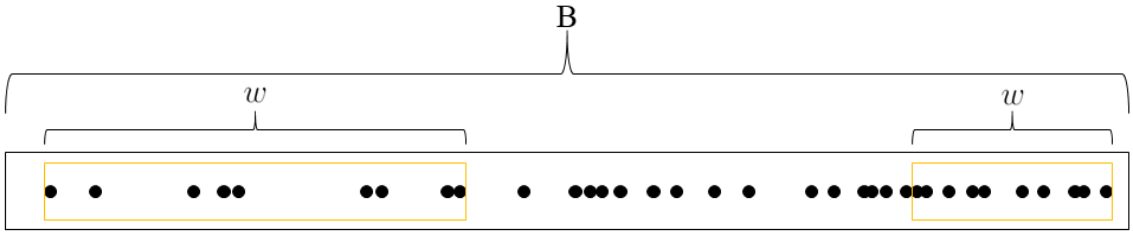

Figure 3.1: R-ratio for a block is calculated by taking the ratio of the average distances between adjacent data points that lie within the windows, w, shown in yellow.

To calculate R-ratio a block's data is initially sorted then the difference of all adjacent pairs of data points are calculated. After, the set of Δx values are sorted

Table 3.1: Table of the parameter set used to generated the SE pdf estimates.

c_0	c_1	p_0	p_1	p_2	p_3	p_4
8	0.125	0.25	1	0.5	1	0.5

too. As shown in equations 3.4 and 3.5, Δx_{min} is the mean density for the $w(N)$ most dense pairs of data points in the block's sample and conversely Δx_{max} is the mean density for the $w(N)$ least dense pairs of data points in the block's sample. Figure 3.1 shows which points are being selected for a given block, B, and w. $w(N)$ is a sample size dependent parameter that is utilized to select the number of values from the set of Δx to use when determining Δx_{min} and Δx_{max}. This parameter is calculated using equation 3.3 as a rounded up portion of the total sample size, N, where c_1 is a percentage coefficient. The R-ratio is defined as the ratio of Δx_{min} to Δx_{max} as shown in equation 3.6. The R-ratio coefficient will approach 1 the more uniform the density of the data points in the block and otherwise will increase the less uniform the density of the data points in the block. The parameters from equations 3.1, 3.2, 3.3 are displayed in table 3.1.

$$w(N) = \lceil c_1 N^{p_4} \rceil \tag{3.3}$$

$$\Delta x_{min} = mean\{\Delta x_k \ for \ k \in [1, w]\} \tag{3.4}$$

$$\Delta x_{max} = mean\{\Delta x_k \ for \ k \in [N - w + 1, N]\} \tag{3.5}$$

$$R = \frac{\Delta x_{max}}{\Delta x_{min}} \tag{3.6}$$

The optimized branching tree algorithm starts by calculating ξ for the entire sample and if $\xi < \Gamma$ then the NMEM is applied to the entire sample without creating blocks, otherwise a random partition, n_0, is picked and ξ is calculated for the two blocks created by the partitioning. Next the difference between the ξ parameter for the two

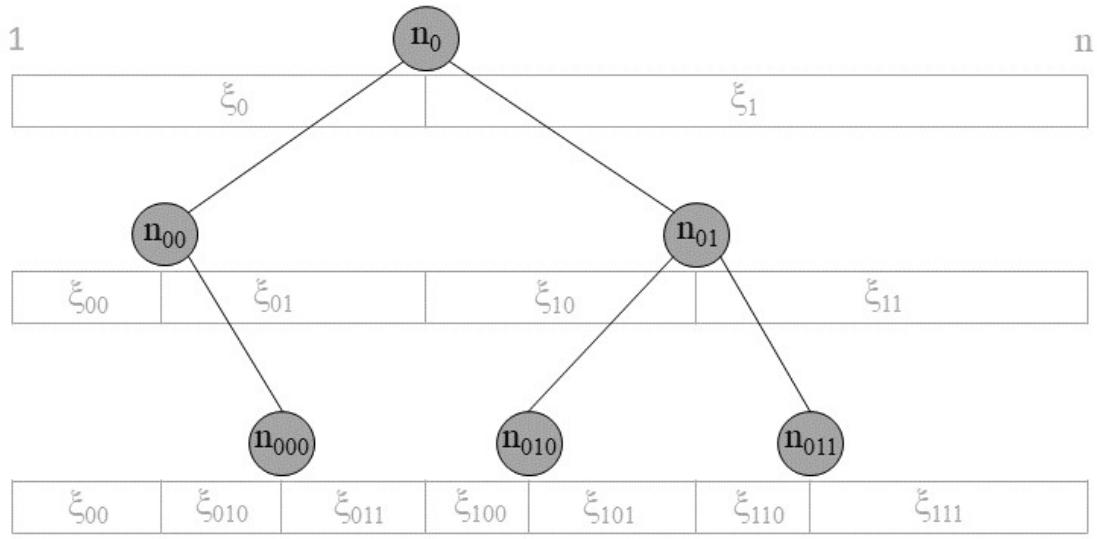

Figure 3.2: Example of branching tree algorithm. n_0 is initially chosen then $\Delta\xi$ is minimized. More levels are created, and the procedure is repeated until all ξ are less than Γ.

blocks,

$$\Delta\xi_0 = \xi_1 - \xi_0 \tag{3.7}$$

is minimized to obtain two blocks that exhibit approximately the same ξ, where ξ_0 is the first level and branch of the tree, while ξ_1 is the first level and second branch of the tree. An example of how the optimized branching algorithm progresses is shown in figure 3.2.

For every level of the tree ξ is compared to Γ and if $\xi < \Gamma$ a new branch in the tree is created, which creates two new blocks. ξ is product of, B, the number of data points per block with, R, a ratio that represents a measure of how uniform the density of the data is for the data points of each partition.

The approach aims to create blocks with subsamples that exhibit a maximum level of uniform density. By requiring the block to contain nearly uniform density data ensures that the estimate made for each block is computationally easier for the NMEM

to obtain and less prone to an inferior estimate for difficult samples.

3.2 Difference error analysis

The mean difference is calculated for every data point x_n from all estimates gained from M trials using equation 3.8. This gives a measure of where on average the stitch estimator is under or over estimating the estimate pdfs of the sample.

$$ME(x_n) = \frac{1}{M}\sum_{i=1}^{M}(f(x_n) - \hat{f}(x_n)_i) \tag{3.8}$$

Knowing the mean error over the estimate pdfs is quite useful information, however, if there are large variations in the estimated pdfs they may counter balance to yield a low mean error. Therefore, the standard deviation of the MAE is computed by equation 3.9 along with the maximum and minimum error for every point x_n that is estimated over the M trials.

$$\sigma(x_n) = \sqrt{\frac{\sum_{i=1}^{M}\left(\left(f(x_n) - \hat{f}(x_n)_i\right) - ME(x_n)_i\right)^2}{M-1}} \tag{3.9}$$

CHAPTER 4: RESULTS

4.1 Optimized branching tree

The block definition for a beta distribution ($a = 2$ and $b = 0.5$) using the optimized branching tree (OBT) method is shown in figure 4.1. Figure 4.1(a) shows the value the natural log of ξ for all blocks per level. Therefore, when $ln(\xi) > ln(T)$ a new level is created with two new branches, otherwise the block exhibits acceptable uniformly dense data. The zeroth level represents the ξ value for the total sample which is shown in figure 4.1(b) as the grey circles with black edges. The first level in figure 4.1(a) shows two black dots, which are the two values of ξ calculated for the blocks created by the black partition dot of level 1 in figure 4.1(b).

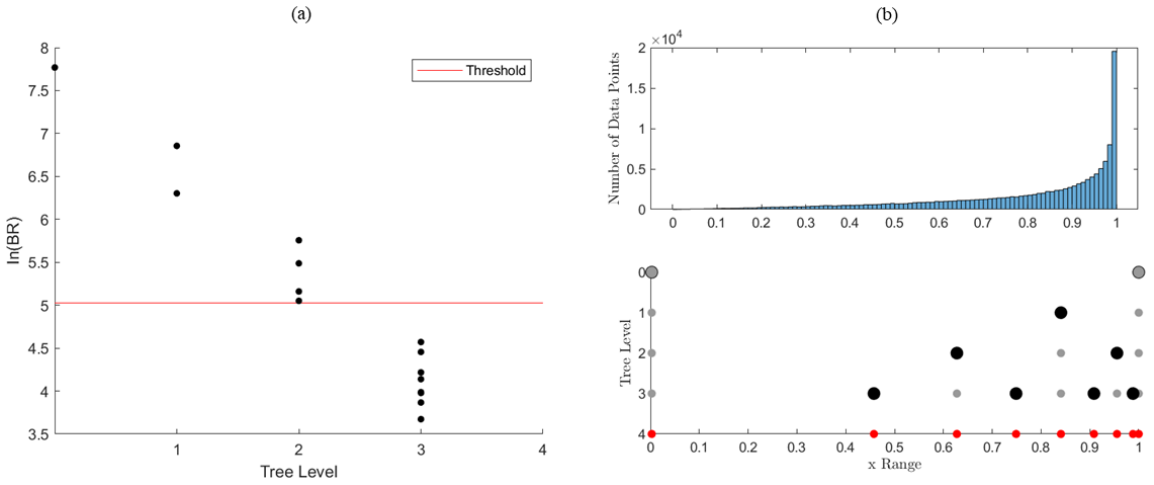

Figure 4.1: (a) Shows the $\xi = BR$ per block for any given level along with the threshold cut off for the given sample size (b) displays the distribution of the data and the block distribution for a beta distribution.

Figure 4.1(b) shows how the blocks are distributed based upon the distribution of the data. More blocks are created towards the region of data that has a higher density to try and maximize how uniform the density is across all of the blocks. This

is done by making the blocks smaller in size, which is shown to be the case in figure 4.2(a).

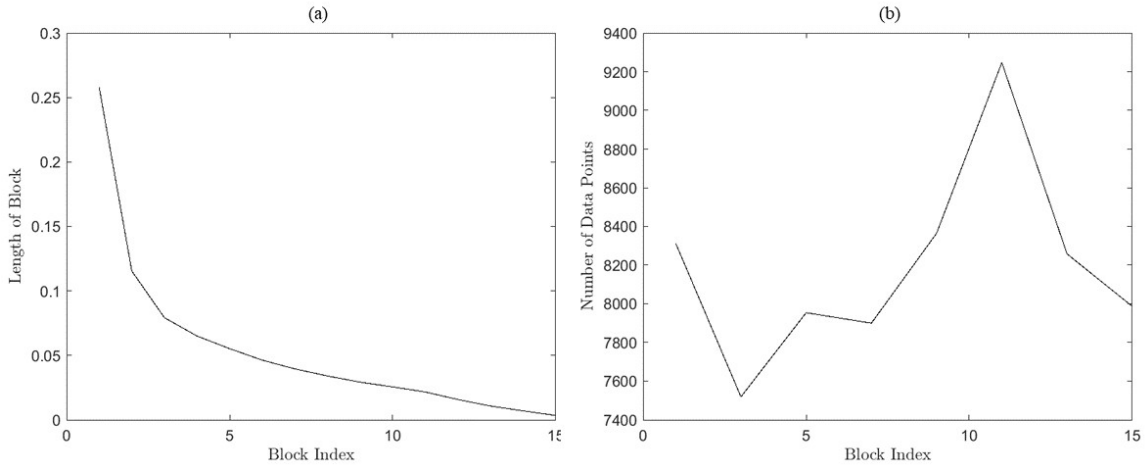

Figure 4.2: (a) Shows the length of each block (b) Shows the number of data points per block for the beta distribution in figure 4.3.

The number of data points per block is shown in figure 4.2(b), which for the beta distribution trends to increases towards the region with higher density. However, due to the random fluctuations in the sample this trend is only approximate, because there can exist clusters of data points in dense or sparse regions.

Figure 4.3(c) shows the pdf estimate obtained from the block distribution shown in figure 4.1(b) and figure 4.2. The OBT block definition method leads to an excellent pdf estimate with the pdf estimates per block shown in figure 4.3(b). However, from the SQR plot in figure 4.3(a) the overall pdf estimate is starting to be over fitted to the sample.

Another example of how the OBT method is able to partition the sample appropriately is shown in figure 4.4(a) and (b) for a contaminated normal distribution. The distribution of the block sizes is displayed in figure 4.5(a) and shows that the blocks decrease in length as the density increases. Although the block length may be decreasing, in general, the block size increases when moving from the sparse tails towards the dense center of the sample as seen in figure 4.5(b).

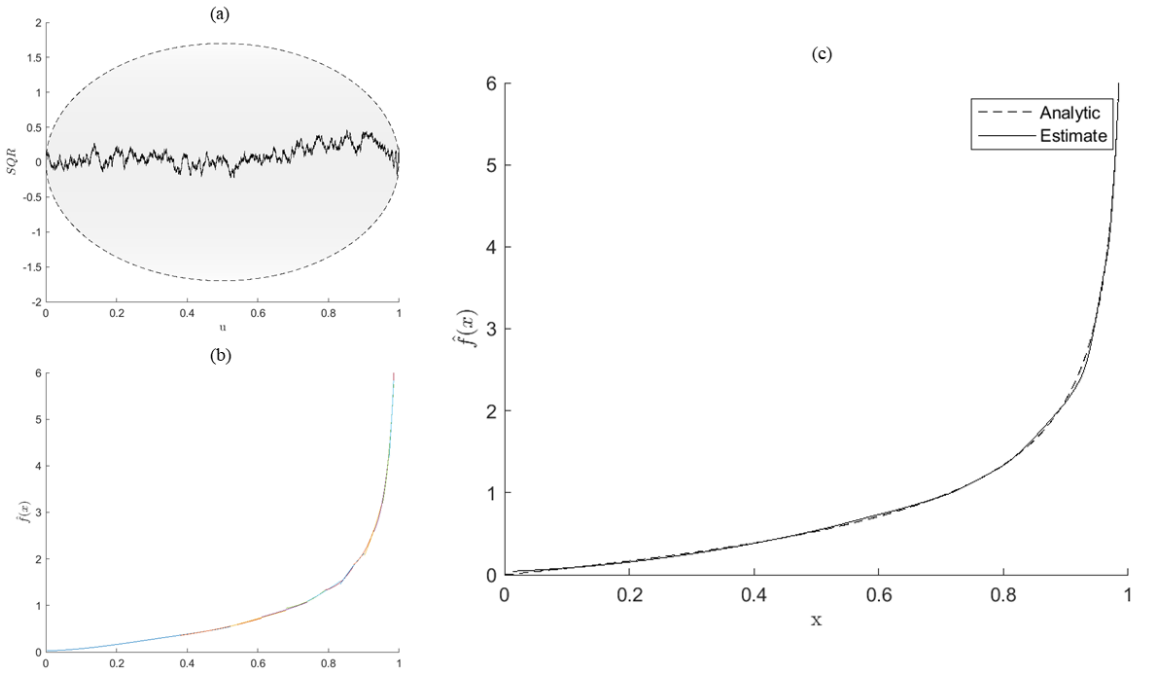

Figure 4.3: (a) The SQR plot for the beta distribution (b) Displays the estimates per block prior to stitching (c) Shows $\hat{f}(x)$ for the beta distribution.

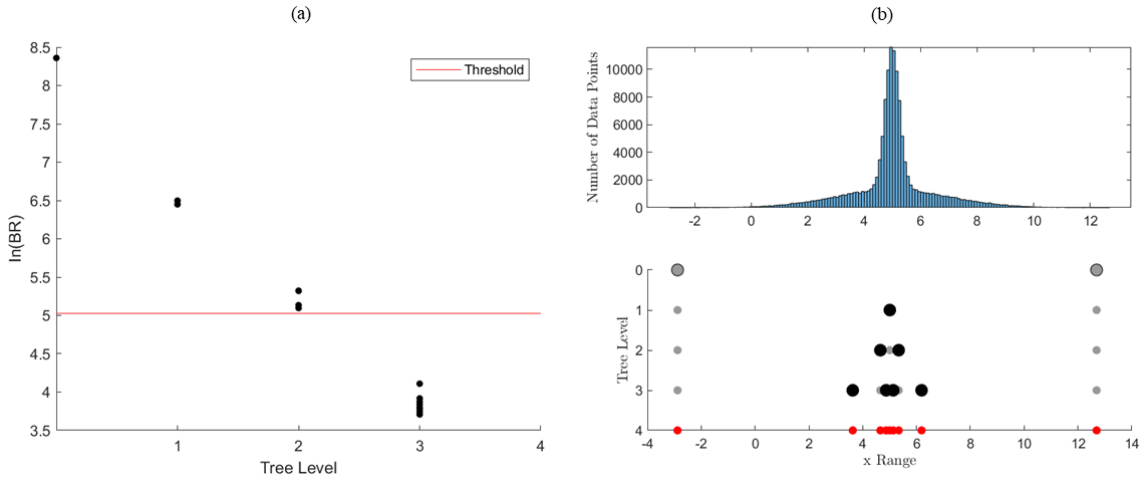

Figure 4.4: (a) Shows the $\xi = BR$ per block for any given level along with the threshold cut off for the given sample size (b) displays the distribution of the data and the block distribution for a contaminated normal distribution.

The overall pdf estimate for the sample along with the pdf estimates per block are shown in figure 4.6(c) and (b) and show the OBT block definition method calculates a block distribution that produces an excellent pdf estimate for the contaminated

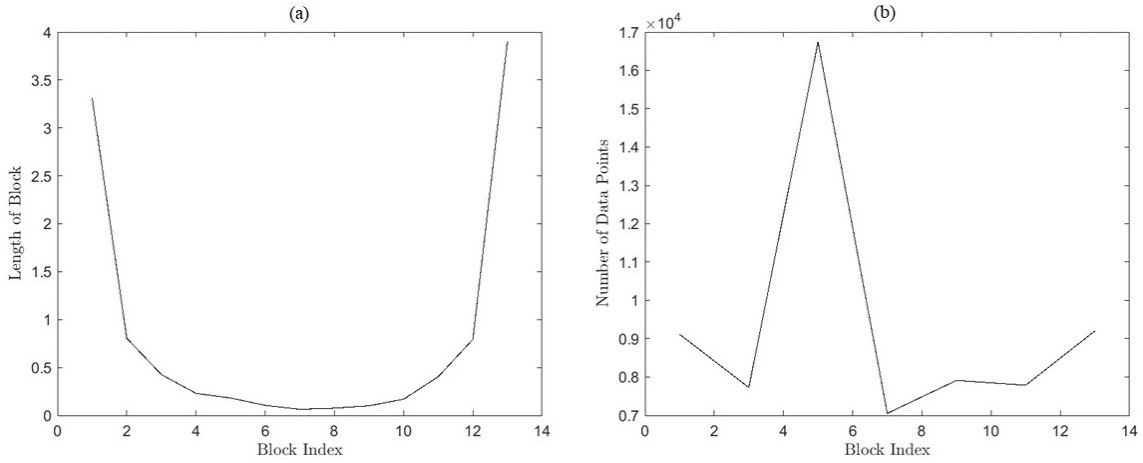

Figure 4.5: (a) Shows the length of each block (b) Shows the number of data points per block for the beta distribution in figure 4.6.

normal distribution sample. The SQR plot in figure 4.6(a) shows that the overall pdf estimate is an acceptable estimate.

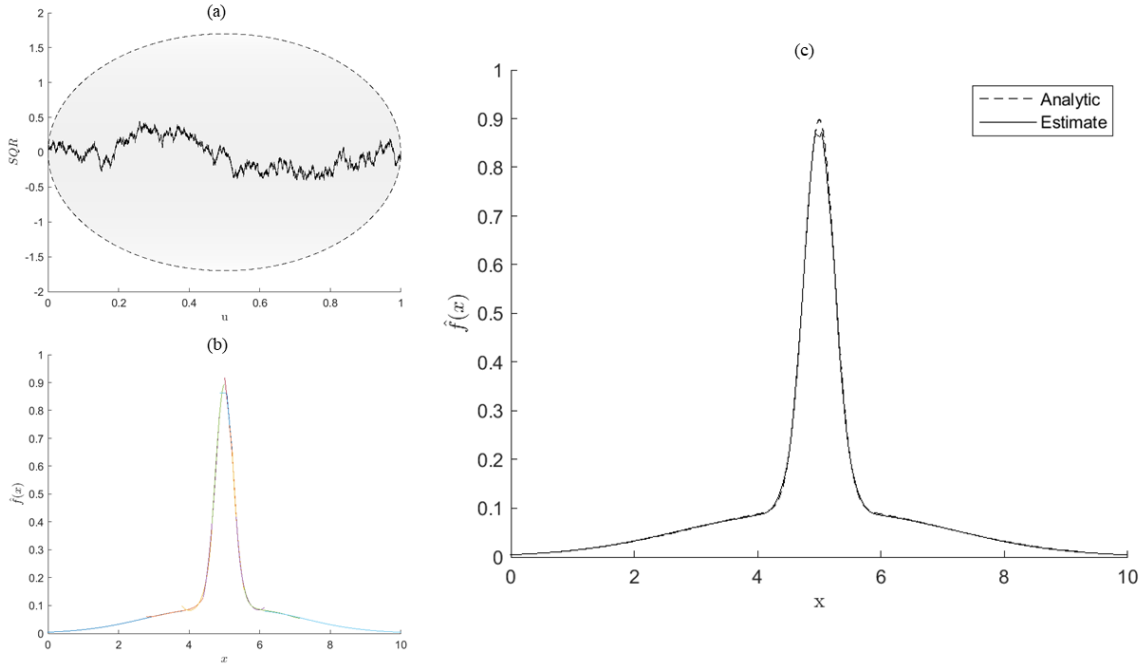

Figure 4.6: (a) The SQR plot for the beta distribution (b) Displays the estimates per block prior to stitching (c) Shows $\hat{f}(x)$ for the contaminated normal distribution.

The methods for minimizing $\Delta\xi$ have all consisted of efficiently finding a global minimum for $\Delta\xi$. When evaluating ξ for a block there are two extremes, either when

ξ is calculated for the entire block or calculated for the smallest allowable potential new block to be created. When ξ is calculated for the entire block both B and R will be maximum, but when ξ is calculated for the smallest allowable potential new block, then B will be its minimum value and R will be less than that for the entire block. Therefore, as ξ_0 for one potentially new block is decreasing in value the ξ_1 for the other potentially new block will be increasing, which will lead to a point where $\xi_0 = \xi_1$ or $\Delta\xi = 0$. Figure4.7 shows an example of the behavior for R, B, and ξ.

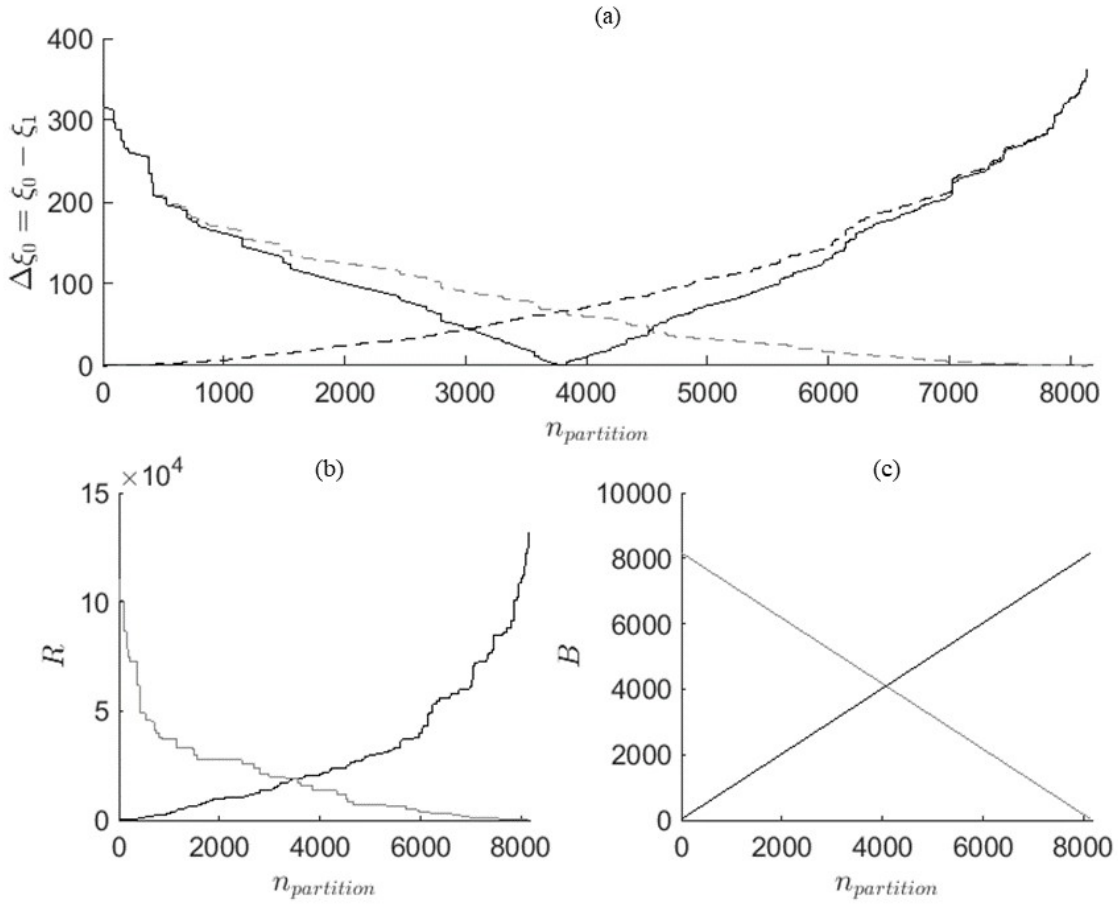

Figure 4.7: (a) The distribution of $\Delta\xi$ for the partition made in level one when the partition varies from 1 to N. ξ_0 (black dashed line) and ξ_1 (grey dashed line) (b) The distribution of R as the partition sweeps over the range of the sample. R_0 (black dashed line) and R_1 (grey dashed line) (c) The distribution of B as the partition sweeps of the range of the sample. B_0 (black dashed line) and B_1 (grey dashed line)

4.1.1 maximum block size

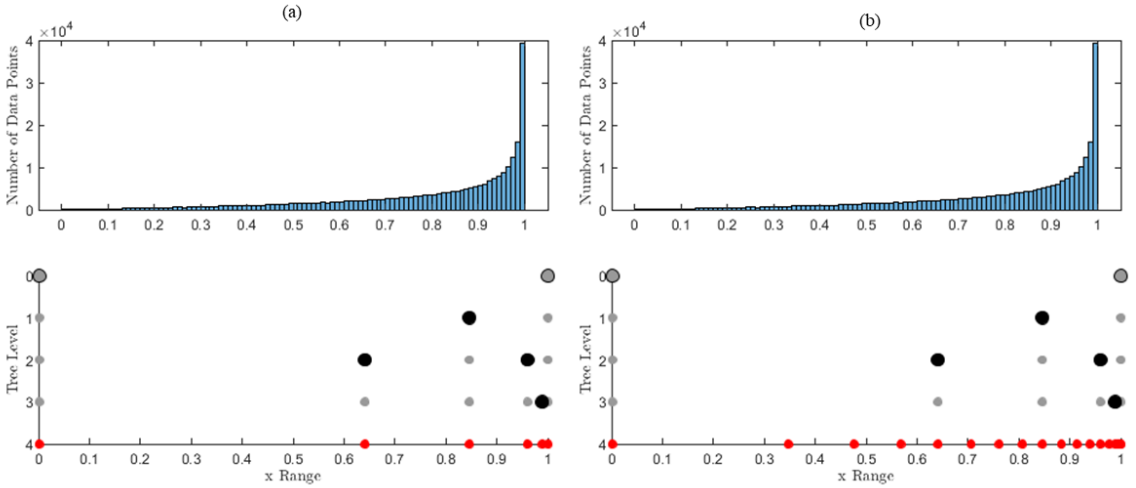

Figure 4.8: (a) The block distribution without limiting the maximum block size (b) The block distribution with limiting the maximum block size to 20,000.

After developing the OBT method for determining the optimum block distribution and heuristically determining a parameter set that worked for general distribution cases, it was observed that the total number of blocks scaled as the sample size, but perhaps not as quickly as desired. Future work will see if this can be mitigated using another parameter set. It could be that the current parameter set is optimal for maintaining uniformity in the density of data within each block even when the sample sizes becomes very large, however, even when this is true it becomes more computationally costly to generate pdf estimates per block. Also, if a block has roughly uniform density across the block for a very large N_{block}, it is reasonable to be able to split the block into smaller blocks that still contain adequate information to make good pdf estimates. For these reasons, there was motivation for the development of a script routine that could control the maximum block size to maintain computational efficiency for very large samples. Figure 4.8(a) shows the block distribution for a sample with 262,144 data points generated from a beta distribution. Figure 4.8(b) shows the new block distribution for the sample after the maximum block size limit

of 20,000 was enforced. This method will be later shown to drastically improve the computation cost the SE requires for large samples.

4.1.2 Average behavior

To evaluate how consistent the SE is when determine the pdf estimate for a specific sized sample, 1000 trials were generated for a sample size of 512 data points for a normal bimodal distribution. Figure 4.9, shows the pdf estimates for all 1000 trial samples and the average overall 1000 trials. There are large fluctuations over the 1000 trials, but all estimates show to capture the features of the data well.

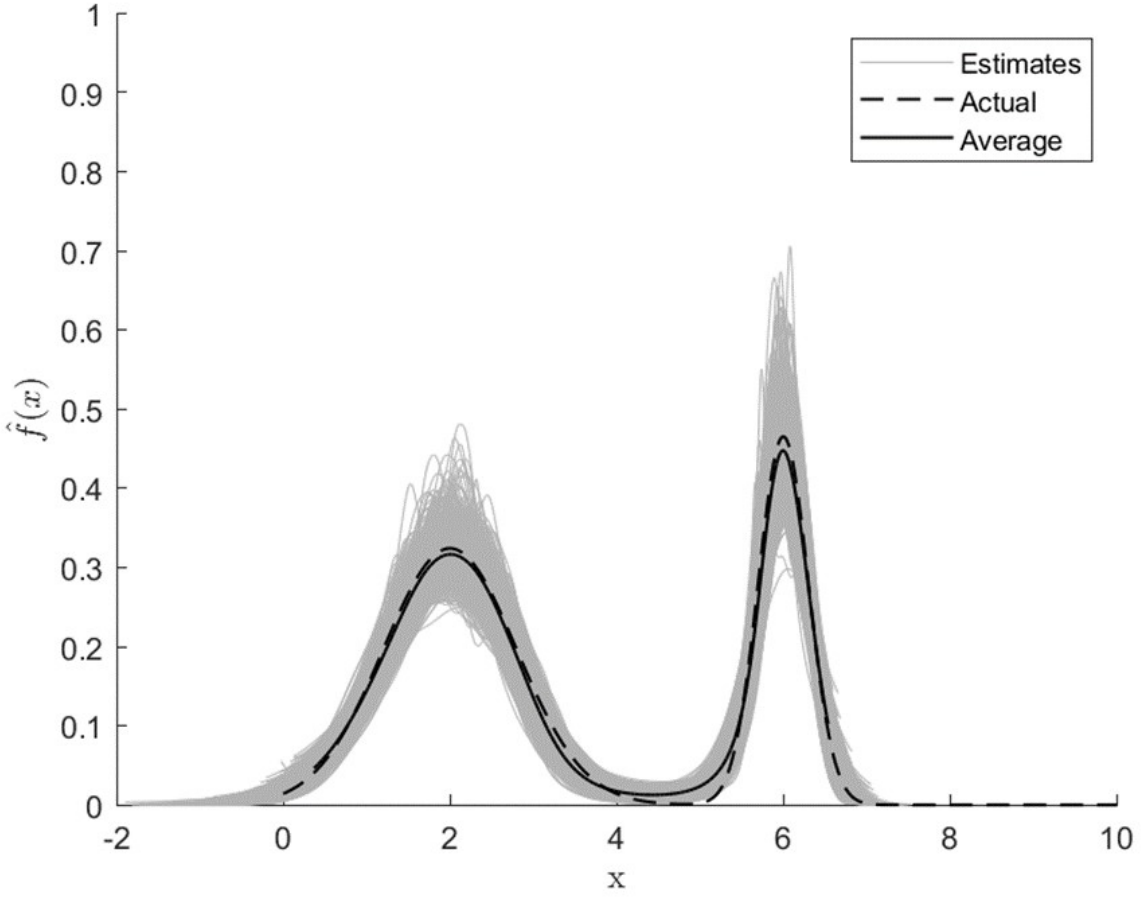

Figure 4.9: The pdf estimates from 1000 trials of samples of size 512 data points and the average across all trials.

Figure 4.10(a) shows the distribution for the ME over the 1000 trials along the with standard deviation, minimum, and maximum error for all pdf points estimated.

This style of figure has been useful for evaluating where the SE is doing well or not for various distributions. To confirm that the pdf estimates for 1000 trials were all acceptable, the maximum and minimum SQR points were checked to ensure that they all stay with in the 99% confidence interval. Figure 4.10(b) shows that for the 1000 trials all of the SQR plots stayed within the 99% confidence interval.

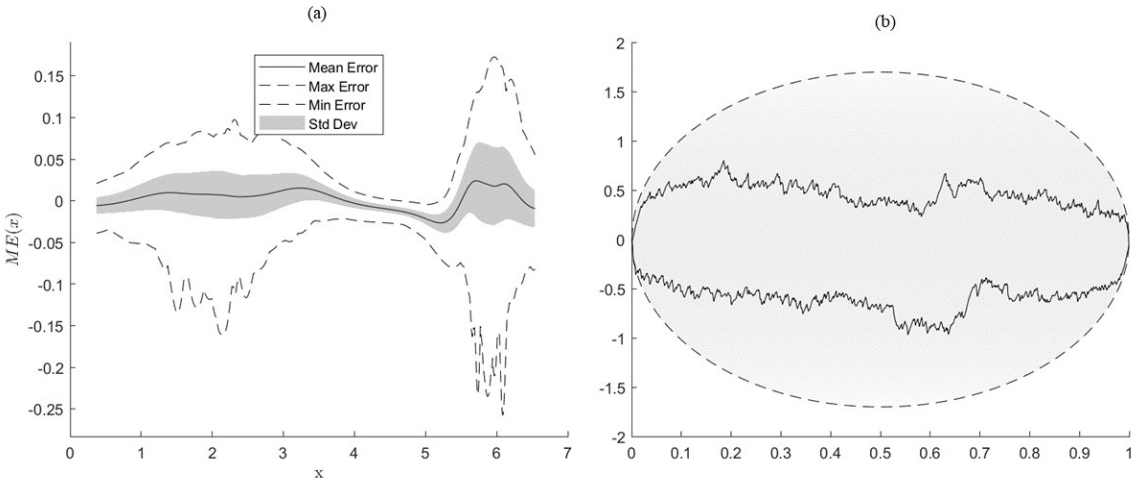

Figure 4.10: (a) The mean error across the distribution for every estimated data point, the standard deviation of the error, and the maximum/minimum error for 1000 trials (b) The maximum/minimum sqr fluctuations across all 1000 trials.

4.1.3 sub-sampling sampling

sub-sampling sampling consist of starting with a sample and then generating new samples from the original. There are many specific detail differences when one talks of sub-sampling sampling, but in this work the sub-sampling sampling procedure consist of generating subsamples without replacement that are smaller than the original sample. Then the pdf estimate is determined for each subsample and averaged to give the final pdf estimate. There were two motivations for implementing this method 1) the quality of the pdf estimate could be improved by averaging over many random fluctuations in pdf estimates that would arise and 2) the random fluctuations in the pdf estimates may reduce over fitting. Over fitting is characterized by the SQR plots having small fluctuations centered around zero.

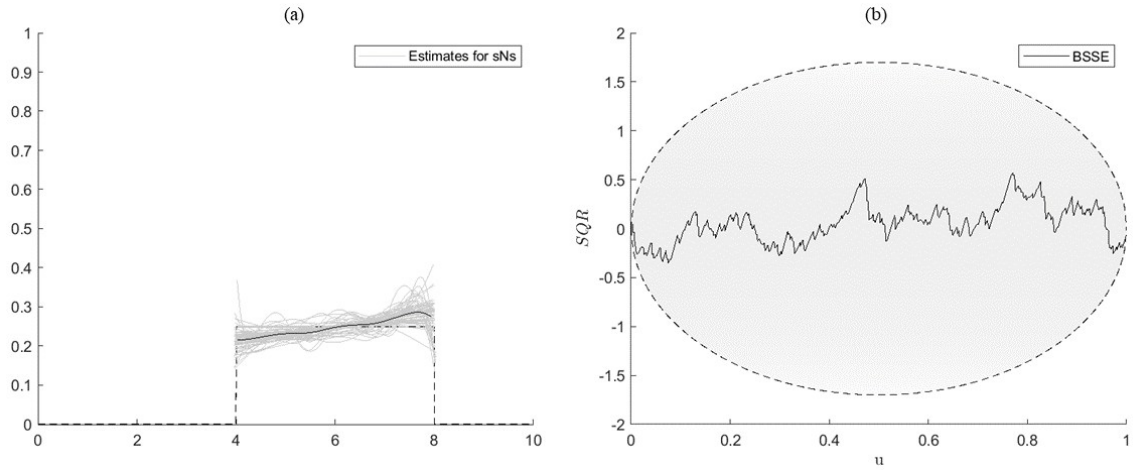

Figure 4.11: $\hat{f}(x)$ determined from averaging 70 pdf estimates from subsamples equal to 60% in size from the original sample of 512 data points. (a) The pdf estimates per subsample and the average pdf estimate (b) The SQR plot for the average pdf estimate.

Figure 4.11(a) displays 70 pdf estimates calculated from subsamples that were 60% the size of the original sample. The pdf estimate for the sample of 512 data points from the uniform distribution shows a good pdf estimate for the actual distribution. Figure 4.11(b) further confirms that the pdf estimate is acceptable for the original sample.

Figure 4.12(a) displays the same sub-sampling sampling procedure but from a sample of size 65,636 data points. The pdf estimates from the boot strap samples all give excellent pdf estimates and prevents over fitting as seen through the random fluctuations in the SQR plots from figure 4.12(b).

4.2 Estimator method comparison

In the figures that follow the SE methods is compared to the NMEM estimator along with several popular pdf estimators commonly used in the programming language R. Packages available in R were used for the qualitative comparison of the SE due to the popularity of the language as well as its accessibility. For the comparison of the five pdf estimators used a range of distributions were evaluated. Some of the illu-

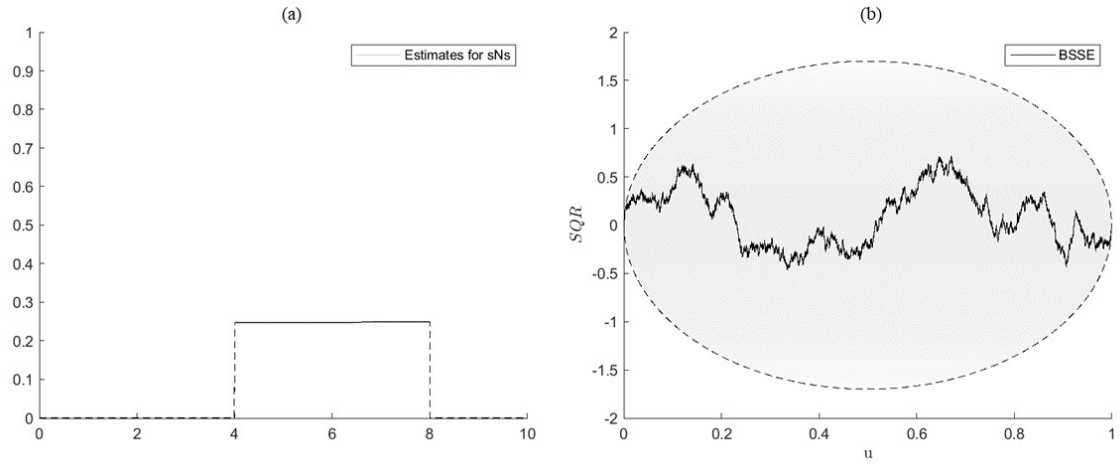

Figure 4.12: $\hat{f}(x)$ determined from averaging 70 pdf estimates from subsamples equal to 60% in size from the original sample of $65,536$ data points. (a) The pdf estimates per subsample and the averge pdf estimate (b) The SQR plot for the average pdf estimate.

minating cases will be immediately discuss, however, other distribution comparisons are available in appendix B.

The five pdf estimators that are evaluated below are the: SE, NMEM estimator, density estimator built into R, bkde estimator in the KernSmooth R package, and kde estimator in the ks R package. The figures showing $\hat{f}(x)$ calculated by the bkde package show two estimates with a gridsize = 406, but one with a bandwidth = 0.05 (blue line) and the other with a bandwidth = 0.25 (black line). Similarly, the figures showing $\hat{f}(x)$ calculated by the density estimator show two estimates, one using Sheather-Jones bandwidth selection [15] (black line) and the other using the Silverman's "rule of thumb" [16] for bandwidth selection (blue line). The kde estimator in the ks package uses a data driven bandwidth selector developed [17]. The true pdf distribution is displayed as the grey line for reference.

The two bandwidth sizes for the bkde estimator were chosen to show how KDE performs at resolving different features of the sample, while the bandwidth selection methods used for the density estimator were picked to show how well KDE performs

without the need for human intervention. As expected, the figures 4.13-B.1 show that in general a larger bandwidth has a hard time resolving fine features in the sample, but is less prone to over fitting. Also, a smaller bandwidth will resolve the fine features, but will over fit and run the risk of resolving erroneous features in the sample. The two data driven bandwidth selection methods used for the density estimator show overall better pdf estimates when compared to just setting a fixed bandwidth for many distributions. This was expected, because the two methods use either the standard deviation or the interquartile range of the sample to determine the bandwidth. In general, the Sheather-Jones bandwidth selection method out preforms Silverman's "rule of thumb" and is able to better resolve features in the sample across many distribution types. The bandwidth selection method employed in the kde estimator produces pdf estimates that in general are somewhere in between the estimates gained from the Silverman's "rule of thumb" and the Sheather-Jones method.

Both the SE and the NEMEM in figure 4.13 (c) and (d) show an excellent estimate of the beta distribution for only 1024 data points, while the KDE methods have difficulty with the divergences. This is due to the selection of the normal distribution as the kernel for these methods. The KDE could be improved with the use of another kernel, but this would require a priori knowledge about the sample.

Similarly to figure 4.13, figure 4.15(c) shows an excellent estimate, however, the NMEM estimator shows wiggles near the divergence, which comes from the use of many Lagrange multipliers to try and resolve the divergence. The NMEM can do better, but the number of Lagrange multipliers is truncated to limit the computational cost. There is no need for this truncation with the use of the NMEM estimator in the SE method, because no block is too large for this to be a problem.

The SE in figure 4.16(c) does a great job of resolving the features in the sample, while being able to handle the extreme outliers that come from the use of the stable distribution; these can typically be on the order of 10^{15} and larger. The NMEM in

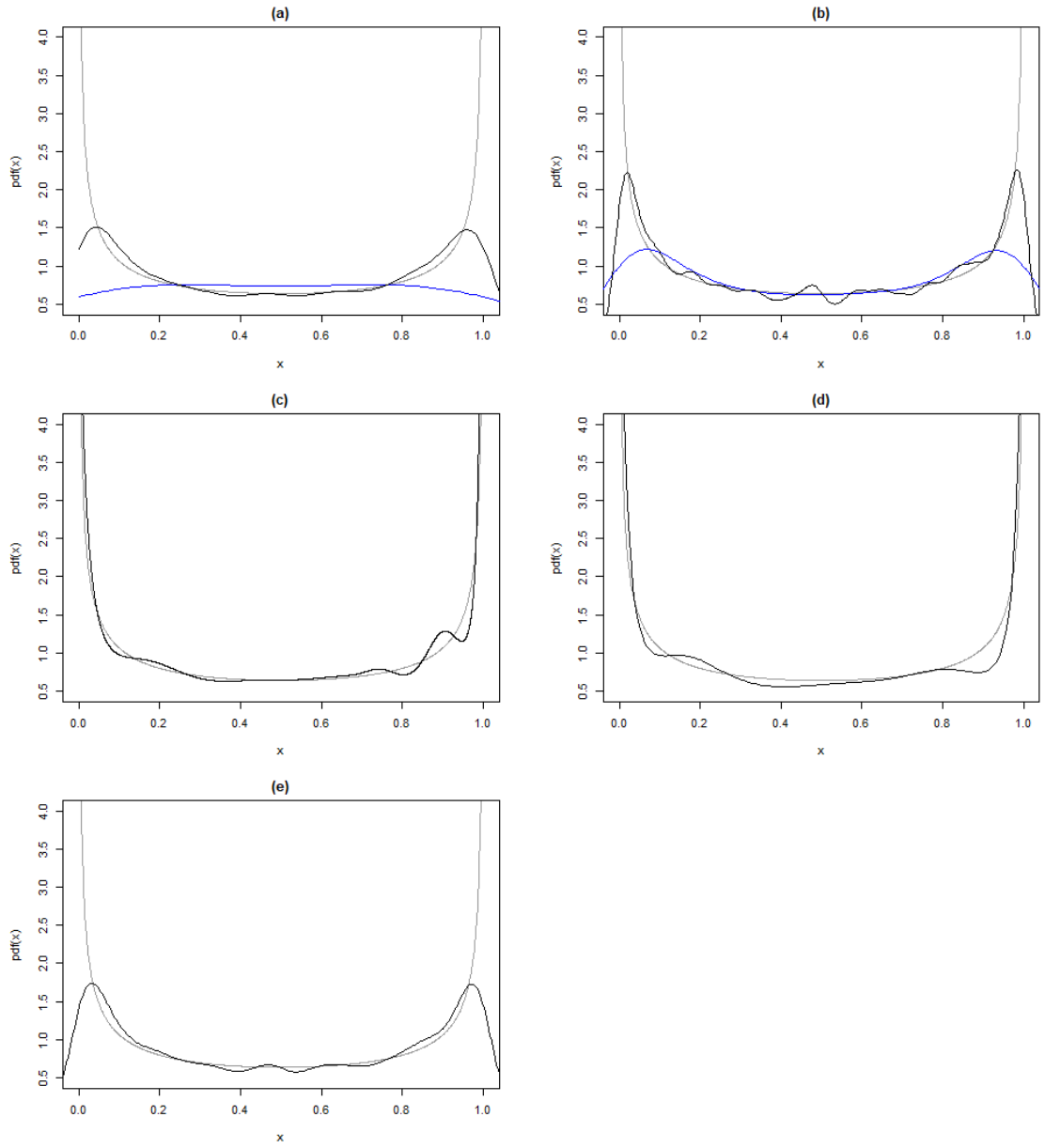

Figure 4.13: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a beta distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

4.16(d) returns a great pdf estimate, but at a slightly lesser resolution compared to the SE. In figures 4.16(b) and (e) the pdf estimate completely fails to resolve the sample's features. This is due to the stable distribution theoretically having infinite variance and an extremely large variance numerically. This causes the data driven

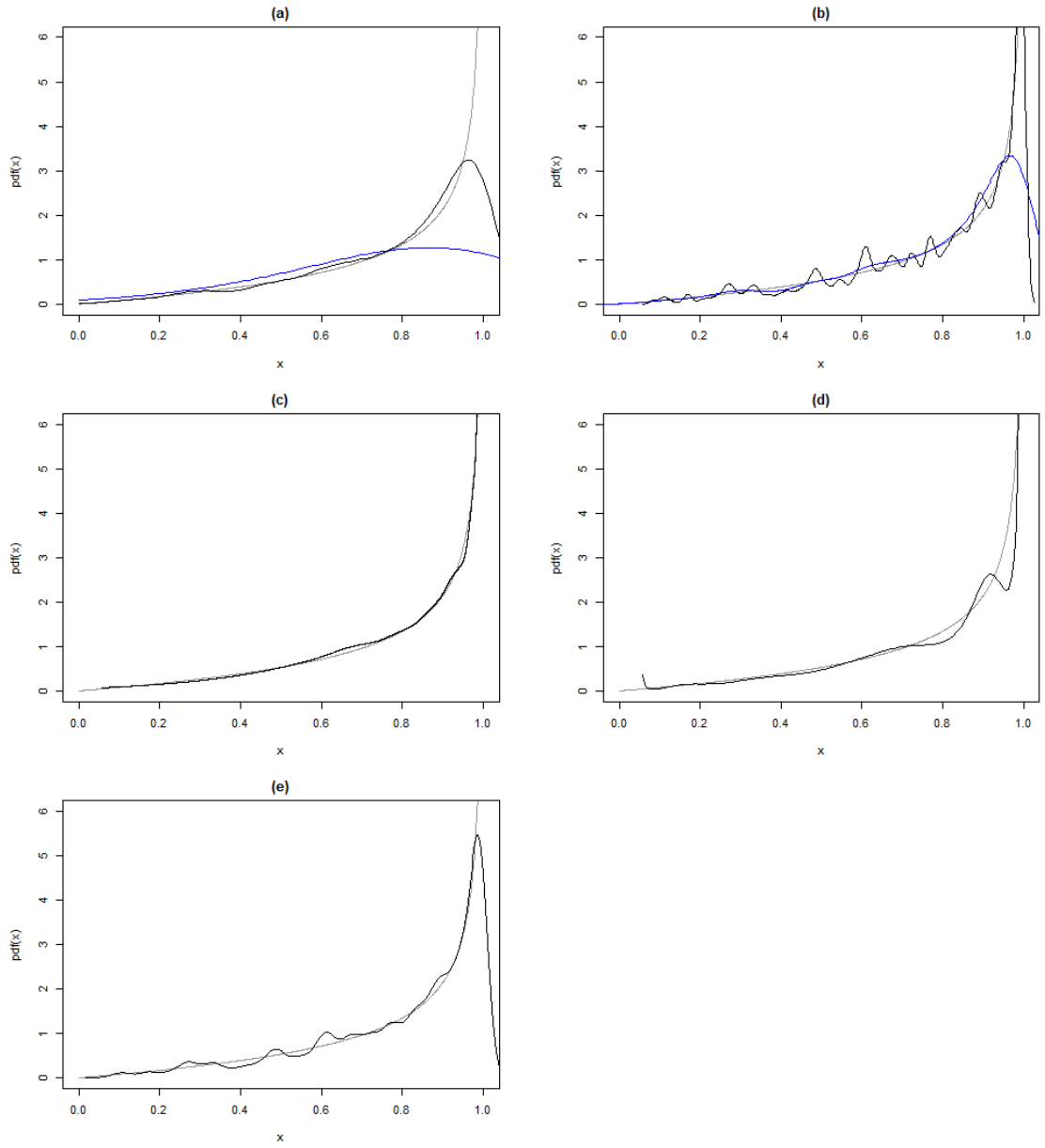

Figure 4.14: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a beta distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

bandwidth selection methods to improperly set the bandwidths to be much too large. On the other hand, the fixed bandwidths in 4.16(a) return pdf estimates that resolve the features well.

Similar to the case in figure 4.16, the generalized extreme value distribution shown

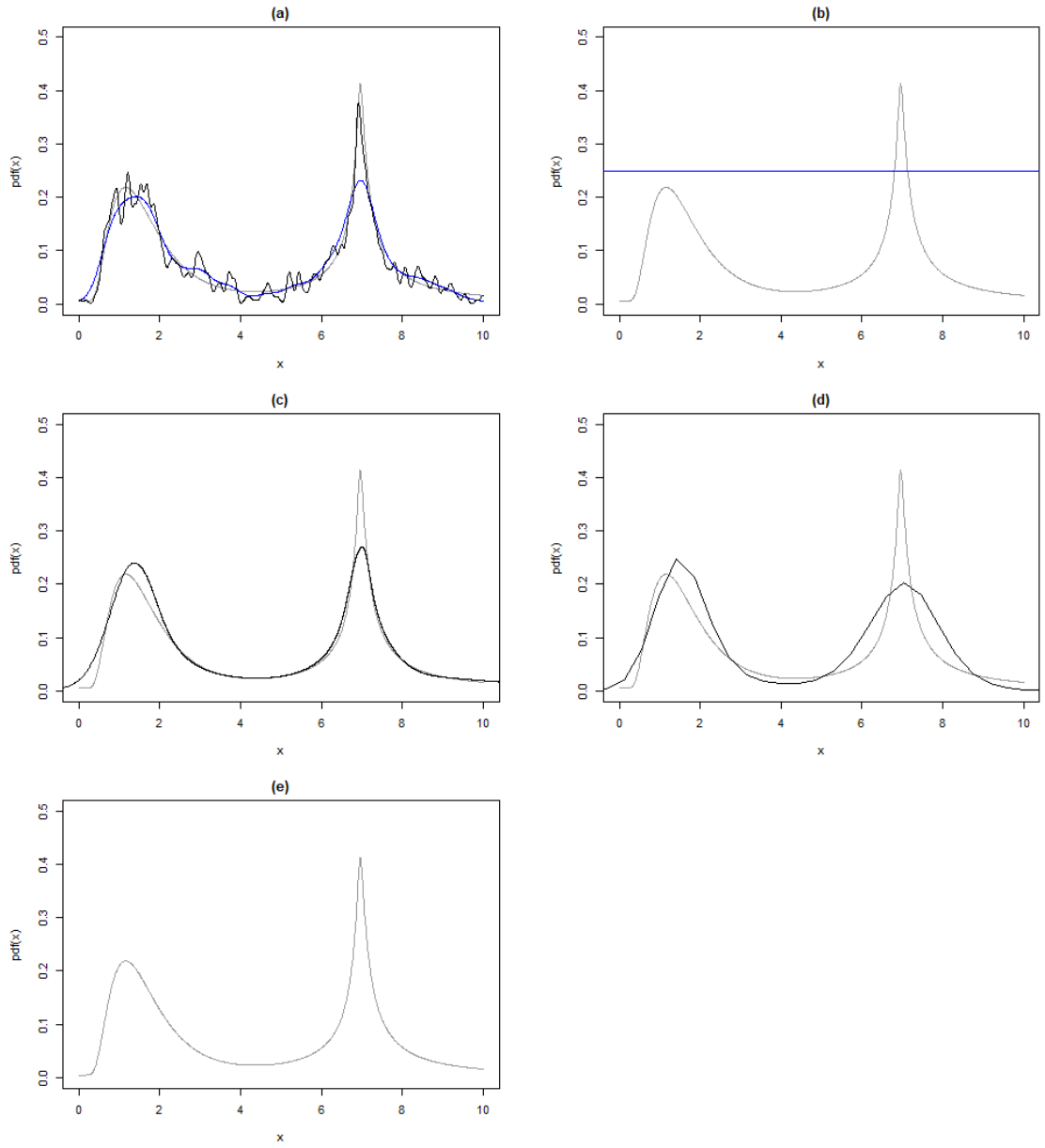

Figure 4.15: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a mixture model of a Birnbaum-Saunders and stable distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator. Both the density estimator and the kde estimator fail to return adequate pdf estimates, and sometimes fail completely as seen for the kde estimator.

in figure 4.19 leads to the same failures for the data driven bandwidth selection methods used in KDE. Both the SE and the NMEM estimator return excellent results,

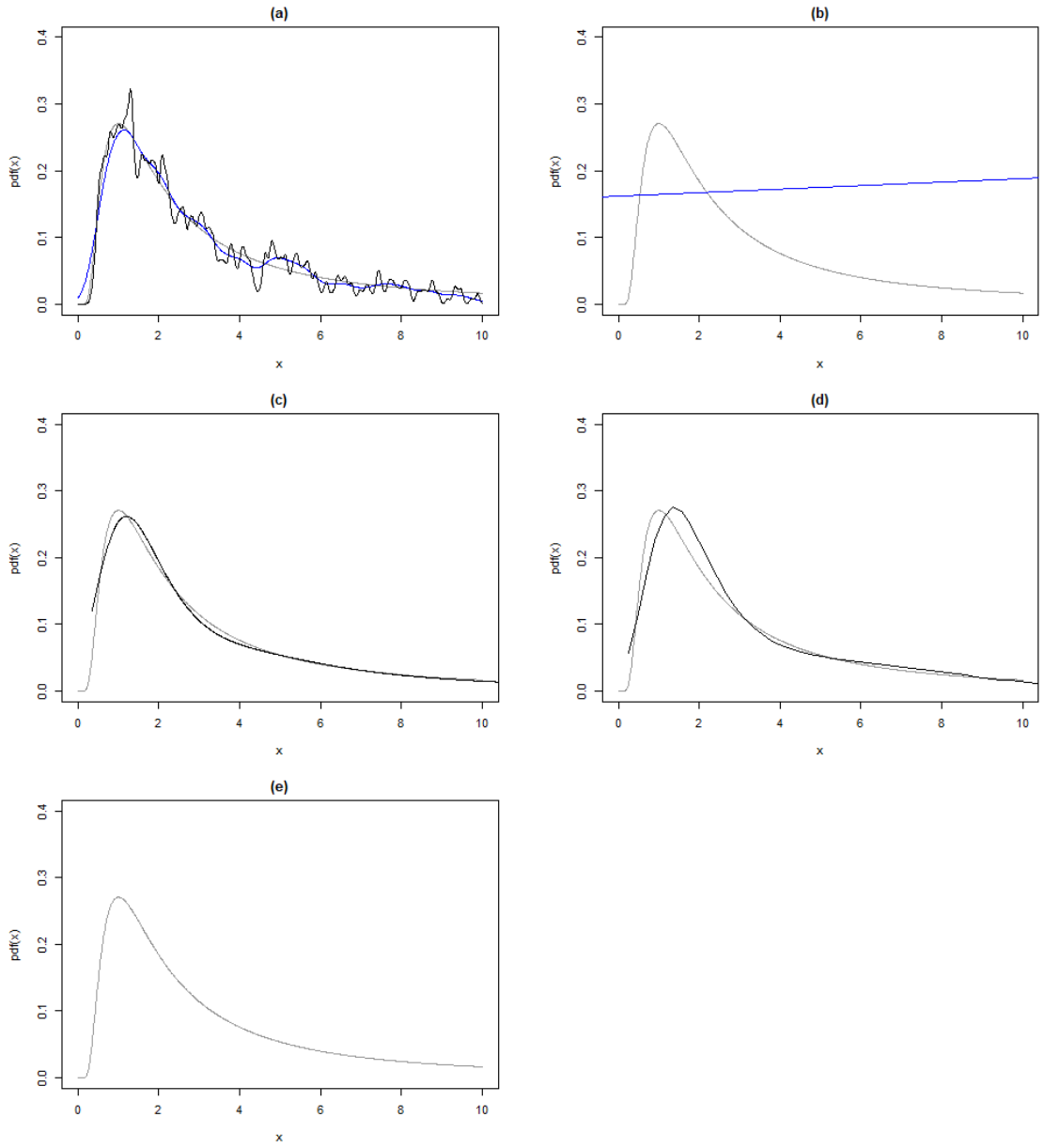

Figure 4.16: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a generalized extreme value distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator. Both the density estimator and the kde estimator fail to return adequate pdf estimates, and sometimes fail completely as seen for the kde estimator.

and the fixed bandwidth estimator does the same for a larger bandwidth. however, the smaller bandwidth introduces high variance into the estimate.

The KDE estimators have the same qualitative behavior as in the cases shown in

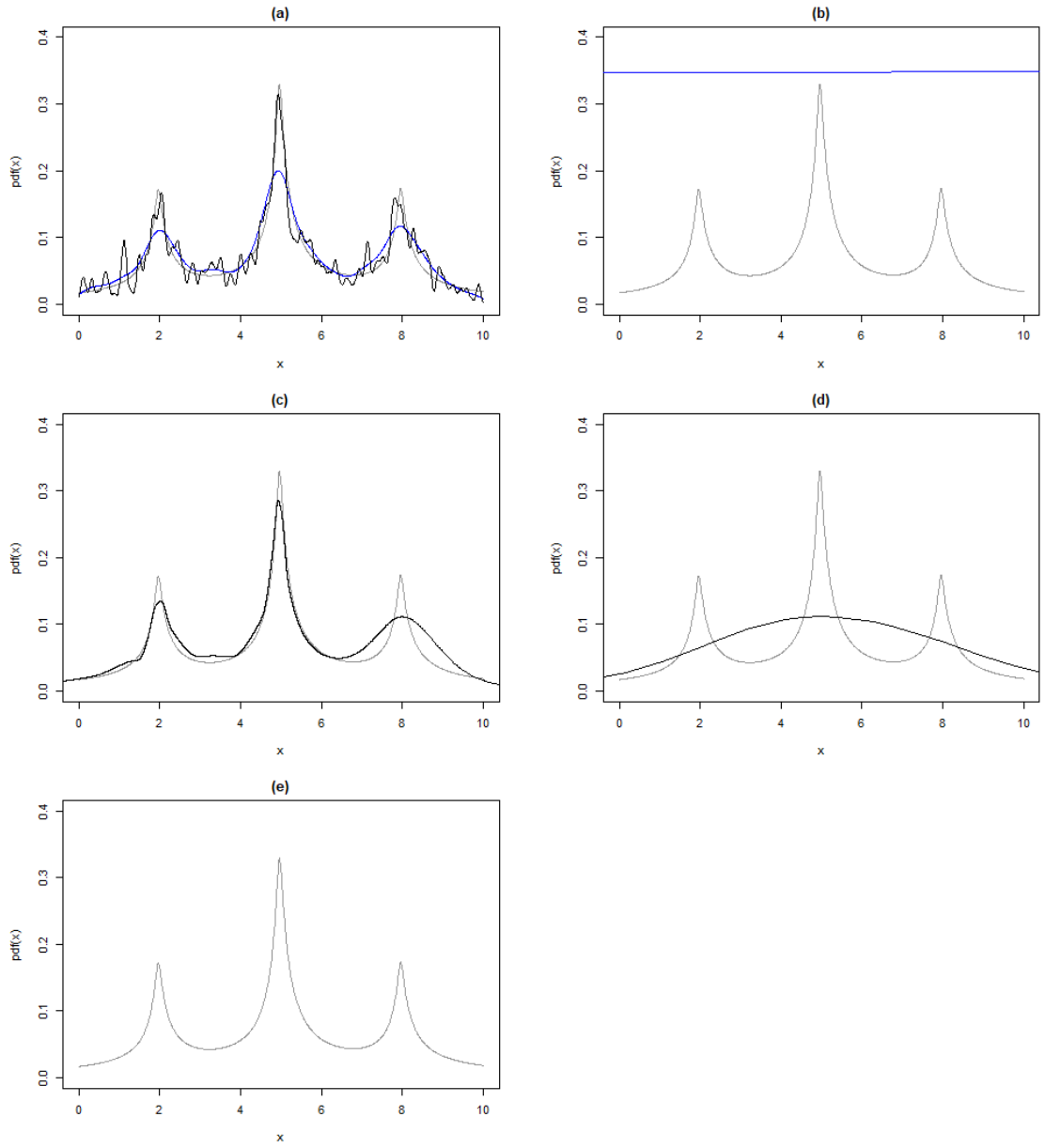

Figure 4.17: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a stable distribution with the parameters given in the code in appendix C.2 under the name "Stable3". (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator. Both the density estimator and the kde estimator fail to return adequate pdf estimates, and sometimes fail completely as seen for the kde estimator.

figure 4.16 and figure 4.19 for the case shown in figure 4.17. This is again from the large variance of the the samples created from these distributions. With the mixture model

of three stable distributions the variance of the sample numerically is significantly larger that the previous two cases. This causes the NMEM estimator's pdf estimate shown in figure 4.17(d) to be heavily smoothed. The smoothing arises from the need of many Lagrange multipliers to force a smooth (and nearly zero) estimate far out onto the tails of the distribution, which makes resolving the sharp center features difficult. The NMEM can be improved if the maximum number of Lagrange multipliers was increased, but this would increase computation cost. The SE estimator shown in figure 4.17(c) does a great job of resolving the features, however, does have a slight difficulty resolving the edge modes as being identical to one another.

In figure B.7, the KDE estimators do a great job of resolving the majority of the features, however, have difficulty with the edges due to the use of the normal distribution kernel. Also, as expected the smaller fixed bandwidth KDE estimator has high variance in the pdf estimate. The SE does a great job of capturing the sharp edges of the uniform distribution, however, can pick up on random fluctuations in the sample. The NMEM estimator returns an excellent pdf estimate for only 1024 data points.

As to be expected, in figure 4.19(a) the large bandwidth has difficulty resolving the sharp edges of the uniform distributions, while the small bandwidth resolves the edges, but high variance. The data driven bandwidth selection methods preform quite well, except for Silverman's "rule of thumb". This is due to the bandwidth in that method being depended solely on the sample size and standard deviation of the sample. In figure 4.19(d) the NMEM estimator shows to have difficulty consistently resolving all of the features as being uniform compared to one another. The SE in 4.19(c) also exhibits this difficulty but to a lesser degree. However, as sample size increases the features of the data are able to be nicely resolved as seen in figure 4.20.

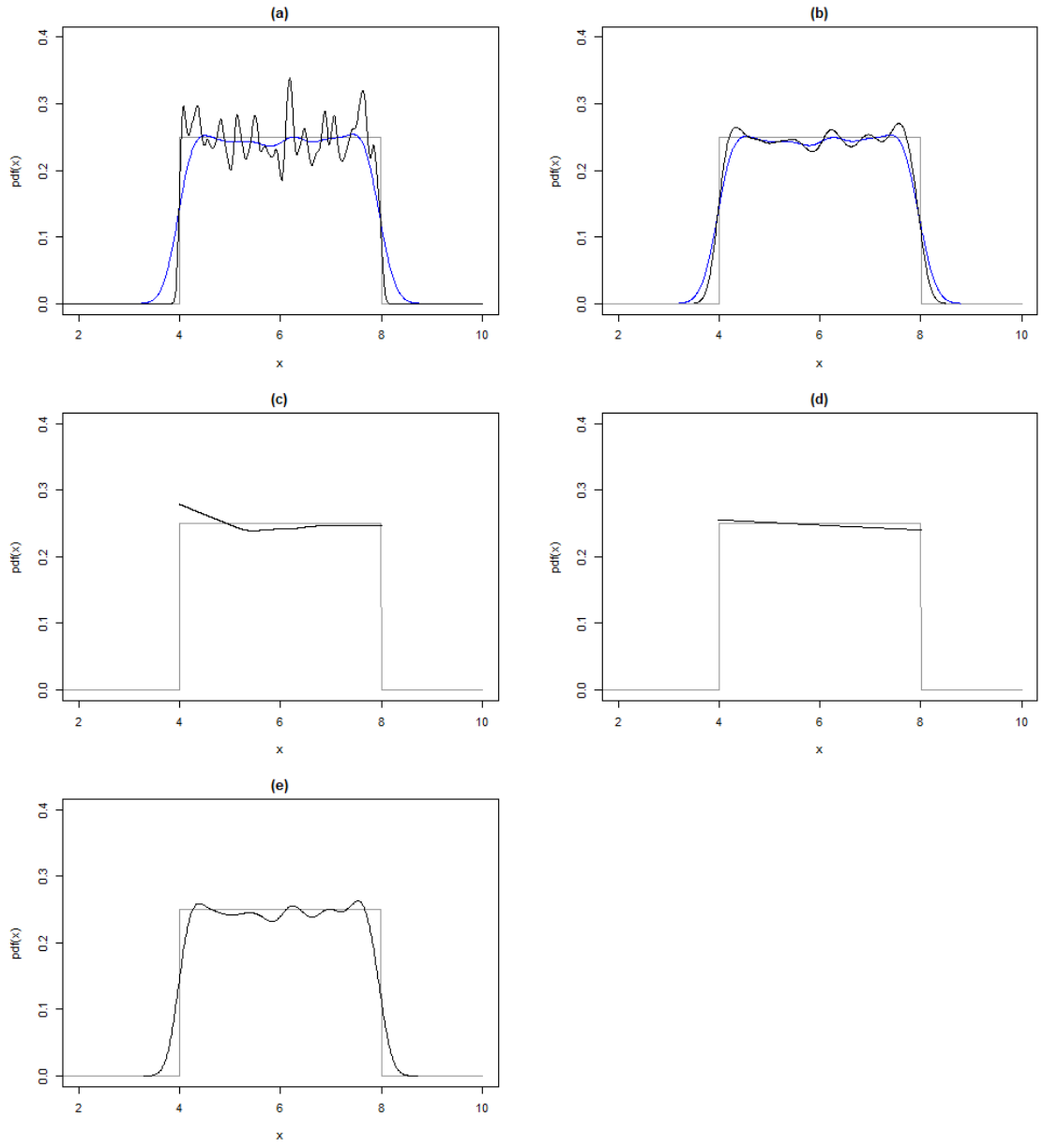

Figure 4.18: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a uniform distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

4.3 Computation time

To evaluate the efficiency of the SE method pdf estimates for the six distributions in figure 4.21 were obtained for a variety of sample sizes. The time of computation was calculated using the tic/toc and cputime functions. The cputime function returns the

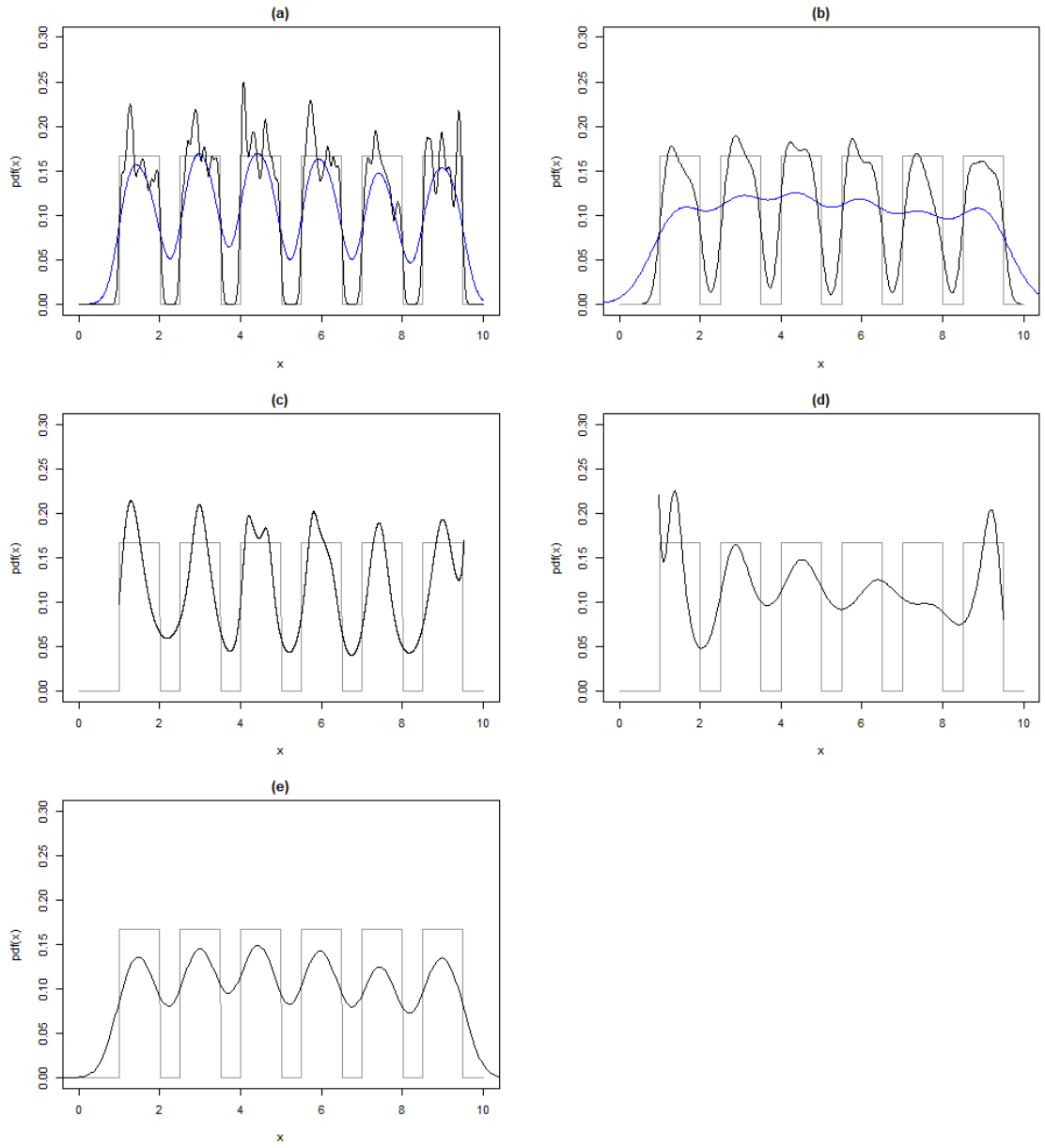

Figure 4.19: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a mixture model of six uniform distributions with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

elapsed CPU time which sums across all threads, while the tic/toc function returns the wall-clock time. For the computational efficiency analysis of the SE method 4 threads were used and the background processes for the machine were kept at a minimum. The SE method may be sped up significantly from the analysis below depending on

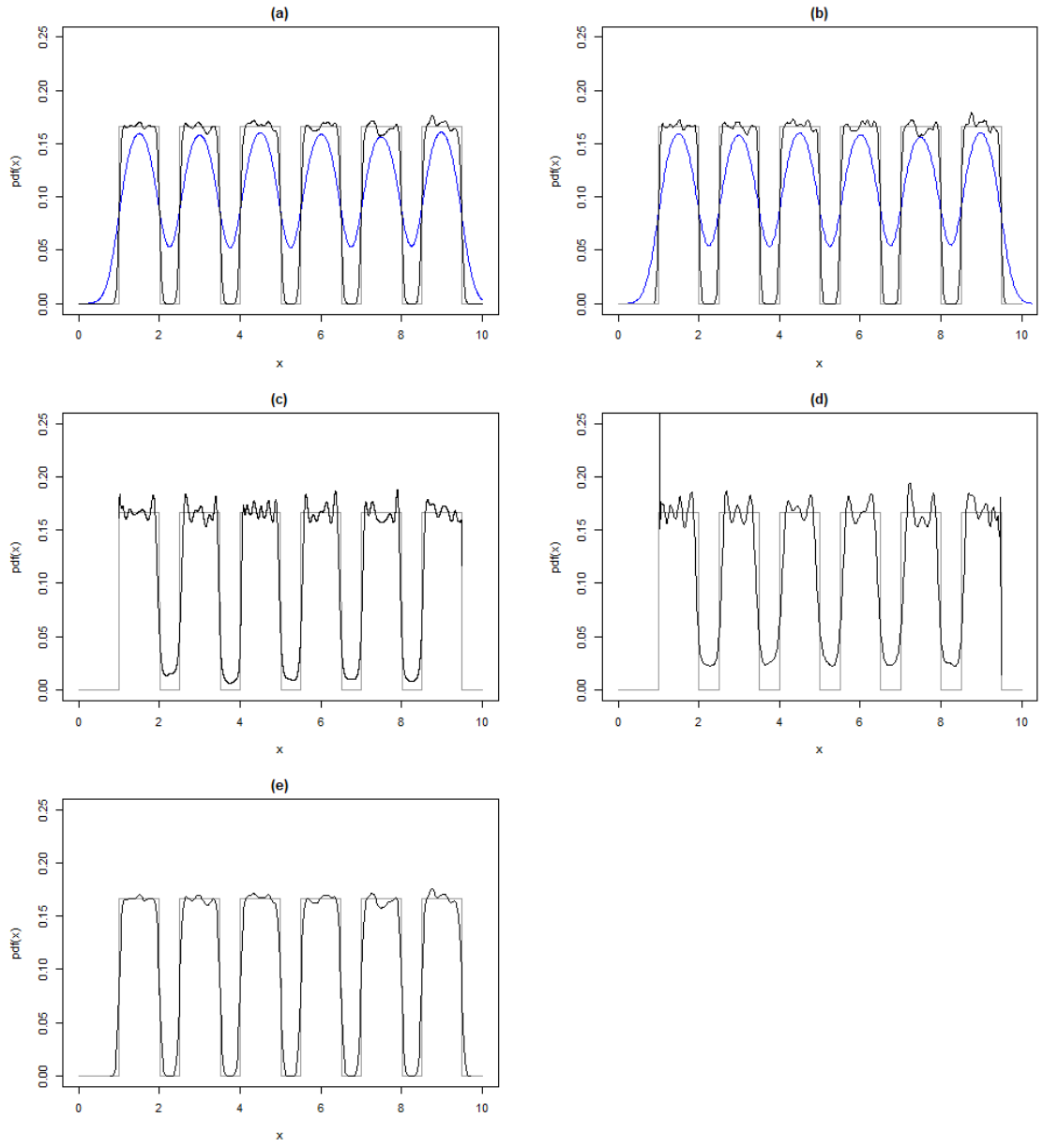

Figure 4.20: $\hat{f}(x)$ for a sample of size $N = 65,536$ from a mixture model of six uniform distributions with the parameters given in the code in appendix C.2 under the name "Uniform-Mix". (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

the userâs available computer hardware and ability to add more threads to compute pdf estimates per block.

The computation time returned for the tic/toc function in figure 4.22(b) in general shows a linear behavior on the log-log plot, however, the wall-clock times for pdf

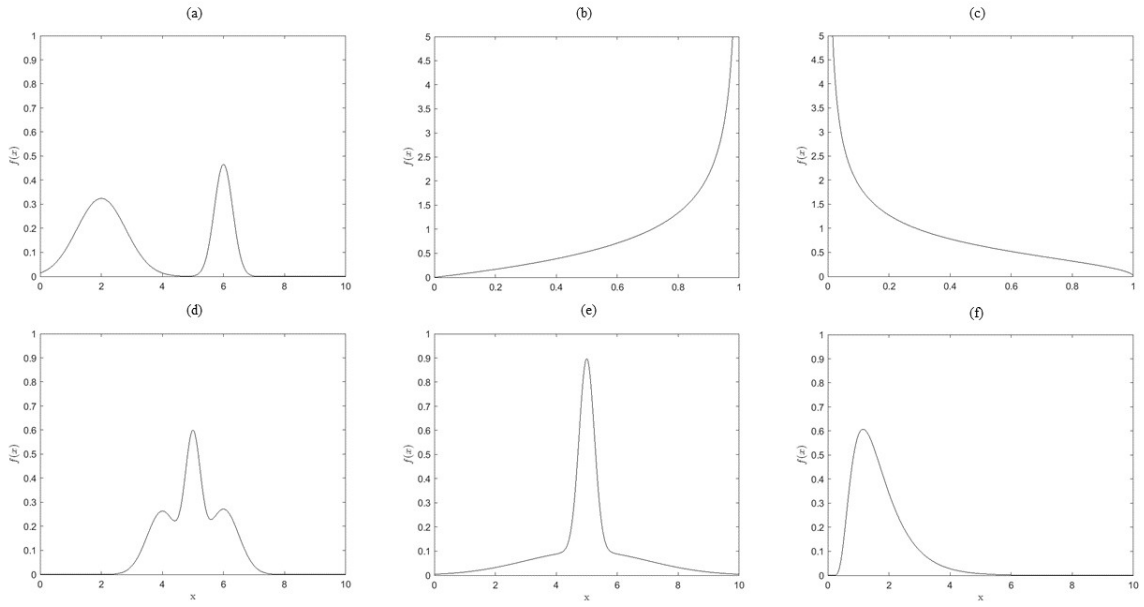

Figure 4.21: Six distributions used for the computational efficiency of the SE method. (a) Bimodal distribution (b) Beta distribution $a = 2$, $b = 0.5$ (c) Beta distribution $a = 0.5$, $b = 1.5$ (d) Trimodal normal distribution (e) Contaminated normal distribution (f) Birnbaum Saunders distribution.

estimates from the two beta distributions have a noticeably larger slope. This increase in wall-clock time for these two distributions is to be expected. The reason for the larger slope is caused by the number of blocks being too small for the size of the sample and the difficulty of the distributions. The divergences in the two beta distributions are computational expensive for the NMEM, because it requires many more Lagrange multipliers to estimate the divergent regions.

The computation times returned for the cputime function in figure 4.22(a) approaches the same slope of approximately $\frac{3}{4}$ as the sample size increases on the log-log plot. The asymptotic behavior for the CPU times seen in figure 4.22(a) is from the NMEM approaching a linear time dependence to N for large samples, but the SE doesn't approach the N time dependence as rapidly as the NMEM estimator; this is due to the partitioning of the sample into blocks. This is confirmed in figures 4.24(a) and 4.23(a) which exhibit the same asymptotic behavior even though the overall times decrease compared to 4.22(a) as well as the rate at which the SE approaches a linear

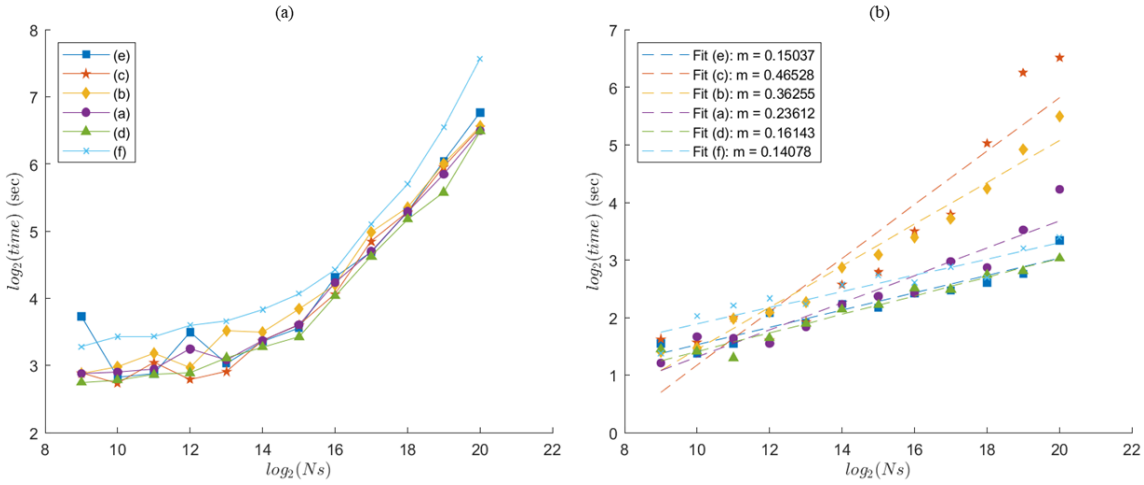

Figure 4.22: The computational times for the SE method which uses the random search minimization code to minimize $\Delta\xi$. (a) The CPU times (b) The wall-clock times for sample sizes 2^9 up to 2^{20}.

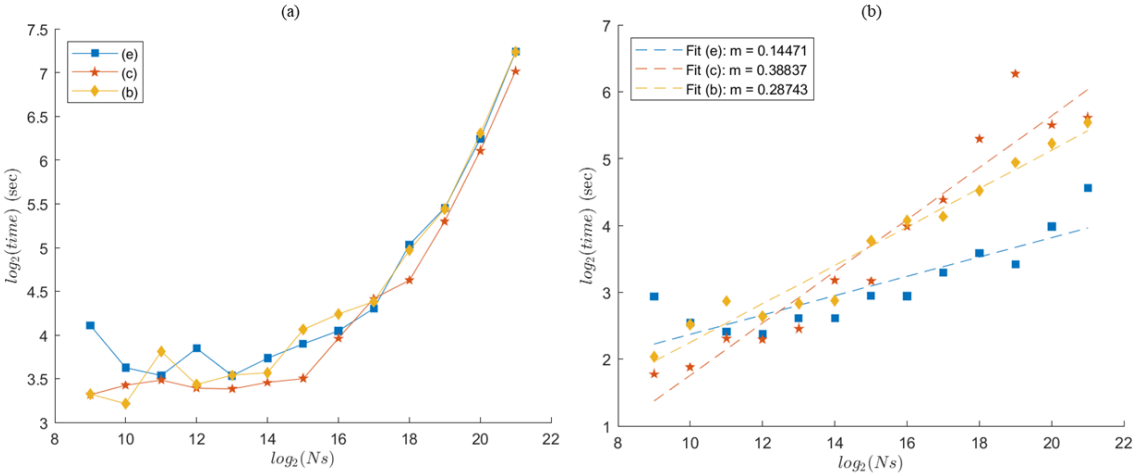

Figure 4.23: The computational times for the SE method which uses the golden ratio bifurcation search minimization code to minimize $\Delta\xi$ and with maximum block sizes of $100,000$. (a) The CPU times (b) The wall-clock times for sample sizes 2^9 up to 2^{21}.

time dependence to N.

While NMEM can handle large block sizes and the results are good, forcing a maximum block size enables even faster estimates from the NMEM. Figure 4.23(a) and (b) shows that by requiring all blocks to not exceed $100,000$ data points that the wall-clock and CPU times can be reduced. To further improve the over wall-clock

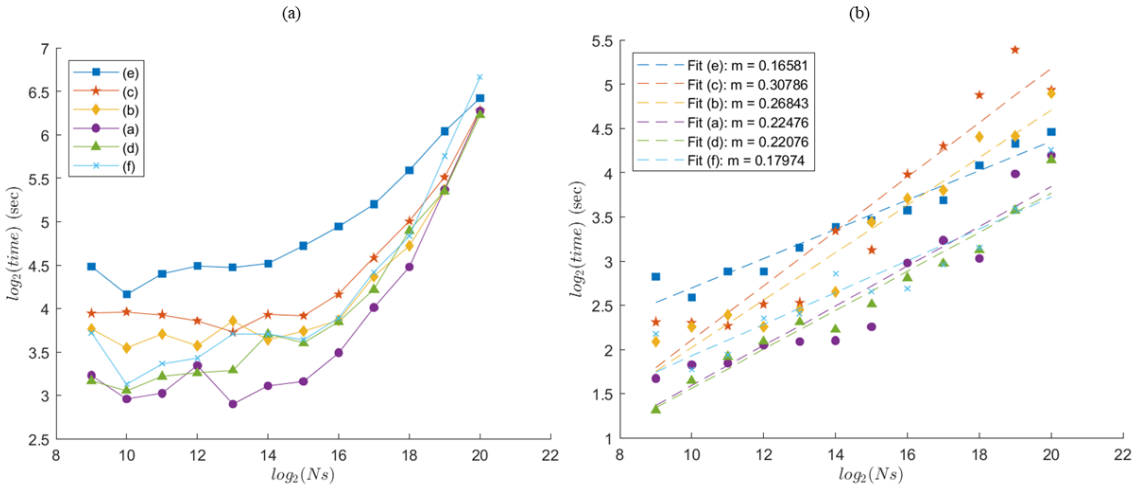

Figure 4.24: The computational times for the SE method which uses the golden ratio bifurcation search minimization code to minimize $\Delta\xi$ and with maximum block sizes of $50,000$. (a) The CPU times (b) The wall-clock times for sample sizes 2^9 up to 2^{20}.

and CPU times of the SE for all distributions, the maximum block size was limited to $50,000$, which is shown in figure 4.24(a). Doing this substantially improved the efficiency of the SE.

Overall, the SE method shows to be computationally efficient. This conclusion is reach since in the cases discussed so far all of the log-log slopes for the (b) figures have a slope less than $\frac{1}{2}$ for the range of sample sizes explored. This means the wall-clock time scales as a power less than $\frac{1}{2}$ compared to sample size. This time dependent relationship is due to the NMEM being less than linearly dependent to N for small samples. If larger samples then 2^{20} were explored it would be expected that the wall-clock times would approach a linear dependence to N. Similarly, in the large sample region the slope of the (a) figures are close to one, which means the CPU time scales approximately linearly with sample size, which is caused from the underlying NMEM estimator.

CHAPTER 5: CONCLUSIONS

The SE method has shown to be a robust nonparametric pdf estimator which has improved upon the work that developed the NMEM. Also, the SE has shown to be a good estimator for high throughput applications for both small and large sample sizes and is largely limited by the available number of processors one has to run a parallel job. The use of the R-ratio method for determining the appropriate number of blocks and block sizes has shown to be an elegant data driven approach. In addition, the sub-sampling sampling method for obtaining pdf estimates yields estimates that do not over fit to the sample under consideration, which has been confirmed by the random fluctuations in the SQR plots for small or large samples. Lastly, the SE can consistently calculate good pdf estimates over many trials of the same sample size, which was confirmed by evaluating the average ME and SQR range for 1000 trials of a sample of 512 data points.

In the future, the SE's code will be improved to perform better with memory usage and ensure the fasted computation operations are being used. The SE can be further improved by writing the algorithm in a programming language that has better multithreading capabilities, such as, C++ or Python. Also, the heuristically determined parameters used to define the threshold Γ will be studied more extensively by running batches of the SE code varying one parameter at a time. This will give a deep insight into the effect each parameter has of the quality of the pdf estimates.

REFERENCES

[1] J. Farmer and D. Jacobs, "High throughput nonparametric probability density estimation.(research article)(report)," *PLoS ONE*, vol. 13, no. 5, p. e0196937, 2018.

[2] L. Yu and Z. Su, "Application of kernel density estimation in lamb wave-based damage detection," *Mathematical Problems in Engineering*, vol. 2012, no. 2012, 2012.

[3] M. J. Baxter, C. C. Beardah, and S. Westwood, "Sample size and related issues in the analysis of lead isotope data," *Journal of Archaeological Science*, vol. 27, no. 10, pp. 973–980, 2000.

[4] J. DiNardo, N. M. Fortin, and T. Lemieux, "Labor market institutions and the distribution of wages, 1973-1992: a semiparametric approach," *Econometrica*, vol. 64, no. 5, p. 1001, 1996.

[5] A. Gramacki, *Nonparametric Kernel Density Estimation and Its Computational Aspects*. Studies in Big Data, 37, Cham: Springer International Publishing, 2018.

[6] I. Narsky, "Statistical analysis techniques in particle physics : fits, density estimation and supervised learning," 2014.

[7] G. R. Terrell and D. W. Scott, "Variable kernel density estimation," *The Annals of Statistics*, vol. 20, no. 3, pp. 1236–1265, 1992.

[8] B. U. Park and J. S. Marron, "Comparison of data-driven bandwidth selectors," *Journal of the American Statistical Association*, vol. 85, no. 409, pp. 66–72, 1990.

[9] T. Keith Yuan Patarroyo, "Mean conservation for density estimation via diffusion using the finite element method," *BoletiÌn de MatemaÌticas*, vol. 24, no. 1, pp. 91–99, 2017.

[10] Z. Botev, J. Grotowski, and D. Kroese, "Kernel density estimation via diffusion," *arXiv.org*, vol. 38, no. 5, 2010.

[11] I. Sadeh, F. B. Abdalla, and O. Lahav, "Annz2: photometric redshift and probability distribution function estimation using machine learning," *Publications of the Astronomical Society of the Pacific*, vol. 128, no. 968, p. 104502, 2016.

[12] V. N. Vapnik, "An overview of statistical learning theory," *Neural Networks, IEEE Transactions on*, vol. 10, no. 5, pp. 988–999, 1999.

[13] D. J. Jacobs, "Best probability density function for random sampled data," *Entropy (Basel, Switzerland)*, vol. 11, no. 4, p. 1001, 2009.

[14] J. Farmer and D. J. Jacobs, "Nonparametric maximum entropy probability density estimation," 2016.

[15] S. J. Sheather and M. C. Jones, "A reliable dataâbased bandwidth selection method for kernel density estimation," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 53, no. 3, pp. 683–690, 1991.

[16] C. D. Kemp and B. W. Silverman, "Density estimation for statistics and data analysis," *The Statistician*, vol. 36, no. 4, p. 420, 1987.

[17] M. P. Wand and M. C. Jones, *Kernel smoothing*. Monographs on statistics and applied probability, London ; New York: Chapman Hall, 1st ed., 1995.

APPENDIX A: Further stitching estimator examples

The figures that follow are further examples of the SE estimator. These figures were omitted from the body of the thesis, but contain useful information about capabilities of the SE. The examples that follow do not employ the subsampling procedure.

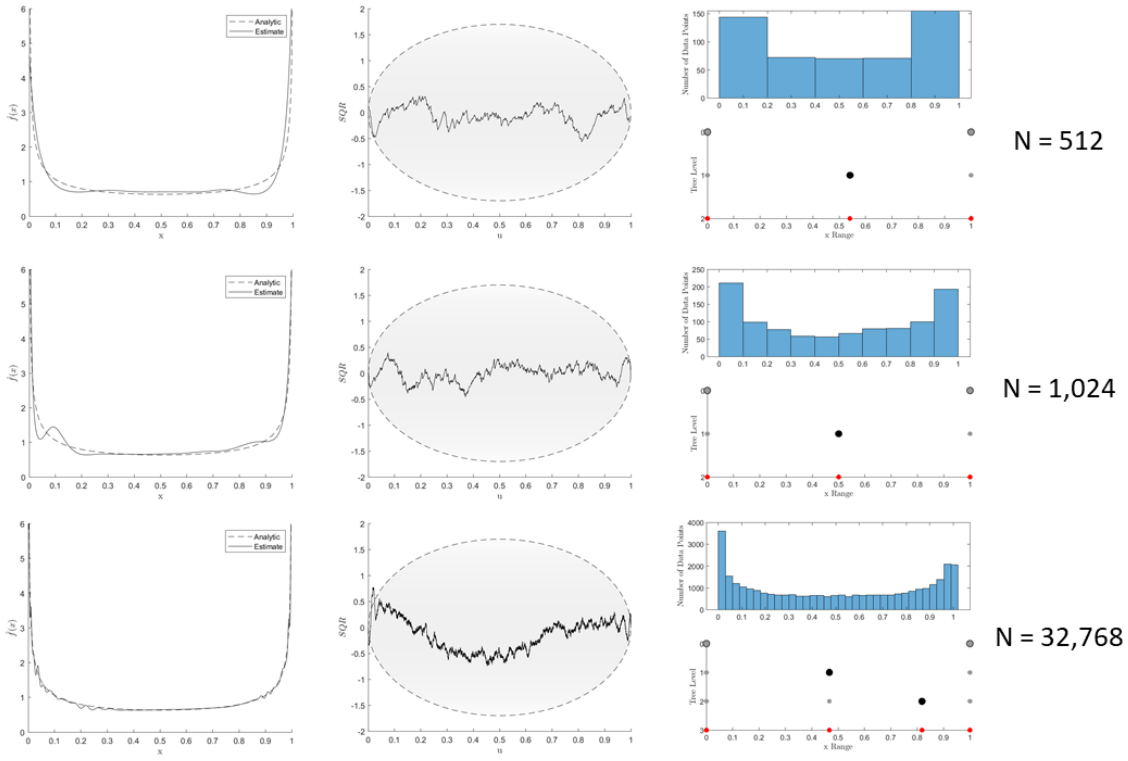

Figure A.1: Estimate for a beta distribution with $a = 0.5$ and $b = 0.5$.

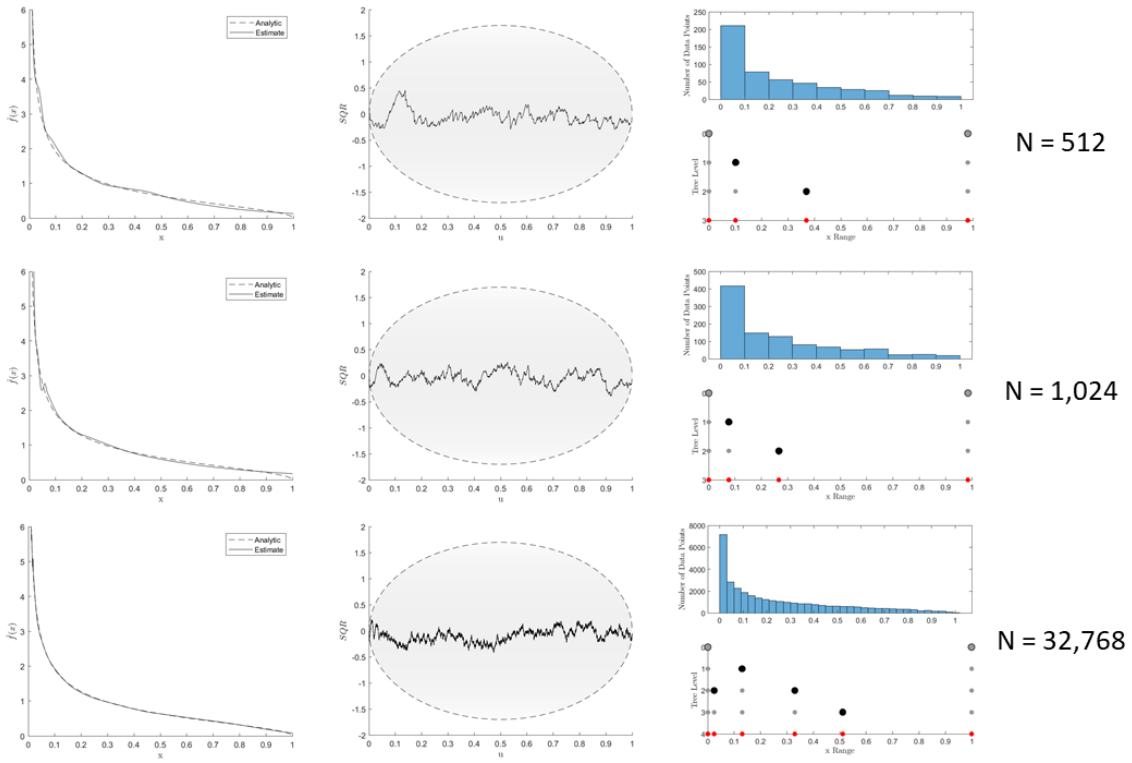

Figure A.2: Estimate for a beta distribution with $a = 1.5$ and $b = 0.5$.

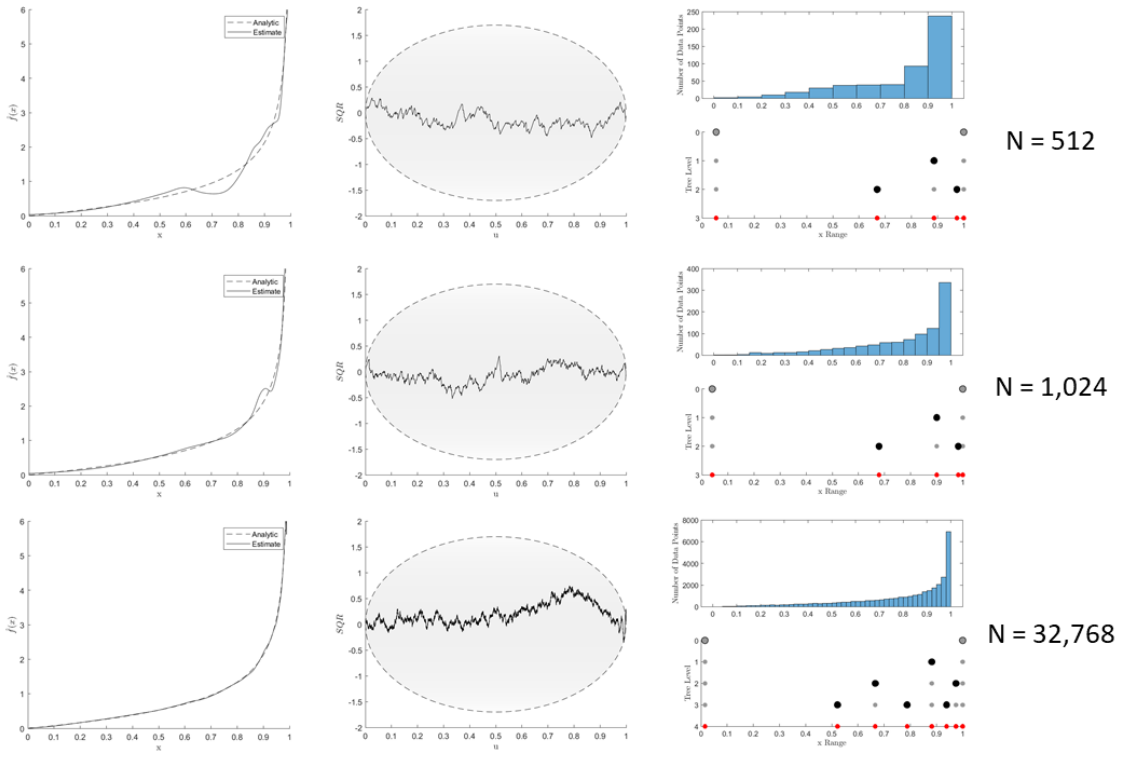

Figure A.3: Estimate for a beta distribution with $a = 2$ and $b = 0.5$.

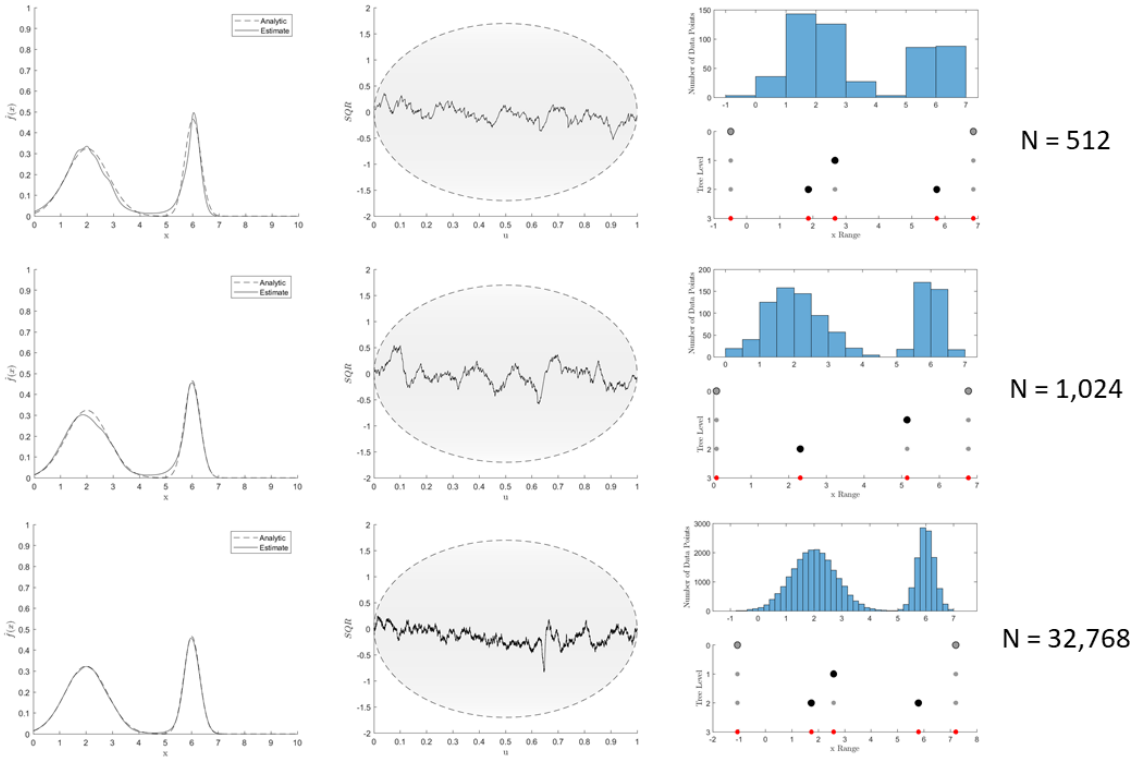

Figure A.4: Estimate for a bimodal normal distribution.

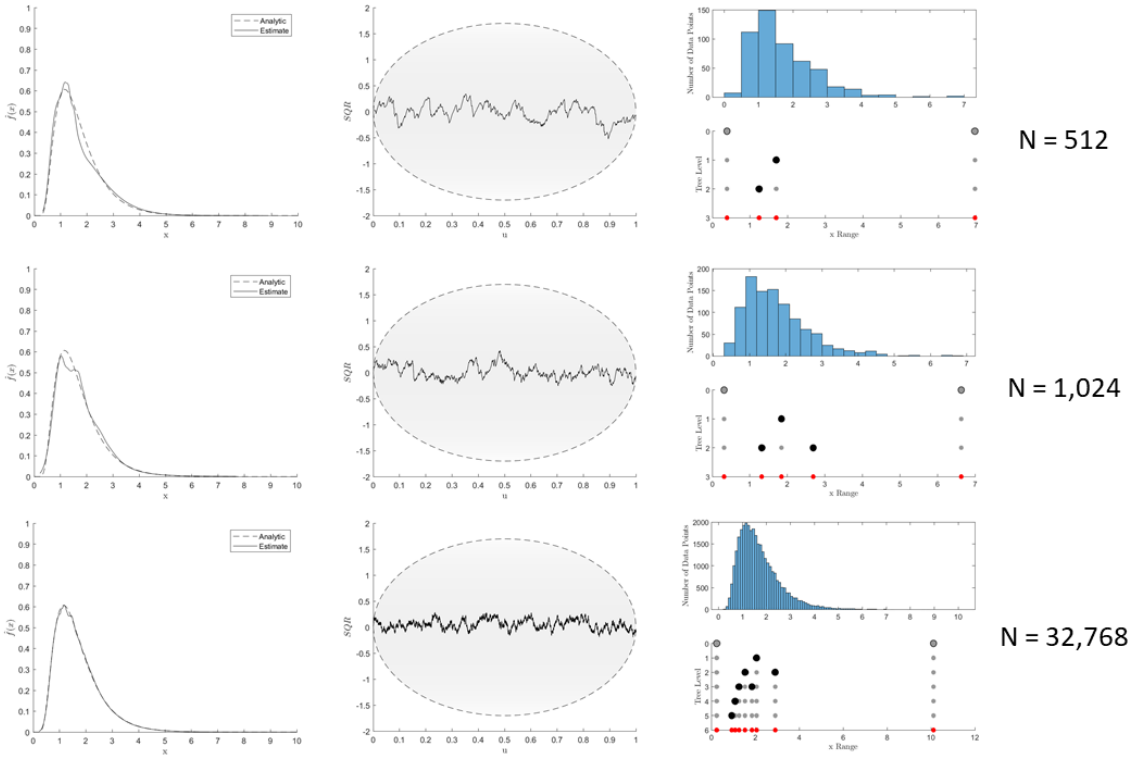

Figure A.5: Estimate for a Birnbaum Saunders distribution.

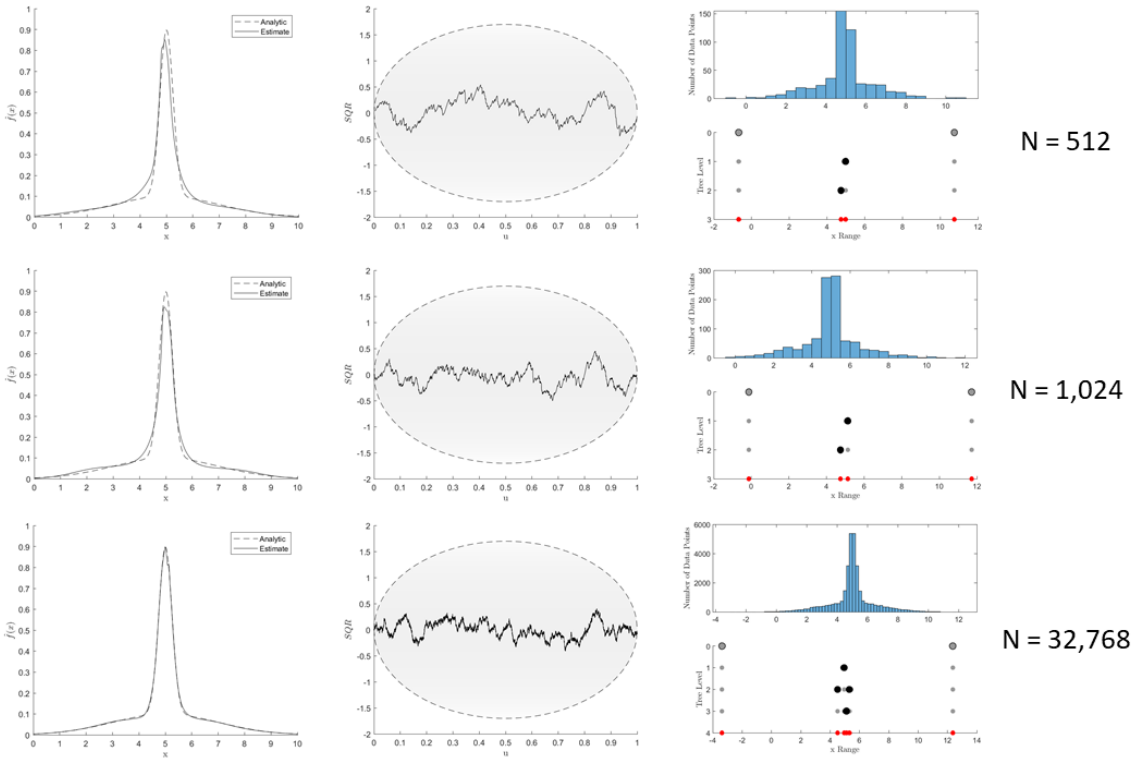

Figure A.6: Estimate for a contaminated normal distribution.

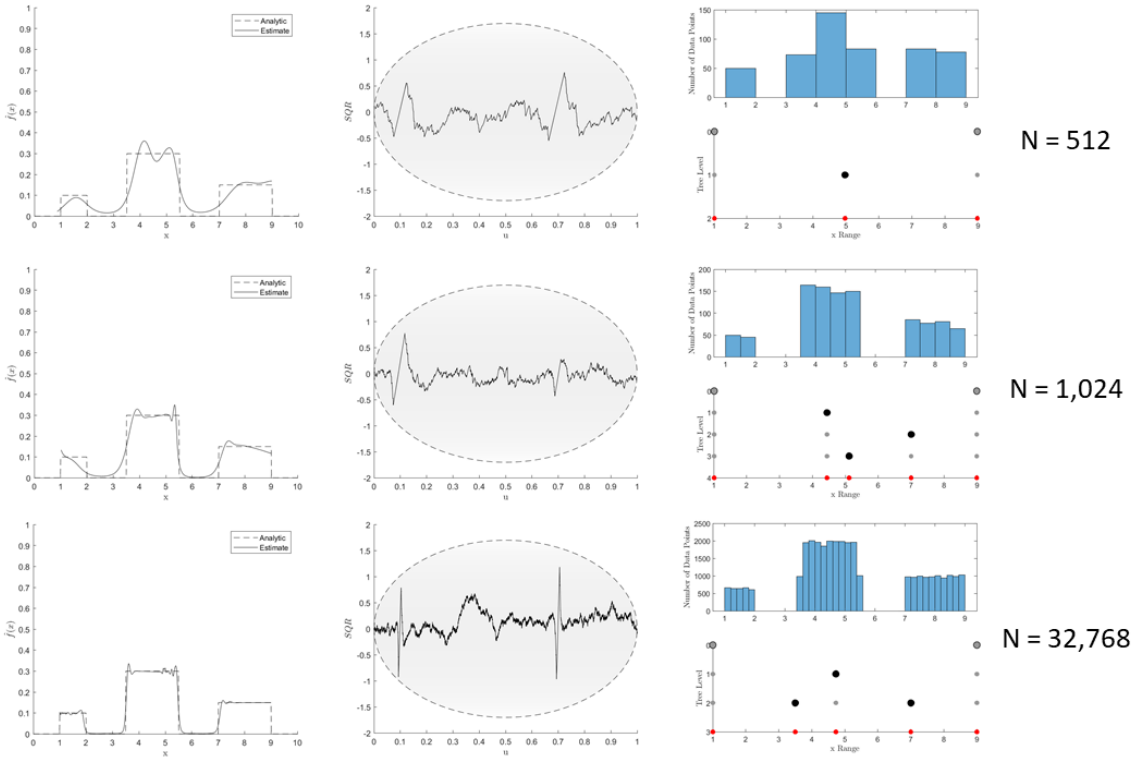

Figure A.7: Estimate for a mixture model created from uniform distributions.

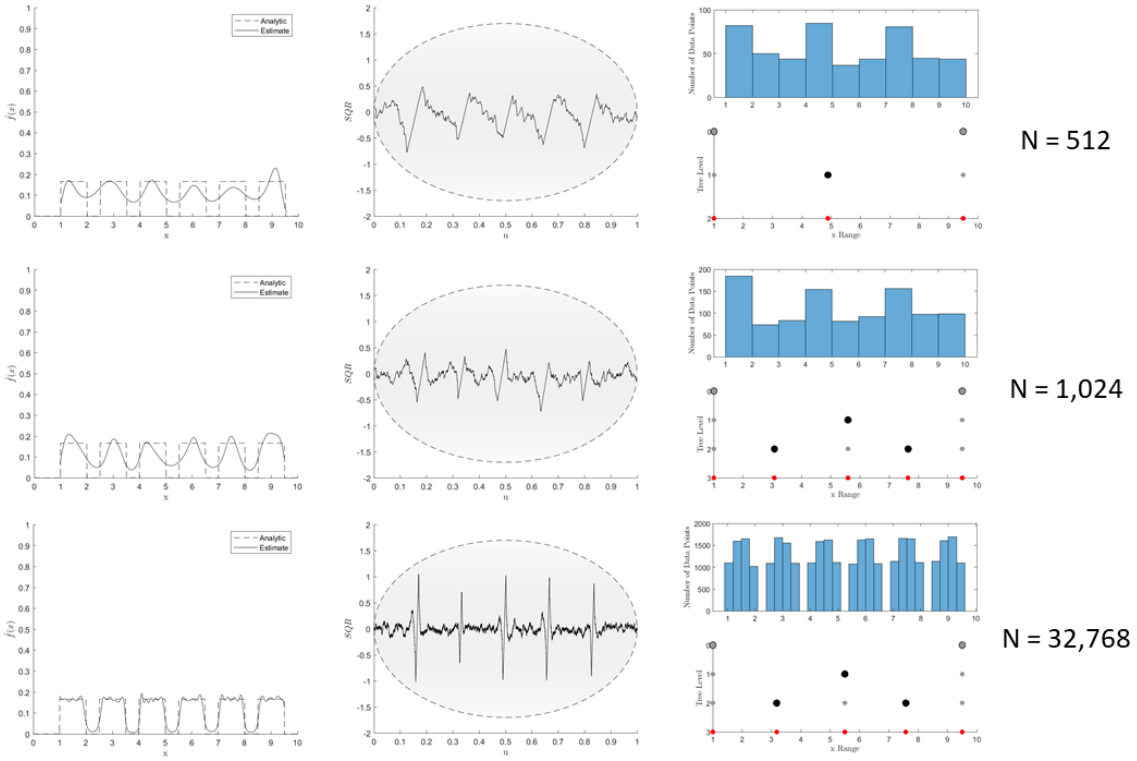

Figure A.8: Estimate for a mixture model created from uniform distributions.

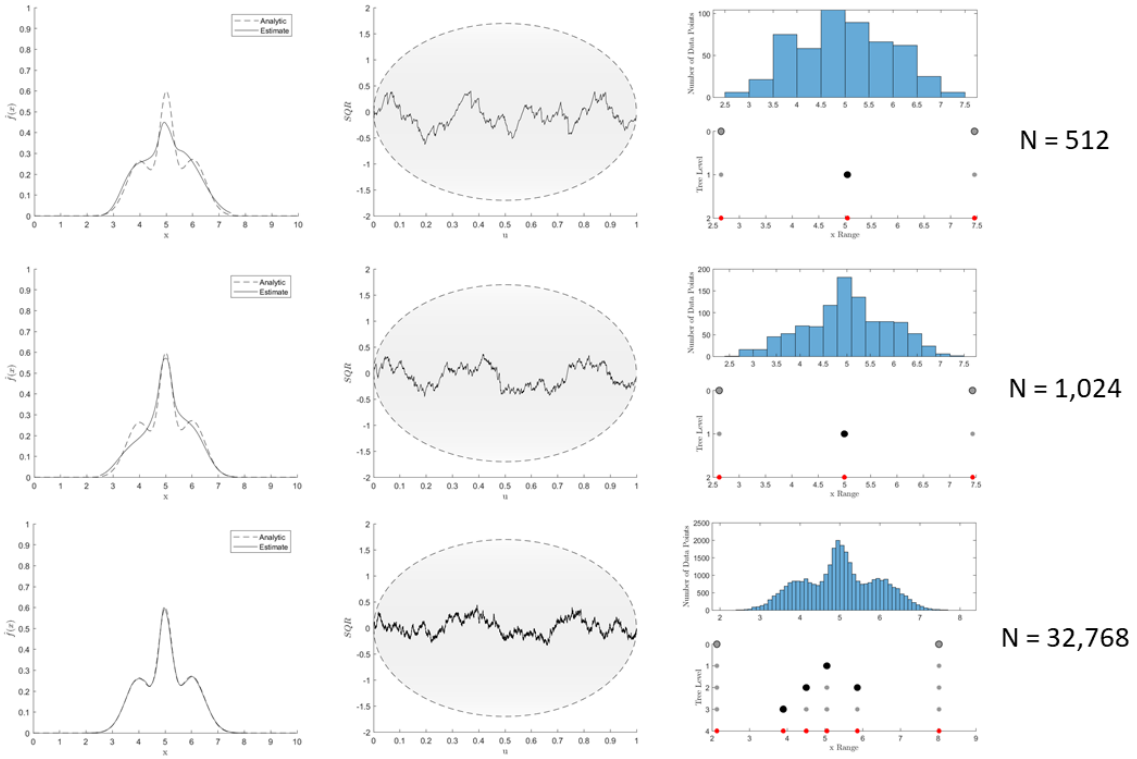

Figure A.9: Estimate for a trimodal normal distribution.

APPENDIX B: Further pdf estimator comparisons

The figures that follow are further comparisons of the SE estimator to KDE methods and the NMEM estimator. These figures were omitted from the body of the thesis but contain useful information about all of the pdf estimators.

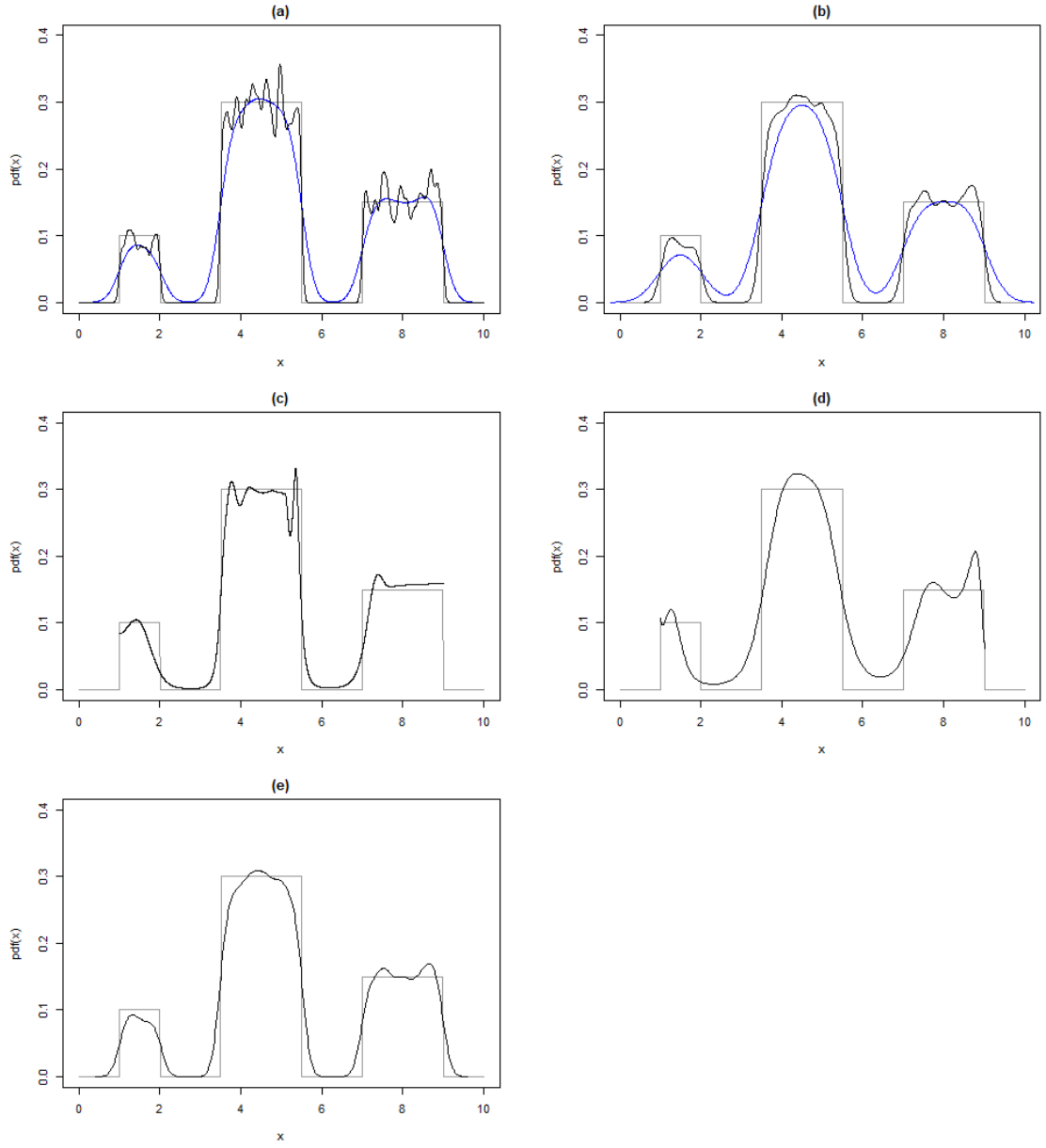

Figure B.1: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a mixture model of three uniform distributions with the parameters given in the code in appendix C.2 under the name "Uniform-Mix". (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

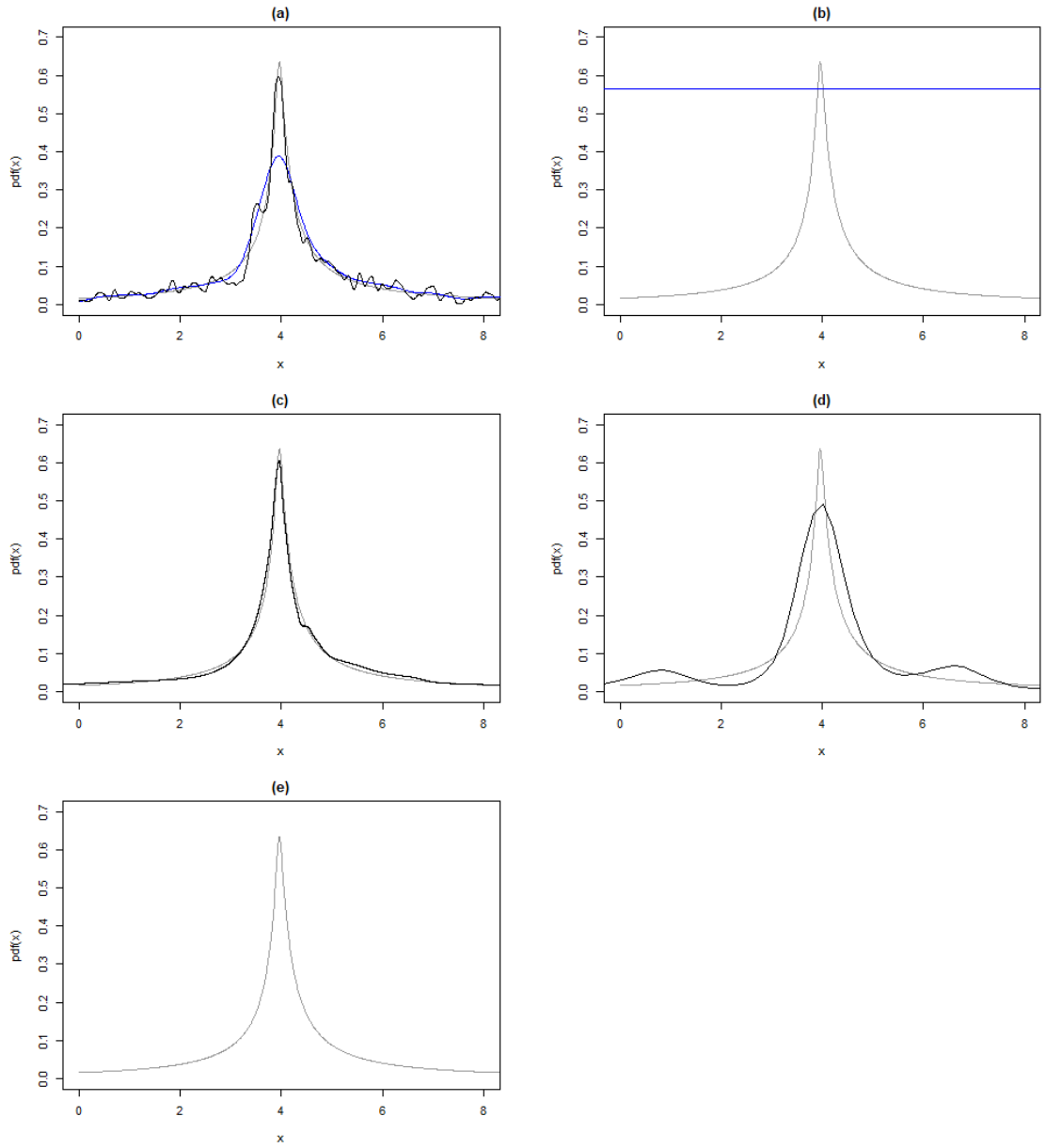

Figure B.2: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a stable distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

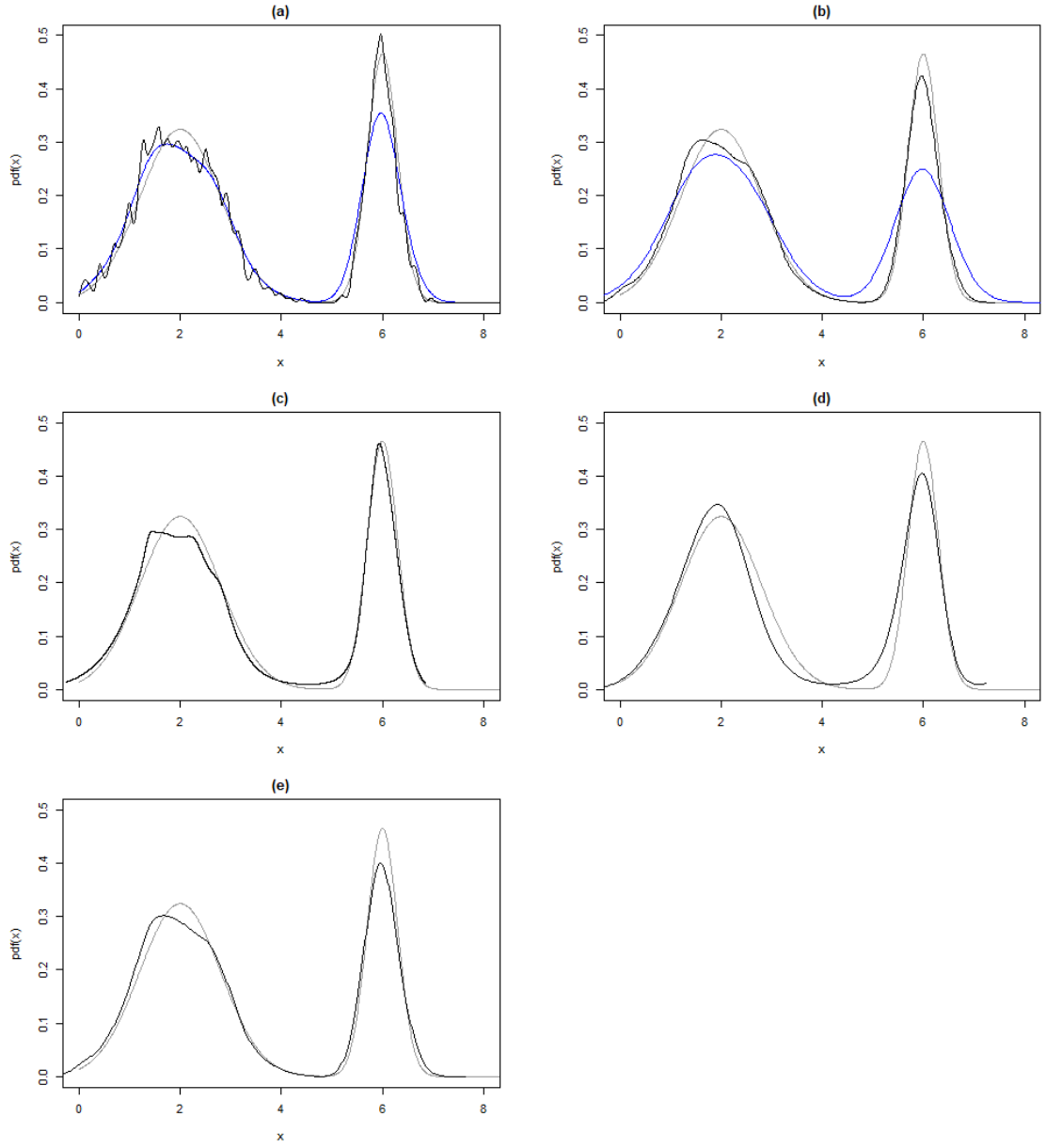

Figure B.3: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a bimodal normal distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

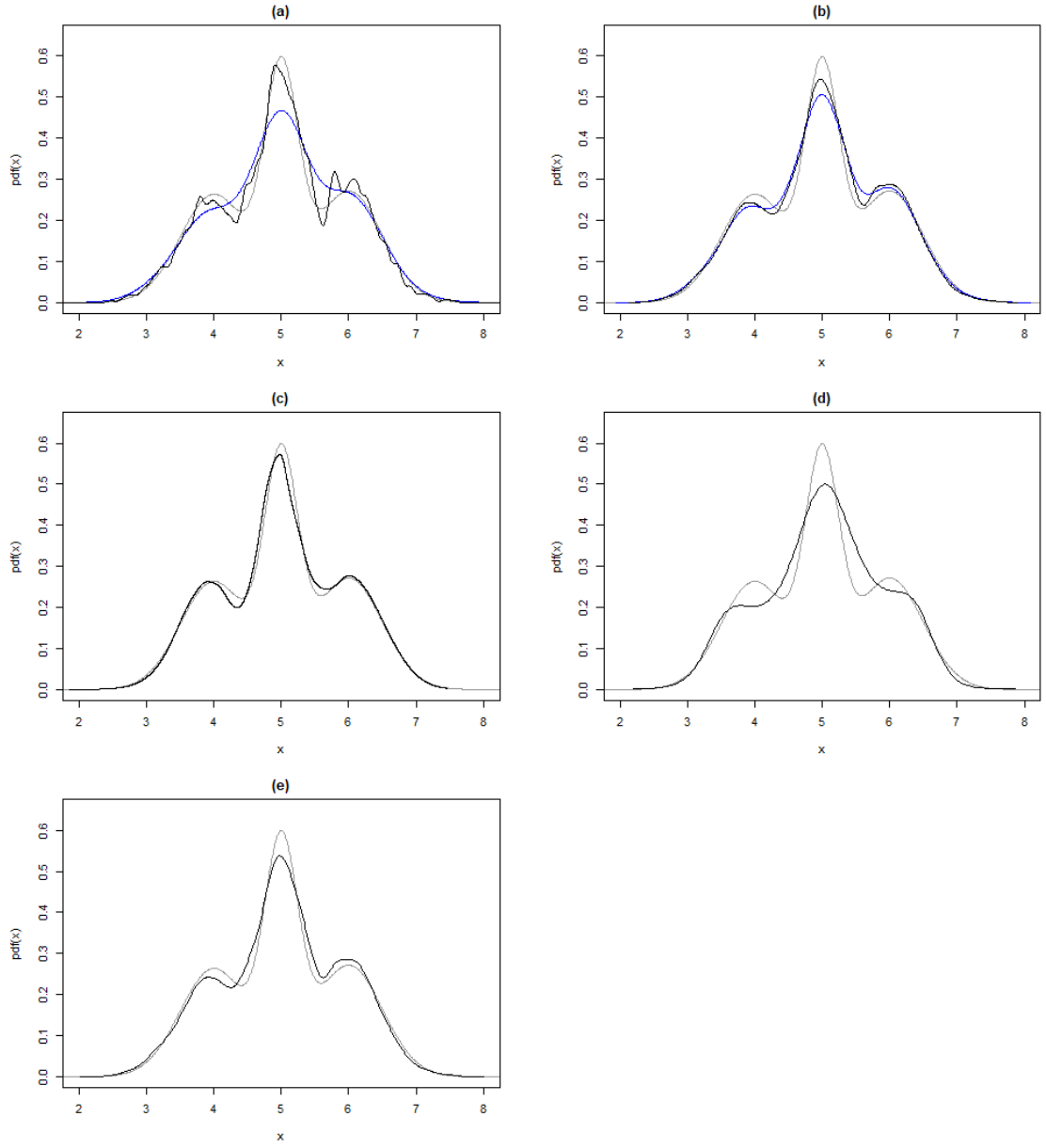

Figure B.4: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a trimodal normal distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

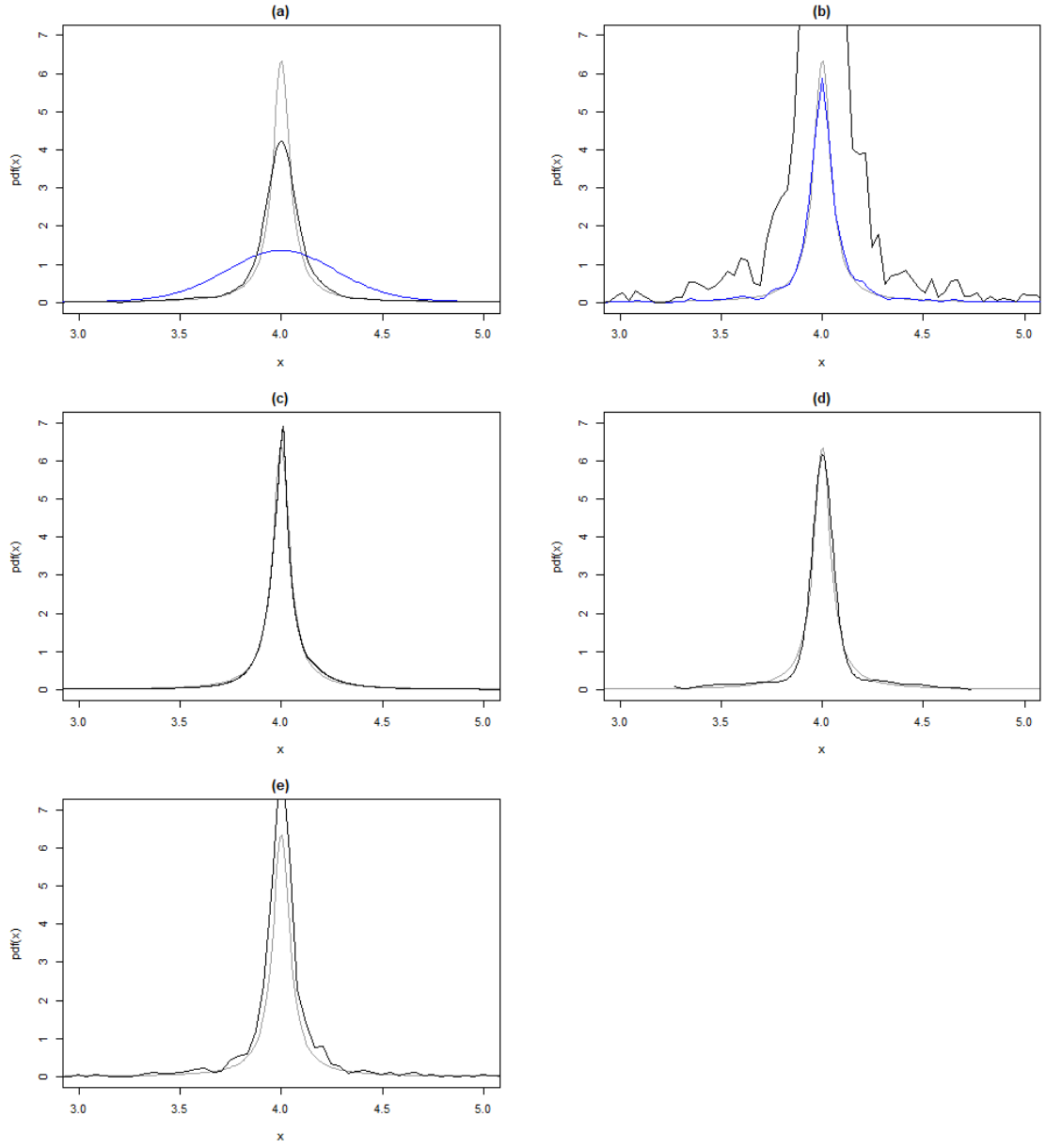

Figure B.5: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a t location-Scale distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

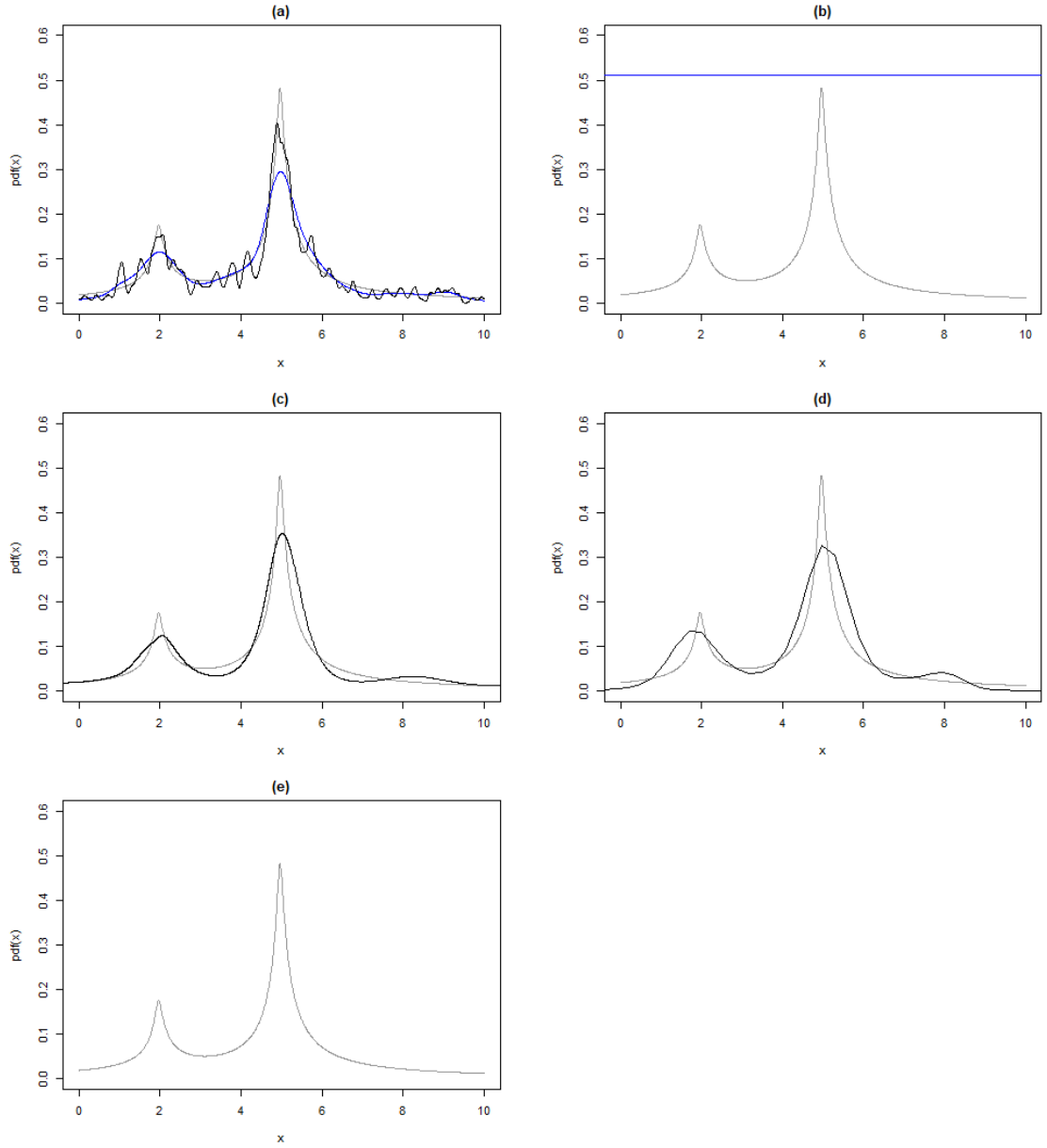

Figure B.6: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a stable distribution with the parameters given in the code in appendix C.2 under the name "Stable2". (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

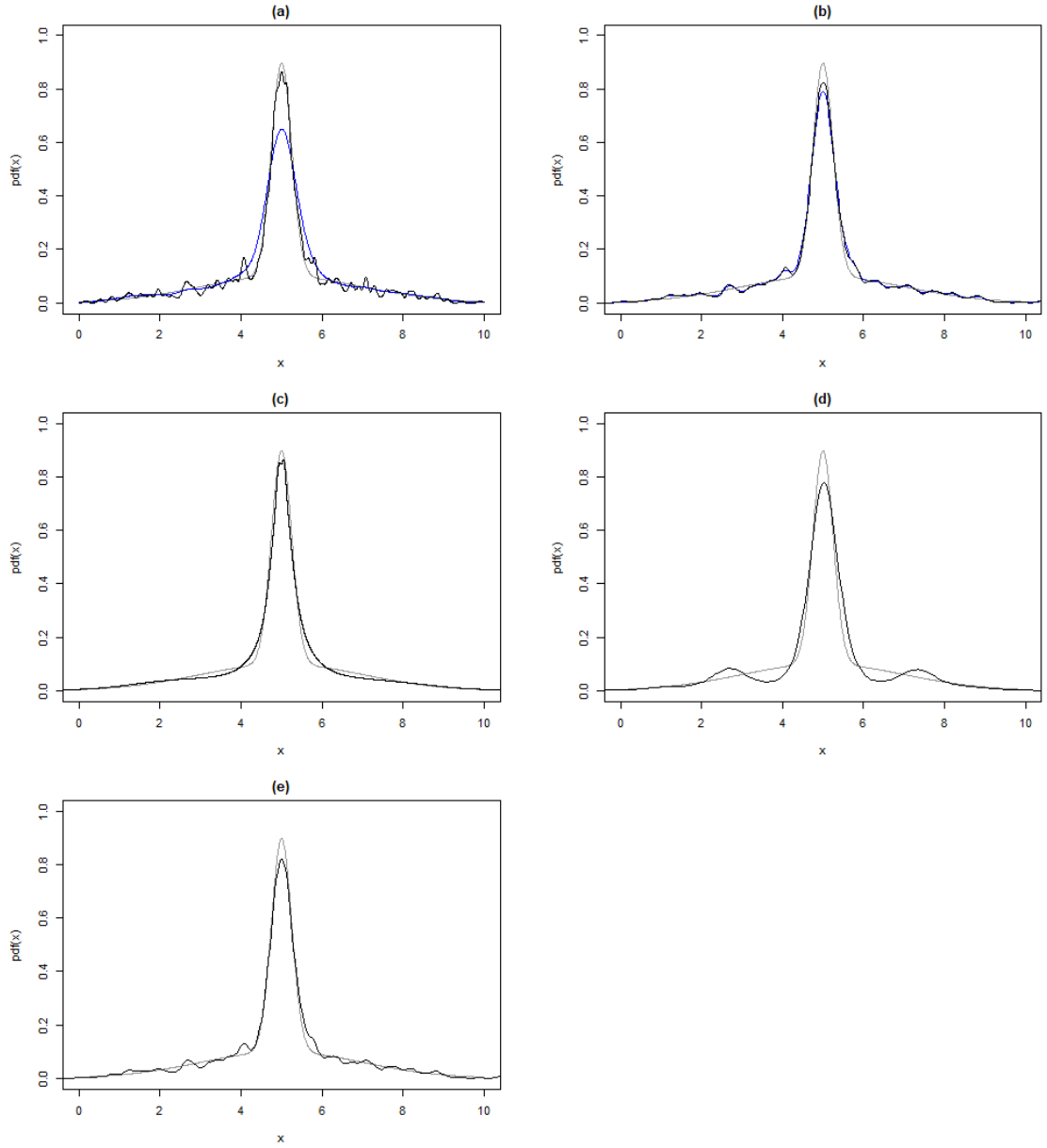

Figure B.7: $\hat{f}(x)$ for a sample of size $N = 1,024$ from a contaminated normal distribution with the parameters given in the code in appendix C.2. (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

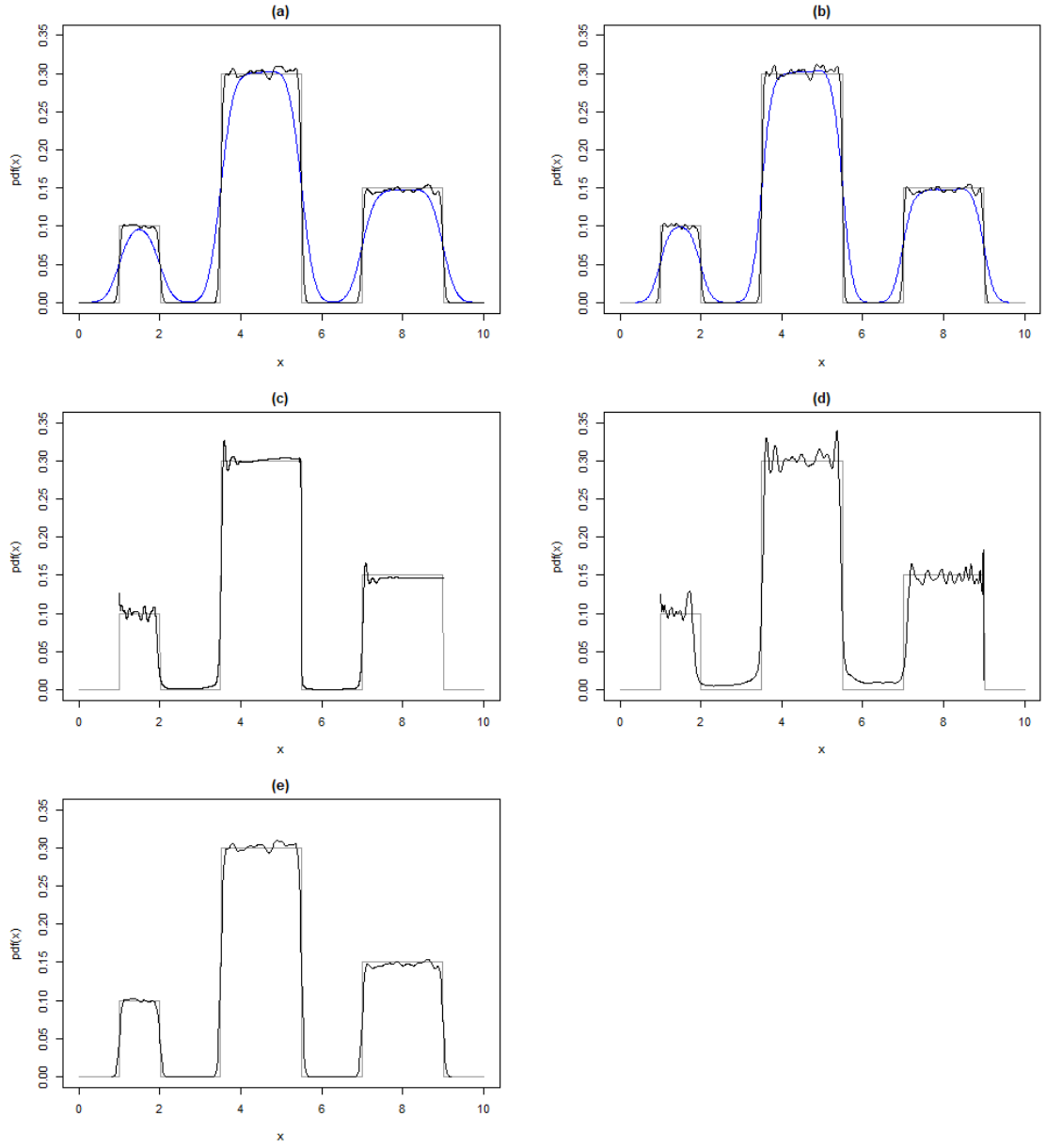

Figure B.8: $\hat{f}(x)$ for a sample of size $N = 65,536$ from a mixture model of a Birnbaum-Saunders and stable distribution with the parameters given in the code in appendix C.2 under the name "Uniform-Mix". (a) bkde estimator. (b) Density estimator (c) SE created in MATLAB (d) NMEM estimator from R (e) kde estimator.

APPENDIX C: MATLAB code

C.1 Optimized branching tree

C.1.1 Rtree.m code

```
close all; clear all; clc;
% Initialize variables
distributionVector = ["BirnbaumSaunders","Bimodal-Normal...
    ","Stable"];
distributionVector = ["Normal-Contaminated","Square-...
    periodic"];

maxSamplesExp =         18; %<-maximum exponent to ...
    generate samples
minSamplesExp =         18; %<-minimum exponent to ...
    generate samples
dataTypeflag =          true; %<-true/false integer powers...
     of
%  2/real powers of 2
ntrials =               1; %<-trials to run to generate ...
    heuristics
%  for programs
step =                  8; %<-control synthetic random ...
    samples to
%  skip being created
lowLim0 =               0; %<-lower limit to plot
upLim0 =                10; %<-upper limit to plot
saveImage =             false; %<-true/false save figures ...
    yes/no

treeType =              false; %<- true/false ballanced/...
    unbalanced

% create functions
br = @brProduct;
r = @getRation;

for j = 1:length(distributionVector)

    % Create vector of  samples labels
    sampleVec = ...
        samplesVector(minSamplesExp,maxSamplesExp,...
           dataTypeflag,step);

    for i = 1:ntrials
```

```
31          for k = 1:length(sampleVec)
32
33              Ns = sampleVec(k);
34
35              % parameters vector--
36              % p = [b, r, n, n-for-T, T-scale, window, ...
                    maximum levels];
37              % BR product function parameters---
38              % br = @(b,r,p,n) (b^p(1)*r^p(2))/n^p(3);
39              %-
40              % p = [1,0.5,1,0.5,0.005,ceil(0.1*Ns),6];
41              % p = [1,0.5,1,0.5,0.25,ceil(0.1*Ns),6];
42              % p = [1,0.5,1,0.5,4,ceil(0.125*Ns^0.5),20];
43              % % % % % % % % what  has been used for all ...
                    figures
44              % % % % % % % p = [1,0.5,1,0.25,8,ceil(0.125*...
                    Ns^0.5),20];
45              p = [1,0.5,1,0.25,8,ceil(0.125*Ns^0.5),20];
46              % % % % p = [1,0.5,1,0.25,8,ceil(0.0125*Ns)...
                    ,7];
47              % p = [1,1,1,0.25,8,ceil(0.125*Ns^0.5),20];
48              % good p vectors
49              % p = [1,1,0,1.5,2.5,100,100];
50              % p = [1,1,0,1.5,2.5,150,100];
51              % p = [1,1,0,1.5,2.5,200,100];
52              % p = [1,0.5,1,0.5,0.05,ceil(0.4*Ns),100];
53              % p = [1,0.5,1,0.5,0.5,ceil(0.1*Ns),6];
54              % create T threshold-
55              T = p(5)*Ns^p(4);
56              % T = p(5)*Log(Ns+1)^p(4);
57              % T = p(5)*log(Ns+1);
58              % window parameter for top/bottom points to ...
                    average------------
59              window = p(6);
60              % maximum number of potential levels/splits...
                    ------
61              maxLevel = p(7);
62              % minimum blocksize--
63              binmin = ceil(2*window);
64
65              % file name based on data generation code
66              filename = sprintf(['D_', char(...
                    distributionVector(j)),...
67                   '_T_','%d', '_S_','%d'],i, Ns);
68
69              % windows
```

```
70              filepath = ['D_',char(distributionVector(j))...
                   ,'\',...
71                   char(filename),'.txt']
72              % linux
73 %              filepath = ['D_',char(distributionVector(j))...
       ,'/',...
74 %                    char(filename),'.txt'];
75
76              % sample to be partitioned
77              sample = importdata(filepath);
78
79              % special cases for plot window limits
80              if distributionVector(j) == "Beta-a0p5-b1p5" ...
                   ||...
81                    distributionVector(j) == "Beta-a2-b0p...
                       5" ||...
82                    distributionVector(j) == "Beta-a0p5-b0...
                       p5"
83                   lowLim = 0;
84                   upLim = 1;
85              else
86                   lowLim = lowLim0;
87                   upLim = upLim0;
88              end
89
90
91              if binmin > Ns
92                   error('sample size too small for window ...
                       size')
93              end
94
95              % display useful variable values
96              disp(['Ns: ', num2str(Ns)])
97              disp(['T: ', num2str(T)])
98              disp(['window: ', num2str(window)])
99              disp(['binmin: ', num2str(binmin)])
100             disp(' ')
101
102             tic
103             % track number of branches per level
104             nbranch = 1;
105             % set end points of sample length as partition...
                   left (pL) and
106             % partion right (pR)
107             pL = 1;
108             pR = Ns;
```

```
109             % initialize vector to track all created ...
                    partitions
110             pList = [pL pR];
111             % track every attempted partition for all ...
                    levels
112             plevel = {{[1;Ns]}};
113             % initialize array to track newly created ...
                    partitions
114             % for plotting purposes
115             pdiff = {[1;Ns]};
116
117             % initial exit flag
118             exit = false;
119
120             % calcualte inital BR of intire sample
121             B0 = Ns;
122             R0 = r(sample,window);
123             BR0 = br(B0,R0,p,Ns);
124             % clear array to hold all BR values per block ...
                    per level
125             BRlevel = {BR0};
126             % beggin level loop
127             if BR0 > T
128
129                 for jj = 1:maxLevel
130                     disp(' ')
131                     disp(['\\\\ START LEVEL: ', num2str(jj...
                            )])
132
133                     % vector to hold all attempted ...
                            partitions per level
134                     plevHold = [];
135                     % vector to hold all BR values per ...
                            level
136                     BRHold = [];
137
138                     % beggin branch loop
139                     for b = 1:nbranch
140                         disp([' START branch: ', num2str(b...
                                )])
141
142                         % define block size (B)
143                         B = pList(b+1) - pList(b);
144
145                         if binmin > B
146                             disp(['block size (', num2str(...
                                    B),...
```

```
147                                 ') smaller than: ', num2...
                                       str(binmin)])
148                                 disp(['End this branch'])
149                                 continue
150                             end
151
152                             % update left
153                             bLeft = binmin + 1;
154                             bRight = B - binmin;
155
156                             % block to small for for ...
                                   minimization
157                             % given window size
158                             if bRight - bLeft < 3
159                                 break;
160                             end
161
162                             % minimization techniques...
                                   ------------
163
164                             % random search minimization
165                             % minimization functions
166 %                             [dxBR,brL,brR,partition,...
    rtreeFlag]...
167 %                                 = minimizeBRrand(...
168 %                                 sample(pList(b):pList(b+1))...
    ,...
169 %                                 window,p,binmin);
170
171                             % golden ration bifraction ...
                                   minimization
172                             [dxBR,brL,brR,partition] =...
173                                 minimizeBRgold(sample(pList(b)...
                                       :pList(b+1)),...
174                                 window,p,binmin);
175
176                             % distribution of dxBR: VERY SLOW ...
                                   FOR N > 2^14
177 %                             minimizeBRdiff(sample(pList(b):...
    pList(b+1)),...
178 %                                 window,p,binmin,saveImage,...
    filename,j);
179                             %---
180
181                             % update block boundaries of ...
                                   sample with correctly
182                             % placed partition
```

```
183                             newPar = pList(b) + partition;
184
185                             % calcualte R and BR for possilbe ...
                                   branches (b)
186 %                             R = r(sample(pList(b):pList(b+1)...
    ),window);
187 %                             BR = br(B,R,p,Ns);
188
189                             Rr = r(sample(newPar:pList(b+1)),...
                                   window);
190                             BRr = br(B,Rr,p,Ns);
191
192                             Rl = r(sample(pList(b):newPar),...
                                   window);
193                             BRl = br(B,Rl,p,Ns);
194
195                             BR = min(BRr,BRl);
196
197                             % hold all BRs per level for later...
                                    evaluation
198                             BRHold = [BRHold,BR];
199
200                             disp(['BR: ',num2str(BR)])
201                             disp(['T:  ', num2str(T)])
202
203                             if treeType
204                                 % balanced tree----
205                                 % STOP SEARCH: if any BR < T
206                                 if BR < T
207                                     break;
208                                     exit = true;
209                                 end
210                                 plevHold = [plevHold, newPar];
211                                 disp(['partition1: ',num2str(...
                                       newPar)])
212                             else
213                                 % un-balanced tree-
214                                 if BR ≥ T
215                                     plevHold = [plevHold, ...
                                           newPar];
216                                     disp(['partition1: ',num2...
                                           str(newPar)])
217                                 end
218                             end
219                         end
220
221                         disp(['BR: ',num2str(BR)])
```

```
222                     disp(['T:  ', num2str(T)])
223
224                     if treeType
225                         % balanced tree--------
226                         if BR ≥ T
227                             pList = [pList plevHold];
228                             disp(['partition2: ',num2str(...
                                   newPar)])
229                         end
230                     else
231                         % un-balanced tree-----
232                         temp = sort(BRHold);
233                         if temp(end) ≥ T
234                             pList = [pList plevHold];
235                             disp(['partition2: ',num2str(...
                                   newPar)])
236                         end
237                     end
238                     pList = sort(pList);
239
240                     % update nbranch
241                     nbranch = length(pList) - 1;
242
243                     % assign partition list to array for ...
                           plotting
244                     plevel{jj+1,1} = {pList'};
245
246                     % exit for special cases where B < ...
                           binmin
247                     % or bRight - bLeft < 3
248                     if isempty(BRHold)
249                         break;
250                     end
251
252                     % assign BR per level to array for ...
                           plotting
253                     BRlevel{jj+1,1} = BRHold;
254
255                     if exit
256                         break;
257                     end
258
259                     % find newly accepted partitions
260                     [C,¬] = setdiff(plevel{jj...
                           +1,1}{1,1}(:,1),...
261                         plevel{jj,1}{1,1}(:,1));
262
```

```
263                     % STOP SEARCH: if no new partitions ...
                           are accepted
264                     if isempty(C)
265                         break;
266                     end
267
268                     % update changes with newly created ...
                           partitions
269                     pdiff{jj+1,1} = C;
270                 end
271
272                 endTime = toc;
273                 pList = pList';
274                 disp(' ')
275                 disp('********* pList FINAL ANSWER')
276                 disp(pList)
277                 disp(['Elapse time: ',num2str(endTime),'s...
                       '])
278                 disp('*****************************')
279                 pList = pList';
280                 sample = sort(sample);
281
282                 %SPLITTING ROUTINE FOR LARGE SAMPLES...
                       ---------
283
284                 % vector to to add new partitons too
285                 LargNcheck = pList;
286                 % vector to hold updated partition list
287                 holder = pList;
288                 % while loop flag
289                 runSplit = true;
290                 while runSplit
291                     % triggers exit flag for while loop
292                     splitCount = 0;
293                     % loop over modified partition list (...
                           holder)
294                     for k = 1:length(holder)-1
295                         % calcualte difference
296                         diff = holder(k+1)-holder(k);
297                         % add partion between elements ...
                               when diff>20000
298                         if holder(k+1)-holder(k) > 20000
299                             split = floor((holder(k+1)-...
                                   holder(k))/2);
300                             % update new partiton list
301                             LargNcheck = [LargNcheck,...
                                   LargNcheck(k)+ split];
```

```
302                             % update counter: number of ...
                                   found splits
303                             splitCount = splitCount + 1;
304                         end
305                     end
306                     LargNcheck = sort(LargNcheck);
307                     holder = LargNcheck;
308                     % if no splits exit routine
309                     if splitCount == 0
310                         runSplit = false;
311                     end
312                 end
313                 pList = LargNcheck;
314                 disp('****************** pList SPLIT')
315                 disp(pList')
316                 disp('******************************')
317                 %-----------
318             end
319             % FIGURES ------
320             figure('Name','br values per level')
321             hold on
322             plot(0:size(plevel,1),log(T*ones(size(plevel...
                   ,1)+1)), '-r');
323             for k = 1:size(BRlevel,1)
324                 plot((k-1)*ones(size(BRlevel{k,1}(1,:),1)...
                       ,1),...
325                     log(BRlevel{k,1}(1,:)),...
326                     'o',...
327                     'MarkerEdgeColor',[0,0,0],...
328                     'MarkerFaceColor',[0,0,0],...
329                     'MarkerSize',4,'DisplayName','none')
330                 levelTrack = 1:size(plevel,1);
331             end
332
333             str = cell(1,size(levelTrack,2));
334             for ii = 1:length(levelTrack)
335                 str{ii} = sprintf('%1.0f',levelTrack(ii));
336             end
337             xticks(levelTrack)
338             xticklabels(str)
339             ylabel('ln(BR)')
340             xlabel('Tree Level')
341             legend('Threshold')
342             if saveImage
343                 binFileName = ['BR_',char(filename)];
344                 pngfile = strcat(binFileName,'.png');
345                 saveas(gcf,pngfile)
```

```
346                 figfile = strcat(binFileName,'.fig');
347                 saveas(gcf,figfile)
348             end
349
350             figure('Name','tree branching')
351             subplot(2,1,1)
352             histogram(sample)
353             ylabel('Number of Data Points','Interpreter','...
                   latex')
354
355             subplot(2,1,2)
356             hold on
357             % branching level track markers
358             for k = 1:size(plevel,1)-1
359                 plot(sample(plevel{k,1}{1,1}(:,1)),...
360                     (size(plevel,1)-k)*...
361                     ones(size(plevel{k,1}{1,1}(:,1),1),1)...
                           ,...
362                     'o',...
363                     'MarkerEdgeColor',[0.6,0.6,0.6],...
364                     'MarkerFaceColor',[0.6,0.6,0.6],...
365                     'MarkerSize',5)
366
367                 levelTrack = 0:size(plevel,1)-1;
368             end
369             % final partition markers
370 %             plot(sample(plevel{end,1}{1,1}(:,1)),...
371 %                 zeros(size(plevel{end,1}{1,1}(:,1),1),1)...
    ,...
372 %                 'o',...
373 %                 'MarkerEdgeColor',[1,0,0],...
374 %                 'MarkerFaceColor',[1,0,0],...
375 %                 'MarkerSize',5)
376
377             plot(sample(pList),...
378                 zeros(length(pList),1),...
379                 'o',...
380                 'MarkerEdgeColor',[1,0,0],...
381                 'MarkerFaceColor',[1,0,0],...
382                 'MarkerSize',5)
383
384             % boundries of sample markers
385             plot(sample(pdiff{1,1}(:,1)),...
386                 (size(plevel,1)-1)*...
387                 ones(size(pdiff{1,1}(:,1),1),1),...
388                 'o',...
389                 'MarkerEdgeColor',[0,0,0],...
```

```
390                 'MarkerFaceColor',[0.6,0.6,0.6],...
391                 'MarkerSize',8)
392             % new partion markers
393             for k = 2:size(pdiff,1)
394                 plot(sample(pdiff{k,1}(:,1)),...
395                     (size(plevel,1)-k)*...
396                     ones(size(pdiff{k,1}(:,1),1),1),...
397                     'o',...
398                     'MarkerEdgeColor',[0,0,0],...
399                     'MarkerFaceColor',[0,0,0],...
400                     'MarkerSize',8)
401             end
402
403             str = cell(1,size(levelTrack,2));
404             for ii = 1:length(levelTrack)
405                 str{ii} = sprintf('%1.0f',levelTrack(end...
                       +1-ii));
406             end
407
408             yticks(levelTrack)
409             yticklabels(str)
410             xlabel('x Range','Interpreter','latex')
411             ylabel('Tree Level','Interpreter','latex')
412             if saveImage
413                 binFileName = ['Tree_',char(filename)];
414                 pngfile = strcat(binFileName,'.png');
415                 saveas(gcf,pngfile)
416                 figfile = strcat(binFileName,'.fig');
417                 saveas(gcf,figfile)
418             end
419         end
420     end
421 end
```

C.1.2 getRatio.m code

```
1 function r = getRatio(sample,window)
2 sample = sort(sample);
3 n = length(sample);
4 dx = zeros(1,n-1);
5 dx(1:n-1) = sample(2:n) - sample(1:n-1);
6 dx = sort(dx');
7 dxmin = mean(dx(1:window));
8 dxmax = mean(dx(end-window:end));
9 r = dxmax/dxmin;
```

```
10 end
```

C.1.3 brProduct.m code

```
1 function BR = brProduct(b,r,p,n)
2 BR = (b^p(1)*r^p(2))/n^p(3);
3 end
```

C.1.4 minimizeBRgold.m code

```
1 function [dxbr1,brL,brR,partition] = minimizeBRgold(sample...
     ,window,p,binmin)
2 % function definition
3 br = @brProduct;
4 % sample size
5 sample = sort(sample);
6 Ns = length(sample);
7 % left and right search boundaries
8 bLeft = 1 + binmin;
9 bRight = Ns - binmin;
10 % golden ratio
11 goldenR = (1+sqrt(5))/2;
12 % distance from patition to evaluate dxBR
13 jiggle = 1;
14 dxbr = [];
15 % initial partion
16 partition = ceil(Ns/2);
17 while bRight-bLeft > 2
18     % update boundries
19     bLeft1 = partition ;
20     bRight1 = Ns - partition ;
21     % calculate R-ratio for given partition
22     rRight1 = getRation(sample(partition:end),window);
23     rLeft1 = getRation(sample(1:partition),window);
24     % calculate dxBR for partition location (center dxBR)
25     dxbrC = abs(br(bLeft1,rLeft1,p,Ns)-br(bRight1,rRight1,...
          p,Ns));
26     % -----------
27     % define new partition to the left of center
28     leftPar = partition - jiggle;
29     bLeft2 = leftPar;
30     bRight2 = Ns - leftPar ;
31     rRight2 = getRation(sample(leftPar:end),window);
```

```
32      rLeft2 = getRation(sample(1:leftPar),window);
33      % calculate dxBR for leftPar location (left dxBR)
34      dxbrL = abs(br(bLeft2,rLeft2,p,Ns)-br(bRight2,rRight2,...
            p,Ns));
35      % -----------
36      % define new partition to the right of center
37      rightPar = partition + jiggle;
38      bLeft3 = rightPar;
39      bRight3 = Ns - rightPar ;
40      rRight3 = getRation(sample(rightPar:end),window);
41      rLeft3 = getRation(sample(1:rightPar),window);
42      % calculate dxBR for leftPar location (rightPar dxBR)
43      dxbrR = abs(br(bLeft3,rLeft3,p,Ns)-br(bRight3,rRight3,...
            p,Ns));
44      % -----------
45      % calculate differences of dxBR for L,R,C
46      dxCR = dxbrC - dxbrR;
47      dxCL = dxbrC - dxbrL;
48      dxLR = dxbrL - dxbrR;
49      % dxbrR is smallest
50      if dxCR ≥ 0 && dxLR > 0
51          bLeft = leftPar;
52          %---
53          if bRight-bLeft < 2
54              % matrix to track useful variables
55              dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;...
                    bRight1;rRight1]);
56          else
57              % matrix to track useful variables
58              dxbr = horzcat(dxbr,[dxbrR;bLeft3;rLeft3;...
                    bRight3;rRight3]);
59              % use golden ratio to define new partion
60              partition = round((bLeft+bRight*goldenR)/(1+...
                    goldenR));
61  %             disp('Shrink --->')
62          end
63      end
64      % dxbrL is smallest
65      if dxCL ≥ 0 && dxLR < 0
66          bRight = rightPar;
67          %---
68          if bRight-bLeft < 2
69              % matrix to track useful variables
70              dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;...
                    bRight1;rRight1]);
71          else
72              % matrix to track useful variables
```

```
73              dxbr = horzcat(dxbr,[dxbrL;bLeft2;rLeft2;...
                   bRight2;rRight2]);
74              % use golden ratio to define new partion
75              partition = round((bLeft*goldenR+bRight)/(1+...
                   goldenR));
76 %             disp('<--- Shrink')
77          end
78      end
79      % dxbrC is smallest: solution found
80      if dxCL < 0 && dxCR < 0
81          dxbrFinal = dxbrC;
82          dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;bRight1;...
               rRight1]);
83          break
84      end
85 end
86 % disp(' ')
87 % disp('Program Results ...
     ======================================')
88 dxbr = dxbr';
89 % dxbrDisplay1 = [dxbr(:,1),dxbr(:,2)];
90 dxbr1 = dxbr(end,1);
91 bL1 = partition;
92 % dxbr(end,2),dxbr(end,3)
93 % dxbr(end,4),dxbr(end,5)
94 % brL = br(bLeft1,rLeft1,p,Ns);
95 % brR = br(bRight1,rRight1,p,Ns);
96 brL = br(dxbr(end,2),dxbr(end,3),p,Ns);
97 brR = br(dxbr(end,4),dxbr(end,5),p,Ns);
98 partition = dxbr(end,2);
99 end
```

C.1.5 minimizeBRrand.m code

```
1 function [dxbr1,brL,brR,partition,rtreeFlag] = ...
2     minimizeBRrand(sample,window,p,binmin)
3 % function definition
4 br = @brProduct;
5 % sample size
6 sample = sort(sample);
7 Ns = length(sample);
8 bLeft = 1 + binmin;
9 bRight = Ns - binmin;
10 jiggle = 1;
11 dxbr = [];
```

```
rtreeFlag = false;
while bRight-bLeft > 2
    partition = round((bRight-bLeft)*rand) + bLeft;
%     disp('--')
%     disp(['partition: ', num2str(partition)])
%     disp('--')
    % -----------
    bLeft1 = partition ;
    bRight1 = Ns - partition ;
    rRight1 = getRation(sample(partition:end),window);
    rLeft1 = getRation(sample(1:partition),window);
%     disp(['bLeft1: ',num2str(bLeft1)])
%     disp(['rLeft1: ',num2str(rLeft1)])
%     disp(['bRight1: ',num2str(bRight1)])
%     disp(['rRight1: ',num2str(rRight1)])
    dxbrC = abs(br(bLeft1,rLeft1,p,Ns)-br(bRight1,rRight1,...
       p,Ns));
%     disp(' ')
%     disp(['dxbrC: ',num2str(dxbrC)])
    % -----------
    Lpar = partition - jiggle;
    bLeft2 = Lpar;
    bRight2 = Ns - Lpar ;
    rRight2 = getRation(sample(Lpar:end),window);
    rLeft2 = getRation(sample(1:Lpar),window);
    dxbrL = abs(br(bLeft2,rLeft2,p,Ns)-br(bRight2,rRight2,...
       p,Ns));
%     disp(['dxbrL: ',num2str(dxbrL)])
    % -----------
    Rpar = partition + jiggle;
    bLeft3 = Rpar;
    bRight3 = Ns - Rpar ;
    rRight3 = getRation(sample(Rpar:end),window);
    rLeft3 = getRation(sample(1:Rpar),window);
    dxbrR = abs(br(bLeft3,rLeft3,p,Ns)-br(bRight3,rRight3,...
       p,Ns));
%     disp(['dxbrR: ',num2str(dxbrR)])
%     disp(' ')
    % -----------
    dxCR = dxbrC - dxbrR;
    dxCL = dxbrC - dxbrL;
    dxLR = dxbrL - dxbrR;
%     disp(['old bLeft: ', num2str(bLeft)])
%     disp(['old bRight: ', num2str(bRight)])
%     disp('--------')
    if dxCR ≥ 0 && dxLR > 0
        bLeft = Lpar;
```

```
56          %---
57          if bRight-bLeft < 2
58              dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;...
                    bRight1;rRight1]);
59          else
60              dxbr = horzcat(dxbr,[dxbrR;bLeft3;rLeft3;...
                    bRight3;rRight3]);
61  %             disp('Shrink --->')
62          end
63      end
64      if dxCL ≥ 0 && dxLR < 0
65          bRight = Rpar;
66          %---
67          if bRight-bLeft < 2
68              dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;...
                    bRight1;rRight1]);
69          else
70              dxbr = horzcat(dxbr,[dxbrL;bLeft2;rLeft2;...
                    bRight2;rRight2]);
71  %             disp('<--- Shrink')
72          end
73      end
74      if dxCL < 0 && dxCR < 0
75          bRight = Rpar;
76          dxbrFinal = dxbrC;
77          dxbr = horzcat(dxbr,[dxbrC;bLeft1;rLeft1;bRight1;...
                rRight1]);
78  %         disp(['Solution: ', num2str(dxbrC)])
79          break
80      end
81  %     disp(['new bLeft: ', num2str(bLeft)])
82  %     disp(['new bRight: ', num2str(bRight)])
83  end
84
85  % disp(' ')
86  % disp('Program Results ...
        ======================================')
87  dxbr = dxbr';
88  % dxbrDisplay1 = [dxbr(:,1),dxbr(:,2)];
89  dxbr1 = dxbr(end,1);
90  bL1 = partition;
91  % dxbr(end,2),dxbr(end,3)
92  % dxbr(end,4),dxbr(end,5)
93  % brL = br(bLeft1,rLeft1,p,Ns);
94  % brR = br(bRight1,rRight1,p,Ns);
95  brL = br(dxbr(end,2),dxbr(end,3),p,Ns);
96  brR = br(dxbr(end,4),dxbr(end,5),p,Ns);
```

```
97 partition = dxbr(end,2);
98 end
```

C.1.6 minimizeBRdiff.m code

```
1 function [dxbr,dxbrTrack,brL,brR,partition,rtreeFlag] = ...
    ...
2     minimizeBRdiff(sample,window,p,binmin,saveImage,...
         filename,j)
3 % function definition
4 br = @brProduct;
5 Ns = length(sample);
6 sample = sort(sample);
7 step = 1;
8 dxbrTrack = [];
9 rLTrack = [];
10 rRTrack = [];
11 bRTrack = [];
12 bLTrack = [];
13 brLTrack = [];
14 brRTrack = [];
15 BRleft = [];
16 BRright = [];
17 trig = -1;
18 rtreeFlag = false;
19 while trig < 0
20 %     bL = bL + binmin + step;
21     bL = binmin + step;
22     bR = Ns - step - binmin;
23     if bR < binmin + 2
24         break;
25         rtreeFlag = true;
26     end
27     rL = getRation(sample(1:bL),window);
28     rR = getRation(sample(bL:end),window);
29     rLTrack = [rLTrack, rL];
30     rRTrack = [rRTrack,rR];
31     dxbr = abs(br(bL,rL,p,Ns)-br(bR,rR,p,Ns));
32     BRleft = [BRleft,br(bL,rL,p,Ns)];
33     BRright = [BRright,br(bR,rR,p,Ns)];
34     brLTrack = [brLTrack, br(bL,rL,p,Ns)];
35     brRTrack = [brRTrack, br(bR,rR,p,Ns)];
36     bLTrack = [bLTrack, bL];
37     bRTrack = [bRTrack,bR];
38     dxbrTrack = [dxbrTrack,dxbr];
```

```
39      % will end loop once solution is found
40  %     if step > 3 && dxbr > dxbrTrack(step-1) &&...
41  %             dxbr > dxbrTrack(step-2) &&...
42  %             dxbr > dxbrTrack(step-3)
43  %         final = dxbrTrack(step-2);
44  %         break
45  %     end
46      step = step + 1;
47  %     disp(['step: ', num2str(step)])
48  %     disp('-------------')
49  %     pause
50  end
51  if j == 1
52      figure('Name','BRleft and BRright')
53      subplot(2,2,[1,2])
54      hold on
55      plot(1:length(BRright),BRright,'--','Color...
            ',[0.5,0.5,0.5])
56      plot(1:length(BRleft),BRleft,'--k')
57      plot(1:length(dxbrTrack),dxbrTrack,'-k')
58      ylabel('$\Delta \xi_0 = \xi_0 - \xi_1$','Interpreter...
            ','latex')
59      xlabel('$n_{partition}$','Interpreter','latex')
60      xlim([0,Ns])
61
62      subplot(2,2,3)
63      hold on
64      plot(1:length(rLTrack),rLTrack,'-k')
65      plot(1:length(rRTrack),rRTrack,'Color',[0.5,0.5,0.5])
66      ylabel('$R$','Interpreter','latex')
67      xlabel('$n_{partition}$','Interpreter','latex')
68      xlim([0,Ns])
69
70      subplot(2,2,4)
71      hold on
72      plot(1:length(bRTrack),bRTrack,'Color',[0.5,0.5,0.5])
73      plot(1:length(bLTrack),bLTrack,'-k')
74      ylabel('$B$','Interpreter','latex')
75      xlabel('$n_{partition}$','Interpreter','latex')
76      xlim([0,Ns])
77
78      if saveImage
79          binFileName = [num2str(Ns),'_BRacrossSample_',char...
                (filename)];
80          pngfile = strcat(binFileName,'.png');
81          saveas(gcf,pngfile)
82          figfile = strcat(binFileName,'.fig');
```

```
83          saveas(gcf,figfile)
84      end
85  end
86  [dxbr,bL]= min(dxbrTrack);
87  brL = brLTrack(bL);
88  brR = brRTrack(bL);
89  partition = bLTrack(bL);
90  dxbrTrack = dxbrTrack';
91  end
```

C.1.7 samplesVector.m

```
1  function [sampleVec] = ...
2      samplesVector(minSamplesExp,maxSamplesExp,dataTypeflag...
           ,step)
3
4  % to be function inputs
5  %--------------
6  %step = 1; %<---- can be changed to skip number of samples...
       created
7  %minSamplesExp;
8  %maxSamplesExp;
9  %dataTypeflag = true; %<--- true/false integer powers of ...
       2/real powers of 2
10
11 % Define a vector of samples to generate
12 %--------------
13 exponents = minSamplesExp:step:maxSamplesExp;
14 sampleVec = zeros(1,length(exponents));
15
16 if dataTypeflag
17     % Generates vector of samples from integer power 2
18     sampleVec(1:length(exponents)) = 2.^exponents(1:length...
           (exponents));
19 else
20     % Generates vector of samples from real power 2
21     for i = 1:length(exponents)
22         n = minSamplesExp + i + rand;
23         sampleVec(i) = floor(2^n);
24     end
25 end
26 end
```

C.2 Sample Generation

C.2.1 randomSampleCreationDriver.m code

```
%-------------
% This script is design to generate univariant data samples for a list of
% standard probability distributions. Multiple trials can be created for
% any range of sample sizes for each distribution. Samples can be created
% over a range of sizes for integer powers of two either for every integer
% power in the range or by skipping integer powers based on a step
% variable.The samples can be created using interger powers of two or using
% real number powers of two. An actual probabilty distribution data file
% can be created for later reference as well as plotted to aid in finding
% appropriate distribution parameters. PNG files may be saved of the actual
% distrbution plots.
%-------------
clc;clear all; close all;tic
% class assignment
actual = distributions;
% User Options
% Script switching board \\\\\\\\
generateRandomData =        false;  %<- true/false generate random data
                                    %   on/off
generateActData =           true;   %<- true/false generate actual
                                    %   distribution on/off
ActDistPlot =               true;   %<- true/false plot actual
                                    %   distribution on/off
dataTypeflag =              true;   %<- true/false integer powers of
                                    %   2/real powers of 2
savePNG =                   true;  %<- true/false save png of plots
```

```
27                                      %   on/off
28 % random data generation parameters \\\\\\
29 maxSamplesExp =             10; %<---- maximum exponent to...
       generate samples
30 minSamplesExp =             10; %<---- minimum exponent to...
       generate samples
31 actual.precision =          15; %<---- condtrol number of ...
      digits for
32                                 %      created data
33 ntrials =                    1;  %<---- trials to run to ...
      generate heuristics
34                                 %      for programs
35 step =                       1;  %<---- control synthetic ...
      random samples to
36                                 %      skip being created
37 actual.lowerLimit =          0;  %<---- lower limit to plot
38 actual.upperLimit =          10; %<---- upper limit to plot
39 % PROBABILITY DISTRIBUTION LIST \
40 % Total set
41 %
42 distributionVector = ["Beta-a0p5-b1p5","Beta-a2-b0p5",...
43     "Beta-a0p5-b0p5","Bimodal-Normal","BirnbaumSaunders...
          ",...
44     "BirnbaumSaunders-Stable","Burr","Exponential",...
45     "Extreme-Value","Gamma","Generalized-Extreme-Value...
          ",...
46     "Generalized-Pareto","HalfNormal","Normal",...
47     "Square-periodic","Stable","Stable2","Stable3",...
48     "tLocationScale","Uniform","Uniform-Mix","Weibull",...
49     "Chisquare","InverseGaussian","Trimodal-Normal"];
50 %}
51 %\\\\\\\\\\
52 % Main Loop for probability distribution data generation ...
      and vizualization
53 for j = 1:length(distributionVector)
54     % Define plot vector for distributions from 0-1
55     if distributionVector(j) == "Beta-a0p5-b1p5" ||...
56             distributionVector(j) == "Beta-a2-b0p5" ||...
57             distributionVector(j) == "Beta-a0p5-b0p5" ...
                  ||...
58             distributionVector(j) == "Mix-Beta-Stable-1"
59         actual.lowerLimit = 0;
60         actual.upperLimit = 1;
61         % vector used to create/plot actual distribution
62         actual.x = linspace(actual.lowerLimit,actual.....
              upperLimit,1000);
63     else
```

```
64          % Define plot vector for distribution from ...
               lowerLimit-upperLimit
65          actual.lowerLimit = 0;
66          actual.upperLimit = 10;
67          % vector used to create/plot actual distribution
68          actual.x = linspace(actual.lowerLimit,actual....
               upperLimit,1000);
69      end
70      % Current distribution name
71      actual.distributionName = distributionVector(j);
72      % file name for actual distribution. "A_" puts at the ...
           top of the folder
73      % for convenience
74      actual.fileName = ...
75          sprintf(['A_', char(actual.distributionName),'_Act...
               ']);
76      % Create actual distribution data and folders
77      if generateActData
78          actual.randomVSactual = "actual";
79          actual = actual.distributionsChoices();
80      end
81      % creat random object
82      random = actual;
83      % generate multiple trials for a given distribution ...
           and sample size
84      for i = 1:ntrials
85          % Create vector of  samples
86          sampleVec = samplesVector(minSamplesExp,...
87              maxSamplesExp,dataTypeflag,step);
88          for k = 1:length(sampleVec)
89              % size of sample to generate
90              random.Ns = sampleVec(k);
91              % Create fun.fileName for each distribtuion
92              random.fileName = sprintf(['D_',...
93                  char(actual.distributionName),...
94                  '_T_','%d', '_S_','%d'],i, random.Ns);
95              % Generate random data for each distribution ...
                   of varying sizes
96              if generateRandomData
97                  random.randomVSactual = "random";
98                  random.distributionsChoices();
99              end
100         end
101     end
102     % Act dist plots
103     if generateActData
104         if ActDistPlot
```

```
105             figure('Name','Standard Distributions')
106             % Plot Actual PDF for each distribution
107             plot(actual.x,actual.pdfCurve,'-k')
108             ylabel('$f(x)$','Interpreter','latex')
109             xlabel('x','Interpreter','latex')
110             title(char(actual.distributionName),'...
                   Interpreter','latex')
111             if max(actual.pdfCurve) > 1
112                 ylim([0,5])
113             else
114                 ylim([0,1])
115             end
116             xlim([actual.lowerLimit,actual.upperLimit])
117             if savePNG
118                 figureName = ['Act_D_',...
119                     char(actual.distributionName),...
120                     'S_',int2str(actual.Ns)];
121                 pngfile = strcat(char(figureName),'.png');
122                 saveas(gcf,pngfile)
123                 figfile = strcat(char(figureName),'.fig');
124                 saveas(gcf,figfile)
125             end
126         end
127     end
128 end
129 toc
```

C.2.2 distributions.m

```
1  classdef distributions
2      properties
3          x
4          Ns
5          fileName
6          pdfCurve
7          precision = 15;
8          lowerLimit = 0;
9          upperLimit = 10;
10         distributionName
11         distInfo
12         randomVSactual = "actual"
13     end
14     methods
15         function obj = distributionsChoices(obj)
16             debug = true;
```

```
17              switch obj.distributionName
18                  case 'Beta-a0p5-b1p5'
19                      % Beta1 Case Statement
20                      % First shape obj
21                      a = 0.5;
22                      % Second shape obj
23                      b = 1.5;
24                      % PDF Curve \
25                      obj.distInfo = makedist('Beta','a',a,'...
                           b',b);
26                      obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
27                      %\
28                      % generate random sample or actual pdf
29                      if obj.randomVSactual == "random"
30                          rndData = random(obj.distInfo,1,...
                               obj.Ns);
31                      elseif obj.randomVSactual == "actual"
32                          data = vertcat(obj.x,obj.pdfCurve)...
                               ;
33                      end
34                  case 'Beta-a2-b0p5'
35                      % Beta2 Case Statement
36                      % First shape obj
37                      a = 2;
38                      % Second shape obj
39                      b = 0.5;
40                      % PDF Curve \
41                      obj.distInfo = makedist('Beta','a',a,'...
                           b',b);
42                      obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
43                      %\
44                      % generate random sample or actual pdf
45                      if obj.randomVSactual == "random"
46                          rndData = random(obj.distInfo,1,...
                               obj.Ns);
47                      elseif obj.randomVSactual == "actual"
48                          data = vertcat(obj.x,obj.pdfCurve)...
                               ;
49                      end
50                  case 'Beta-a0p5-b0p5'
51                      % Beta3 Case Statement
52                      % First shape obj
53                      a = 0.5;
54                      % Second shape obj
55                      b = 0.5;
```

```
56                      % PDF Curve \
57                      obj.distInfo = makedist('Beta','a',a,'...
                           b',b);
58                      obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
59                      %\
60                      % generate random sample or actual pdf
61                      if obj.randomVSactual == "random"
62                          rndData = random(obj.distInfo,1,...
                               obj.Ns);
63                      elseif obj.randomVSactual == "actual"
64                          data = vertcat(obj.x,obj.pdfCurve)...
                               ;
65                      end
66                  case 'Bimodal-Normal'
67                      % Normal Case Statement
68                      % mixture weights
69                      p1 = 0.65;
70                      p2 = 1 - p1;
71                      p = [p1,p2];
72                      % Mean
73                      Mu1 = 2;
74                      Mu2 = 6;
75                      % Standard deviation
76                      Sigma1 = 0.8;
77                      Sigma2 = 0.3;
78                      % PDF Curve \
79                      % Distribution 1
80                      distributionLabel1 = 'Normal';
81                      distInfo1 = makedist(distributionLabel...
                           1,...
82                          'Mu', Mu1, 'Sigma', Sigma1);
83                      pdfCurve1 = pdf(distInfo1,obj.x);
84                      % Distribution 2
85                      distInfo2 = makedist(distributionLabel...
                           1,...
86                          'Mu', Mu2, 'Sigma', Sigma2);
87                      pdfCurve2 = pdf(distInfo2,obj.x);
88                      % Mixture PDF Curve
89                      obj.pdfCurve = p(1)*pdfCurve1 + p(2)*...
                           pdfCurve2;
90                      data = vertcat(obj.x,obj.pdfCurve);
91                      %\
92                      % generate random sample or actual pdf
93                      if obj.randomVSactual == "random"
94                          % mixture string array flag for ...
                               mixSampling()
```

```
95                          mixtureType = "two";
96                          % generate n vector for mixture ...
                               samplings
97                          n = mixSampling(obj.Ns,p,...
                               mixtureType);
98                          % generate random sample
99                          rndData1 = random(distInfo1,1,n(1)...
                               );
100                         rndData2 = random(distInfo2,1,n(2)...
                               );
101                         rndData = [rndData1,rndData2];
102                     elseif obj.randomVSactual == "actual"
103                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
104                     end
105                     % CREATE DISTRIBUTION OBJECT ...
                           ------------
106                     % mixture string array flag for ...
                           mixSampling()
107                     mixtureType = "two";
108                     % generate m vector for mixture ...
                           samplings
109                     m = mixSampling(10000,p,mixtureType);
110                     % generate random sample to create ...
                           distribution
111                     % object
112                     actData1 = random(distInfo1,1,m(1));
113                     actData2 = random(distInfo2,1,m(2));
114                     actData = [actData1,actData2];
115                     % generate numerical cdf: f=cdf, s=x-...
                           coordinates
116                     [f,s] = ecdf(actData);
117                     f = f(1:2:end);
118                     s = s(1:2:end);
119                     % generate distribution object
120                     obj.distInfo = ...
121                         makedist('PiecewiseLinear','x',s,'...
                               Fx',f);
122                     % create cdf/pdf from distribution ...
                           object.
123                     % for debugging and vizualization.
124                     if debug
125                         if obj.randomVSactual == "actual"
126                             xMix = linspace(obj.lowerLimit...
                                   ,...
127                                  obj.upperLimit,1000);
128                             CDF = cdf(obj.distInfo,xMix);
```

```
129                                 % numerically differentiate
130                                 PDF = zeros(1,size(CDF(1:end...
                                        -1),2));
131                                 for i = 2:size(CDF,2)-1
132                                     dx1 = (xMix(i+1)-xMix(i-1)...
                                            );
133                                     PDF(i) =  (CDF(i+1) - CDF(...
                                            i-1))/dx1;
134                                 end
135                                 % smooth pdf data
136                                 smoo1 = smooth(xMix(1:end-1),...
                                        PDF,0.03);
137                                 % plot cdf,pdf,smoothed-pdf
138                                 figure('Name',['Debug: ',...
139                                     char(obj.distributionName)...
                                            ])
140                                 subplot(2,1,1)
141                                 hold on
142                                 plot(xMix,CDF,'-m')
143                                 plot(xMix(1:end-1),PDF,'-r')
144                                 plot(xMix(1:end-1),smoo1,'-b')
145                                 ylabel('$f(x) or F(x)$','...
                                        Interpreter','latex')
146                                 xlabel('x','Interpreter','...
                                        latex')
147                                 legend('cdf','pdf','smoothed-...
                                        pdf')
148                                 % plot histogram for random ...
                                        sample
149                                 subplot(2,1,2)
150                                 histogram(random(obj.distInfo...
                                        ,1000,1),...
151                                     'Normalization','...
                                            probability')
152                             end
153                         end
154                         %-------
155                     case 'Binomial'
156                         % Binomial Case Statement
157                         % Number of trials
158                         n = 2000;
159                         % Porbability of success for each ...
                                trial
160                         p = 0.2;
161                         % PDF Curve \
162                         obj.distInfo = ...
```

```
163                         makedist(obj.distributionName,'n',...
                               n,'p',p);
164                     obj.pdfCurve = binopdf(obj.x,n,p);
165                     %\
166                     % generate random sample or actual pdf
167                     if obj.randomVSactual == "random"
168                         rndData = binornd(obj.Ns,p);
169                     elseif obj.randomVSactual == "actual"
170                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
171                     end
172                 case 'BirnbaumSaunders'
173                     % BirnbaumSaunders Case Statement
174                     % Scale parameter
175                     Beta = 1.5;
176                     % Shape parameter
177                     Gamma = 0.5;
178                     % PDF Curve \
179                     obj.distInfo = makedist(obj....
                           distributionName,...
180                         'Beta',Beta,'Gamma',Gamma);
181                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
182                     %\
183                     % generate random sample or actual pdf
184                     if obj.randomVSactual == "random"
185                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
186                     elseif obj.randomVSactual == "actual"
187                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
188                     end
189                 case 'BirnbaumSaunders-Stable'
190                     % BirnbaumSaunders Case Statement
191                     % mixture weights
192                     p1 = 0.35;
193                     p2 = 1 - p1;
194                     p = [p1,p2];
195                     % BirnbaumSaunders distribution ...
                           ---------
196                     % Scale parameter
197                     Beta = 1.5;
198                     % Shape parameter
199                     Gamma = 0.5;
200                     % Stable distribution -----
201                     % First shape parameter
202                     Alpha1 = 0.5;
```

```
203                     % Second shape parameter: -1 ≤ Beta ≤ ...
                           1
204                     Beta1 = 0.05;
205                     % Scale parameter
206                     Gam1 = 1;
207                     % Location parameter
208                     Delta1 = 7;
209                     % PDF Curve \
210                     % BirnbaumSaunders distribution
211                     distributionLabel1 = 'BirnbaumSaunders...
                           ';
212                     distInfo1 = makedist(distributionLabel...
                           1,...
213                      'Beta',Beta,'Gamma',Gamma);
214                     pdfCurve1 = pdf(distInfo1,obj.x);
215                     % Stable distribution
216                     distributionLabel2 = 'Stable';
217                     distInfo2 = makedist(distributionLabel...
                           2,...
218                      'Alpha', Alpha1,'Beta', Beta1,...
219                      'Gam', Gam1, 'Delta', Delta1);
220                     pdfCurve2 = pdf(distInfo2, obj.x);
221                     % Mixture PDF Curve
222                     obj.pdfCurve = p(1)*pdfCurve1 + p(2)*...
                           pdfCurve2;
223                     data = vertcat(obj.x,obj.pdfCurve);
224                     %\
225                     % generate random sample or actual pdf
226                     if obj.randomVSactual == "random"
227                         % mixture string array flag for ...
                               mixSampling()
228                         mixtureType = "two";
229                         % generate n vector for mixture ...
                               samplings
230                         n = mixSampling(obj.Ns,p,...
                               mixtureType);
231                         % generate random sample
232                         rndData1 = random(distInfo1,1,n(1)...
                               );
233                         rndData2 = random(distInfo2,1,n(2)...
                               );
234                         rndData = [rndData1,rndData2];
235                     elseif obj.randomVSactual == "actual"
236                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
237                     end
```

```
238                     % CREATE DISTRIBUTION OBJECT ...
                           ------------
239                     % mixture string array flag for ...
                           mixSampling()
240                     mixtureType = "two";
241                     % generate m vector for mixture ...
                           samplings
242                     m = mixSampling(10000,p,mixtureType);
243                     % generate random sample to create ...
                           distribution
244                     % object
245                     actData1 = random(distInfo1,1,m(1));
246                     actData2 = random(distInfo2,1,m(2));
247                     actData = [actData1,actData2];
248                     % generate numerical cdf: f=cdf, s=x-...
                           coordinates
249                     [f,s] = ecdf(actData);
250                     f = f(1:2:end);
251                     s = s(1:2:end);
252                     % generate distribution object
253                     obj.distInfo = ...
254                         makedist('PiecewiseLinear','x',s,'...
                               Fx',f);
255                     % create cdf/pdf from distribution ...
                           object.
256                     % for debugging and vizualization.
257                     if debug
258                         if obj.randomVSactual == "actual"
259                             xMix = linspace(obj.lowerLimit...
                                   ,...
260                                  obj.upperLimit,1000);
261                             CDF = cdf(obj.distInfo,xMix);
262                             % numerically differentiate
263                             PDF = zeros(1,size(CDF(1:end...
                                   -1),2));
264                             for i = 2:size(CDF,2)-1
265                                 dx1 = (xMix(i+1)-xMix(i-1)...
                                       );
266                                 PDF(i) =  (CDF(i+1) - CDF(...
                                       i-1))/dx1;
267                             end
268                             % smooth pdf data
269                             smoo1 = smooth(xMix(1:end-1),...
                                   PDF,0.03);
270                             % plot cdf,pdf,smoothed-pdf
271                             figure('Name',['Debug: ',...
```

```
272                                 char(obj.distributionName)...
                                       ])
273                             subplot(2,1,1)
274                             hold on
275                             plot(xMix,CDF,'-m')
276                             plot(xMix(1:end-1),PDF,'-r')
277                             plot(xMix(1:end-1),smoo1,'-b')
278                             ylabel('$f(x) or F(x)$','...
                                   Interpreter','latex')
279                             xlabel('x','Interpreter','...
                                   latex')
280                             legend('cdf','pdf','smoothed-...
                                   pdf')
281                             % plot histogram for random ...
                                   sample
282                             subplot(2,1,2)
283                             histogram(random(obj.distInfo...
                                   ,1000,1),...
284                                  'Normalization','...
                                       probability')
285                         end
286                     end
287                     %-------
288                 case 'Burr'
289                     % Burr Case Statement
290                     % Scale parameter
291                     Alpha = 1;
292                     % Shape parameter one
293                     c = 2;
294                     % Shape parameter one two
295                     k = 2;
296                     % PDF Curve \
297                     obj.distInfo = makedist(obj....
                           distributionName,...
298                          'Alpha',Alpha,'c',c,'k',k);
299                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
300                     %\
301                     % generate random sample or actual pdf
302                     if obj.randomVSactual == "random"
303                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
304                     elseif obj.randomVSactual == "actual"
305                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
306                     end
307                 case 'Chisquare'
```

```
308                     % Chisquare Case Statement
309                     % Degrees of freedom
310                     Nu = 4;
311                     % PDF Curve \
312                     obj.pdfCurve = chi2pdf(obj.x,Nu);
313                     %\
314                     % generate random sample or actual pdf
315                     if obj.randomVSactual == "random"
316                         rndData = chi2rnd(Nu,1,obj.Ns);
317                     elseif obj.randomVSactual == "actual"
318                         data = vertcat(obj.x,obj.pdfCurve)...
                                ;
319                     end
320                 case 'Exponential'
321                     % Exponential Case Statement
322                     % Mean
323                     Mu = 1;
324                     % PDF Curve \
325                     obj.distInfo = makedist(obj....
                            distributionName,'Mu',Mu);
326                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                            ;
327                     %\
328                     % generate random sample or actual pdf
329                     if obj.randomVSactual == "random"
330                         rndData = random(obj.distInfo,1,...
                                obj.Ns);
331                     elseif obj.randomVSactual == "actual"
332                         data = vertcat(obj.x,obj.pdfCurve)...
                                ;
333                     end
334                 case 'Extreme-Value'
335                     % Extreme Value Case Statement
336                     % Location parameter
337                     Mu = 1;
338                     % Scale parameter
339                     Sigma = 2;
340                     % PDF Curve \
341                     distributionLabel = 'Extreme Value';
342                     obj.distInfo = makedist(...
                            distributionLabel,...
343                         'Mu',Mu, 'Sigma', Sigma);
344                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                            ;
345                     %\
346                     % generate random sample or actual pdf
347                     if obj.randomVSactual == "random"
```

```
348                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
349                     elseif obj.randomVSactual == "actual"
350                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
351                     end
352                 case 'Gamma'
353                     % Gamma Case Statement
354                     % Shape parameter
355                     a = 2;
356                     % Scale parameter
357                     b = 2;
358                     % PDF Curve \
359                     obj.distInfo = ...
360                         makedist(obj.distributionName,'a',...
                               a,'b',b);
361                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
362                     %\
363                     % generate random sample or actual pdf
364                     if obj.randomVSactual == "random"
365                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
366                     elseif obj.randomVSactual == "actual"
367                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
368                     end
369                 case 'Generalized-Extreme-Value'
370                     % Generalized Extreme Value Value ...
                           Statement
371                     % Shape parameter
372                     k = 1;
373                     % Scale parameter
374                     Sigma = 2;
375                     % Location parameter
376                     Mu = 2;
377                     % PDF Curve \
378                     distributionLabel = 'Generalized ...
                           Extreme Value';
379                     obj.distInfo = makedist(...
                           distributionLabel,...
380                          'k',k, 'Sigma', Sigma,'Mu',Mu);
381                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
382                     %\
383                     % generate random sample or actual pdf
384                     if obj.randomVSactual == "random"
```

```
385                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
386                     elseif obj.randomVSactual == "actual"
387                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
388                     end
389                 case 'Generalized-Pareto'
390                     % Generalized Pareto Value Value Case ...
                           Statement
391                     % Tail MemTracker (shape) parameter
392                     k = 2;
393                     % Scale parameter
394                     Sigma = 1;
395                     % Threshold (location) parameter
396                     theta = 0;
397                     % PDF Curve \
398                     distributionLabel = 'Generalized ...
                           Pareto';
399                     obj.distInfo = makedist(...
                           distributionLabel,...
400                          'k',k, 'Sigma', Sigma,'Theta', ...
                               theta);
401                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
402                     % Mu parameter is not recognized
403                     %\
404                     % generate random sample or actual pdf
405                     if obj.randomVSactual == "random"
406                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
407                     elseif obj.randomVSactual == "actual"
408                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
409                     end
410                 case 'HalfNormal'
411                     % Half Normal Value Case Statement
412                     % Location parameter
413                     Mu = 0;
414                     % Scale parameter
415                     Sigma = 1;
416                     % PDF Curve \
417                     obj.distInfo = makedist(obj....
                           distributionName,...
418                          'Mu', Mu, 'Sigma', Sigma);
419                     obj.pdfCurve = pdf(obj....
                           distributionName,obj.x);
420                     %\
```

```
421                     % generate random sample or actual pdf
422                     if obj.randomVSactual == "random"
423                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
424                     elseif obj.randomVSactual == "actual"
425                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
426                     end
427                 case 'InverseGaussian'
428                     % Inverse Gaussian Case Statement
429                     % Scale parameter
430                     Mu = 1;
431                     % Shape parameter
432                     Lambda = 1;
433                     % PDF Curve \
434                     obj.distInfo = makedist(obj....
                           distributionName,...
435                         'mu', Mu, 'lambda', Lambda);
436                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
437                     %\
438                     % generate random sample or actual pdf
439                     if obj.randomVSactual == "random"
440                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
441                     elseif obj.randomVSactual == "actual"
442                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
443                     end
444                 case 'Normal'
445                     % Normal Case Statement
446                     % Mean
447                     Mu = 5;
448                     % Standard deviation
449                     Sigma = 1;
450                     % PDF Curve \
451                     obj.distInfo = makedist(obj....
                           distributionName,...
452                         'Mu', Mu, 'Sigma', Sigma);
453                     obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
454                     %\
455                     % generate random sample or actual pdf
456                     if obj.randomVSactual == "random"
457                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
458                     elseif obj.randomVSactual == "actual"
```

```
459                         data = vertcat(obj.x,obj.pdfCurve)...
                                ;
460                     end
461                 case 'Normal-Contaminated'
462                     % Normal Case Statement
463                     % mixture weights
464                     p1 = 0.5;
465                     p2 = 1 - p1;
466                     p = [p1,p2];
467                     % Mean
468                     Mu1 = 5;
469                     Mu2 = 5;
470                     % Standard deviation
471                     Sigma1 = 2;
472                     Sigma2 = 0.25;
473                     % PDF Curve \
474                     % Distribution 1
475                     distributionLabel1 = 'Normal';
476                     distInfo1 = makedist(distributionLabel...
                           1,...
477                         'Mu', Mu1, 'Sigma', Sigma1);
478                     pdfCurve1 = pdf(distInfo1,obj.x);
479                     % Distribution 2
480                     distInfo2 = makedist(distributionLabel...
                           1,...
481                         'Mu', Mu2, 'Sigma', Sigma2);
482                     pdfCurve2 = pdf(distInfo2,obj.x);
483                     % Mixture PDF Curve
484                     obj.pdfCurve = p(1)*pdfCurve1 + p(2)*...
                           pdfCurve2;
485                     data = vertcat(obj.x,obj.pdfCurve);
486                     %\
487                     % generate random sample or actual pdf
488                     if obj.randomVSactual == "random"
489                         % mixture string array flag for ...
                               mixSampling()
490                         mixtureType = "two";
491                         % generate n vector for mixture ...
                               samplings
492                         n = mixSampling(obj.Ns,p,...
                               mixtureType);
493                         % generate random sample
494                         rndData1 = random(distInfo1,1,n(1)...
                               );
495                         rndData2 = random(distInfo2,1,n(2)...
                               );
496                         rndData = [rndData1,rndData2];
```

```
497                         elseif obj.randomVSactual == "actual"
498                             data = vertcat(obj.x,obj.pdfCurve)...
                                    ;
499                         end
500                         % CREATE DISTRIBUTION OBJECT ...
                            ------------
501                         % mixture string array flag for ...
                            mixSampling()
502                         mixtureType = "two";
503                         % generate m vector for mixture ...
                            samplings
504                         m = mixSampling(10000,p,mixtureType);
505                         % generate random sample to create ...
                            distribution
506                         % object
507                         actData1 = random(distInfo1,1,m(1));
508                         actData2 = random(distInfo2,1,m(2));
509                         actData = [actData1,actData2];
510                         % generate numerical cdf: f=cdf, s=x-...
                            coordinates
511                         [f,s] = ecdf(actData);
512                         f = f(1:2:end);
513                         s = s(1:2:end);
514                         % generate distribution object
515                         obj.distInfo = ...
516                             makedist('PiecewiseLinear','x',s,'...
                                Fx',f);
517                         % create cdf/pdf from distribution ...
                            object.
518                         % for debugging and vizualization.
519                         if debug
520                             if obj.randomVSactual == "actual"
521                                 xMix = linspace(obj.lowerLimit...
                                    ,...
522                                     obj.upperLimit,1000);
523                                 CDF = cdf(obj.distInfo,xMix);
524                                 % numerically differentiate
525                                 PDF = zeros(1,size(CDF(1:end...
                                    -1),2));
526                                 for i = 2:size(CDF,2)-1
527                                     dx1 = (xMix(i+1)-xMix(i-1)...
                                        );
528                                     PDF(i) =  (CDF(i+1) - CDF(...
                                        i-1))/dx1;
529                                 end
530                                 % smooth pdf data
```

```
531                                 smoo1 = smooth(xMix(1:end-1),...
                                       PDF,0.03);
532                                 % plot cdf,pdf,smoothed-pdf
533                                 figure('Name',['Debug: ',...
534                                     char(obj.distributionName)...
                                        ])
535                                 subplot(2,1,1)
536                                 hold on
537                                 plot(xMix,CDF,'-m')
538                                 plot(xMix(1:end-1),PDF,'-r')
539                                 plot(xMix(1:end-1),smoo1,'-b')
540                                 ylabel('$f(x) or F(x)$','...
                                       Interpreter','latex')
541                                 xlabel('x','Interpreter','...
                                       latex')
542                                 legend('cdf','pdf','smoothed-...
                                       pdf')
543                                 % plot histogram for random ...
                                       sample
544                                 subplot(2,1,2)
545                                 histogram(random(obj.distInfo...
                                       ,1000,1),...
546                                     'Normalization','...
                                        probability')
547                             end
548                         end
549                         %-------
550                     case 'Square-periodic'
551                         % Uniform Case Statement
552                         % mixture weights
553                         p1 = 1/6;
554                         p2 = 1/6;
555                         p3 = 1/6;
556                         p4 = 1/6;
557                         p5 = 1/6;
558                         p6 = 1 - p1 - p2 - p3 - p4- p5 ;
559                         p = [p1,p2,p3,p4,p5,p6];
560                         % Lower bound
561                         Lower1 = 1;
562                         Lower2 = 2.5;
563                         Lower3 = 4;
564                         Lower4 = 5.5;
565                         Lower5 = 7;
566                         Lower6 = 8.5;
567                         % Upper Bound
568                         Upper1 = 2;
569                         Upper2 = 3.5;
```

```
570                     Upper3 = 5;
571                     Upper4 = 6.5;
572                     Upper5 = 8;
573                     Upper6 = 9.5;
574                     % PDF Curve \
575                     distributionLabel1 = 'Uniform';
576                     % Distribution 1
577                     distInfo1 = makedist(distributionLabel...
                           1,...
578                         'Lower', Lower1, 'Upper', Upper1);
579                     pdfCurve1 = pdf(distInfo1,obj.x);
580                     % Distribution 2
581                     distInfo2 = makedist(distributionLabel...
                           1,...
582                         'Lower', Lower2, 'Upper', Upper2);
583                     pdfCurve2 = pdf(distInfo2,obj.x);
584                     % Distribution 3
585                     distInfo3 = makedist(distributionLabel...
                           1,...
586                         'Lower', Lower3, 'Upper', Upper3);
587                     pdfCurve3 = pdf(distInfo3,obj.x);
588                     % Distribution 4
589                     distInfo4 = makedist(distributionLabel...
                           1,...
590                         'Lower', Lower4, 'Upper', Upper4);
591                     pdfCurve4 = pdf(distInfo4,obj.x);
592                     % Distribution 5
593                     distInfo5 = makedist(distributionLabel...
                           1,...
594                         'Lower', Lower5, 'Upper', Upper5);
595                     pdfCurve5 = pdf(distInfo5,obj.x);
596                     % Distribution 6
597                     distInfo6 = makedist(distributionLabel...
                           1,...
598                         'Lower', Lower6, 'Upper', Upper6);
599                     pdfCurve6 = pdf(distInfo6,obj.x);
600                     % Mixture PDF Curve
601                     obj.pdfCurve = p(1)*pdfCurve1 + p(2)*...
                           pdfCurve2 +...
602                         p(3)*pdfCurve3 + p(4)*pdfCurve4 ...
                               +...
603                         p(5)*pdfCurve5 + p(6)*pdfCurve6;
604                     %\
605                     % generate random sample or actual pdf
606                     if obj.randomVSactual == "random"
607                         % mixture string array flag for ...
                               mixSampling()
```

```
608                             mixtureType = "six";
609                             % generate n vector for mixture ...
                                   samplings
610                             n = mixSampling(obj.Ns,p,...
                                   mixtureType);
611                             % generate random sample
612                             rndData1 = random(distInfo1,1,n(1)...
                                   );
613                             rndData2 = random(distInfo2,1,n(2)...
                                   );
614                             rndData3 = random(distInfo3,1,n(3)...
                                   );
615                             rndData4 = random(distInfo4,1,n(4)...
                                   );
616                             rndData5 = random(distInfo5,1,n(5)...
                                   );
617                             rndData6 = random(distInfo6,1,n(6)...
                                   );
618                             rndData = [rndData1,rndData2,...
                                   rndData3,...
619                                  rndData4,rndData5,rndData6];
620                         elseif obj.randomVSactual == "actual"
621                             data = vertcat(obj.x,obj.pdfCurve)...
                                   ;
622                         end
623                         % CREATE DISTRIBUTION OBJECT ...
                               ------------
624                         % mixture string array flag for ...
                               mixSampling()
625                         mixtureType = "six";
626                         % generate m vector for mixture ...
                               samplings
627                         m = mixSampling(10000,p,mixtureType);
628                         % generate random sample to create ...
                               distribution
629                         % object
630                         actData1 = random(distInfo1,1,m(1));
631                         actData2 = random(distInfo2,1,m(2));
632                         actData3 = random(distInfo3,1,m(3));
633                         actData4 = random(distInfo4,1,m(4));
634                         actData5 = random(distInfo5,1,m(5));
635                         actData6 = random(distInfo6,1,m(6));
636                         actData = [actData1,actData2,actData...
                               3,...
637                              actData4,actData5,actData6];
638                         % generate numerical cdf: f=cdf, s=x-...
                               coordinates
```

```
639                         [f,s] = ecdf(actData);
640                         f = f(1:2:end);
641                         s = s(1:2:end);
642                         % generate distribution object
643                         obj.distInfo = ...
644                             makedist('PiecewiseLinear','x',s,'...
                                   Fx',f);
645                         % create cdf/pdf from distribution ...
                               object.
646                         % for debugging and vizualization.
647                         if debug
648                             if obj.randomVSactual == "actual"
649                                 xMix = linspace(obj.lowerLimit...
                                       ,...
650                                     obj.upperLimit,1000);
651                                 CDF = cdf(obj.distInfo,xMix);
652                                 % numerically differentiate
653                                 PDF = zeros(1,size(CDF(1:end...
                                       -1),2));
654                                 for i = 2:size(CDF,2)-1
655                                     dx1 = (xMix(i+1)-xMix(i-1)...
                                           );
656                                     PDF(i) =  (CDF(i+1) - CDF(...
                                           i-1))/dx1;
657                                 end
658                                 % smooth pdf data
659                                 smoo1 = smooth(xMix(1:end-1),...
                                       PDF,0.03);
660                                 % plot cdf,pdf,smoothed-pdf
661                                 figure('Name',['Debug: ',...
662                                     char(obj.distributionName)...
                                           ])
663                                 subplot(2,1,1)
664                                 hold on
665                                 plot(xMix,CDF,'-m')
666                                 plot(xMix(1:end-1),PDF,'-r')
667                                 plot(xMix(1:end-1),smoo1,'-b')
668                                 ylabel('$f(x) or F(x)$','...
                                       Interpreter','latex')
669                                 xlabel('x','Interpreter','...
                                       latex')
670                                 legend('cdf','pdf','smoothed-...
                                       pdf')
671                                 % plot histogram for random ...
                                       sample
672                                 subplot(2,1,2)
```

```
673                             histogram(random(obj.distInfo...
                                   ,1000,1),...
674                                 'Normalization','...
                                       probability')
675                         end
676                     end
677                     %}
678                     %-------
679                 case 'Stable'
680                     % Stable Case Statement
681                     % First shape parameter
682                     Alpha = 0.5;
683                     %{
684                     Alpha = 0.4;
685                     Alpha = 0.35
686                     Alpha = 0.5  %<---- use to start
687                     Alpha = 0.2; %<---- very hard to ...
                           estimate
688                     %}
689                     % Second shape parameter: -1 ≤ Beta ≤ ...
                           1
690                     Beta = 0.05;
691                     %{
692                     Beta = 0.9;
693                     Beta = 1;
694                     Beta = .05;
695                     Beta = .05;
696                     %}
697                     % Scale parameter
698                     Gam = 1;
699                     % Location parameter
700                     Delta = 4;
701                     % PDF Curve \
702                     obj.distInfo = makedist(obj....
                           distributionName,...
703                          'Alpha', Alpha,'Beta', Beta,...
704                          'Gam', Gam, 'Delta', Delta);
705                     obj.pdfCurve = pdf(obj.distInfo, obj.x...
                           );
706                     %\
707                     % generate random sample or actual pdf
708                     if obj.randomVSactual == "random"
709                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
710                     elseif obj.randomVSactual == "actual"
711                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
```

```
712                     end
713                 case 'Stable1'
714                     % Stable Case Statement
715                     % First shape parameter
716                     Alpha = 0.2;
717                     %{
718                     Alpha = 0.4;
719                     Alpha = 0.35
720                     Alpha = 0.5  %<---- use to start
721                     Alpha = 0.2; %<---- very hard to ...
                           estimate
722                     %}
723                     % Second shape parameter: -1 ≤ Beta ≤ ...
                           1
724                     Beta = 0.05;
725                     %{
726                     Beta = 0.9;
727                     Beta = 1;
728                     Beta = .05;
729                     Beta = .05;
730                     %}
731                     % Scale parameter
732                     Gam = 1;
733                     % Location parameter
734                     Delta = 4;
735                     % PDF Curve \
736                     distributionLabel = 'Stable';
737                     obj.distInfo = makedist(...
                           distributionLabel,...
738                         'Alpha', Alpha,'Beta', Beta,...
739                         'Gam', Gam, 'Delta', Delta);
740                     obj.pdfCurve = pdf(obj.distInfo, obj.x...
                           );
741                     %\
742                     % generate random sample or actual pdf
743                     if obj.randomVSactual == "random"
744                         rndData = random(obj.distInfo,1,...
                               obj.Ns);
745                     elseif obj.randomVSactual == "actual"
746                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
747                     end
748                 case 'Stable2'
749                     % mixture model for 2 stable ...
                           distributions
750                     % mixture weights
751                     p1 = 0.25;
```

```
752                         p2 = 1 - p1;
753                         p = [p1,p2];
754                         % First shape parameter
755                         Alpha1 = 0.5;
756                         Alpha2 = 0.5;
757                         % Second shape parameter: -1 ≤ Beta ≤ ...
                               1
758                         Beta1 = 0.05;
759                         Beta2 = 0.05;
760                         % Scale parameter
761                         Gam1 = 1;
762                         Gam2 = 1;
763                         % Location parameter
764                         Delta1 = 2;
765                         Delta2 = 5;
766                         % PDF Curve \
767                         distributionLabel = 'Stable';
768                         % stable 1
769                         distInfo1 = makedist(distributionLabel...
                               ,...
770                              'Alpha', Alpha1,'Beta', Beta1,...
771                              'Gam', Gam1, 'Delta', Delta1);
772                         pdfCurve1 = pdf(distInfo1, obj.x);
773                         % stable 2
774                         distInfo2 = makedist(distributionLabel...
                               ,...
775                              'Alpha', Alpha2,'Beta', Beta2,...
776                              'Gam', Gam2, 'Delta', Delta2);
777                         pdfCurve2 = pdf(distInfo2, obj.x);
778                         % Mixture PDF Curve
779                         obj.pdfCurve = p(1)*pdfCurve1 + p(2)*...
                               pdfCurve2;
780                         %\
781                         % generate random sample or actual pdf
782                         if obj.randomVSactual == "random"
783                              % mixture string array flag for ...
                                    mixSampling()
784                              mixtureType = "two";
785                              % generate n vector for mixture ...
                                    samplings
786                              n = mixSampling(obj.Ns,p,...
                                    mixtureType);
787                              % generate random sample
788                              rndData1 = random(distInfo1,1,n(1)...
                                    );
789                              rndData2 = random(distInfo2,1,n(2)...
                                    );
```

```
790                         rndData = [rndData1,rndData2];
791                     elseif obj.randomVSactual == "actual"
792                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
793                     end
794                     % CREATE DISTRIBUTION OBJECT ...
                        ------------
795                     % mixture string array flag for ...
                        mixSampling()
796                     mixtureType = "two";
797                     % generate m vector for mixture ...
                        samplings
798                     m = mixSampling(10000,p,mixtureType);
799                     % generate random sample to create ...
                        distribution
800                     % object
801                     actData1 = random(distInfo1,1,m(1));
802                     actData2 = random(distInfo2,1,m(2));
803                     actData = [actData1,actData2];
804                     % generate numerical cdf: f=cdf, s=x-...
                        coordinates
805                     [f,s] = ecdf(actData);
806                     f = f(1:2:end);
807                     s = s(1:2:end);
808                     % generate distribution object
809                     obj.distInfo = ...
810                         makedist('PiecewiseLinear','x',s,'...
                               Fx',f);
811                     % create cdf/pdf from distribution ...
                        object.
812                     % for debugging and vizualization.
813                     if debug
814                         if obj.randomVSactual == "actual"
815                             xMix = linspace(obj.lowerLimit...
                                   ,...
816                                  obj.upperLimit,1000);
817                             CDF = cdf(obj.distInfo,xMix);
818                             % numerically differentiate
819                             PDF = zeros(1,size(CDF(1:end...
                                   -1),2));
820                             for i = 2:size(CDF,2)-1
821                                 dx1 = (xMix(i+1)-xMix(i-1)...
                                       );
822                                 PDF(i) =  (CDF(i+1) - CDF(...
                                       i-1))/dx1;
823                             end
824                             % smooth pdf data
```

```
825                             smoo1 = smooth(xMix(1:end-1),...
                                   PDF,0.03);
826                             % plot cdf,pdf,smoothed-pdf
827                             figure('Name',['Debug: ',...
828                                 char(obj.distributionName)...
                                       ])
829                             subplot(2,1,1)
830                             hold on
831                             plot(xMix,CDF,'-m')
832                             plot(xMix(1:end-1),PDF,'-r')
833                             plot(xMix(1:end-1),smoo1,'-b')
834                             ylabel('$f(x) or F(x)$','...
                                   Interpreter','latex')
835                             xlabel('x','Interpreter','...
                                   latex')
836                             legend('cdf','pdf','smoothed-...
                                   pdf')
837                             % plot histogram for random ...
                                   sample
838                             subplot(2,1,2)
839                             histogram(random(obj.distInfo...
                                   ,1000,1),...
840                                 'Normalization','...
                                       probability')
841                         end
842                     end
843                     %-------
844                 case 'Stable3'
845                     % mixture model for 3 stable ...
                           distributions
846                     % mixture weights
847                     p1 = 0.25;
848                     p2 = 0.5;
849                     p3 = 1 - p1 - p2;
850                     p = [p1,p2,p3];
851                     % Stable distributions ----
852                     % First shape parameter
853                     Alpha1 = 0.5;
854                     Alpha2 = 0.5;
855                     Alpha3 = 0.5;
856                     % Second shape parameter: -1 ≤ Beta ≤ ...
                           1
857                     Beta1 = 0.05;
858                     Beta2 = 0.05;
859                     Beta3 = 0.05;
860                     % Scale parameter
861                     Gam1 = 1;
```

```
862                     Gam2 = 1;
863                     Gam3 = 1;
864                     % Location parameter
865                     Delta1 = 2;
866                     Delta2 = 5;
867                     Delta3 = 8;
868                     % PDF Curve \
869                     distributionLabel = 'Stable';
870                     % stable 1
871                     distInfo1 = makedist(distributionLabel...
                            ,...
872                          'Alpha', Alpha1,'Beta', Beta1,...
873                          'Gam', Gam1, 'Delta', Delta1);
874                     pdfCurve1 = pdf(distInfo1,obj.x);
875                     % stable 2
876                     distInfo2 = makedist(distributionLabel...
                            ,...
877                          'Alpha', Alpha2,'Beta', Beta2,...
878                          'Gam', Gam2, 'Delta', Delta2);
879                     pdfCurve2 = pdf(distInfo2,obj.x);
880                     % stable 3
881                     distInfo3 = makedist(distributionLabel...
                            ,...
882                          'Alpha', Alpha3,'Beta', Beta3,...
883                          'Gam', Gam3, 'Delta', Delta3);
884                     pdfCurve3 = pdf(distInfo3,obj.x);
885                     % Mixture PDF Curve
886                     obj.pdfCurve = p(1)*pdfCurve1 +...
887                          p(2)*pdfCurve2 + p(3)*pdfCurve3;
888                     %\
889                     % generate random sample or actual pdf
890                     if obj.randomVSactual == "random"
891                          % mixture string array flag for ...
                                mixSampling()
892                          mixtureType = "three";
893                          % generate n vector for mixture ...
                                samplings
894                          n = mixSampling(obj.Ns,p,...
                                mixtureType);
895                          % generate random sample
896                          rndData1 = random(distInfo1,1,n(1)...
                                );
897                          rndData2 = random(distInfo2,1,n(2)...
                                );
898                          rndData3 = random(distInfo3,1,n(3)...
                                );
```

```
899                             rndData = [rndData1,rndData2,...
                                    rndData3];
900                         elseif obj.randomVSactual == "actual"
901                             data = vertcat(obj.x,obj.pdfCurve)...
                                    ;
902                         end
903                         % CREATE DISTRIBUTION OBJECT ...
                             ------------
904                         % mixture string array flag for ...
                             mixSampling()
905                         mixtureType = "three";
906                         % generate m vector for mixture ...
                             samplings
907                         m = mixSampling(10000,p,mixtureType);
908                         % generate random sample to create ...
                             distribution
909                         % object
910                         actData1 = random(distInfo1,1,m(1));
911                         actData2 = random(distInfo2,1,m(2));
912                         actData3 = random(distInfo3,1,m(3));
913                         actData = [actData1,actData2,actData...
                             3];
914                         % generate numerical cdf: f=cdf, s=x-...
                             coordinates
915                         [f,s] = ecdf(actData);
916                         f = f(1:2:end);
917                         s = s(1:2:end);
918                         % generate distribution object
919                         obj.distInfo = ...
920                             makedist('PiecewiseLinear','x',s,'...
                                Fx',f);
921                         % create cdf/pdf from distribution ...
                             object.
922                         % for debugging and vizualization.
923                         if debug
924                             if obj.randomVSactual == "actual"
925                                 xMix = linspace(obj.lowerLimit...
                                        ,...
926                                     obj.upperLimit,1000);
927                                 CDF = cdf(obj.distInfo,xMix);
928                                 % numerically differentiate
929                                 PDF = zeros(1,size(CDF(1:end...
                                        -1),2));
930                                 for i = 2:size(CDF,2)-1
931                                     dx1 = (xMix(i+1)-xMix(i-1)...
                                            );
```

```
932                                 PDF(i) =  (CDF(i+1) - CDF(...
                                       i-1))/dx1;
933                             end
934                             % smooth pdf data
935                             smoo1 = smooth(xMix(1:end-1),...
                                   PDF,0.03);
936                             % plot cdf,pdf,smoothed-pdf
937                             figure('Name',['Debug: ',...
938                                 char(obj.distributionName)...
                                       ])
939                             subplot(2,1,1)
940                             hold on
941                             plot(xMix,CDF,'-m')
942                             plot(xMix(1:end-1),PDF,'-r')
943                             plot(xMix(1:end-1),smoo1,'-b')
944                             ylabel('$f(x) or F(x)$','...
                                   Interpreter','latex')
945                             xlabel('x','Interpreter','...
                                   latex')
946                             legend('cdf','pdf','smoothed-...
                                   pdf')
947                             % plot histogram for random ...
                                   sample
948                             subplot(2,1,2)
949                             histogram(random(obj.distInfo...
                                   ,1000,1),...
950                                 'Normalization','...
                                       probability')
951                         end
952                     end
953                     %-------
954                 case 'Trimodal-Normal'
955                     % Normal Case Statement
956                     % mixture weights
957                     p1 = 0.33;
958                     p2 = 0.33;
959                     p3 = 1 - p1 - p2;
960                     p = [p1,p2,p3];
961                     % Mean
962                     Mu1 = 4;
963                     Mu2 = 5;
964                     Mu3 = 6;
965                     % Standard deviation
966                     Sigma1 = 0.5;
967                     Sigma2 = 0.25;
968                     Sigma3 = 0.5;
969                     % PDF Curve \
```

```
970                     % Distribution 1
971                     distributionLabel1 = 'Normal';
972                     distInfo1 = makedist(distributionLabel...
                           1,...
973                         'Mu', Mu1, 'Sigma', Sigma1);
974                     pdfCurve1 = pdf(distInfo1,obj.x);
975                     % Distribution 2
976                     distInfo2 = makedist(distributionLabel...
                           1,...
977                         'Mu', Mu2, 'Sigma', Sigma2);
978                     pdfCurve2 = pdf(distInfo2,obj.x);
979                     % Distribution 3
980                     distInfo3 = makedist(distributionLabel...
                           1,...
981                         'Mu', Mu3, 'Sigma', Sigma3);
982                     pdfCurve3 = pdf(distInfo3,obj.x);
983                     % Mixture PDF Curve
984                     obj.pdfCurve = p(1)*pdfCurve1 +...
985                         p(2)*pdfCurve2 + p(3)*pdfCurve3;
986                     %\
987                     % generate random sample or actual pdf
988                     if obj.randomVSactual == "random"
989                         % mixture string array flag for ...
                               mixSampling()
990                         mixtureType = "three";
991                         % generate n vector for mixture ...
                               samplings
992                         n = mixSampling(obj.Ns,p,...
                               mixtureType);
993                         % generate random sample
994                         rndData1 = random(distInfo1,1,n(1)...
                               );
995                         rndData2 = random(distInfo2,1,n(2)...
                               );
996                         rndData3 = random(distInfo3,1,n(3)...
                               );
997                         rndData = [rndData1,rndData2,...
                               rndData3];
998                     elseif obj.randomVSactual == "actual"
999                         data = vertcat(obj.x,obj.pdfCurve)...
                               ;
1000                    end
1001                    % CREATE DISTRIBUTION OBJECT ...
                           ------------
1002                    % mixture string array flag for ...
                           mixSampling()
1003                    mixtureType = "three";
```

```
1004                        % generate m vector for mixture ...
                               samplings
1005                        m = mixSampling(10000,p,mixtureType);
1006                        % generate random sample to create ...
                               distribution
1007                        % object
1008                        actData1 = random(distInfo1,1,m(1));
1009                        actData2 = random(distInfo2,1,m(2));
1010                        actData3 = random(distInfo3,1,m(3));
1011                        actData = [actData1,actData2,actData...
                               3];
1012                        % generate numerical cdf: f=cdf, s=x-...
                               coordinates
1013                        [f,s] = ecdf(actData);
1014                        f = f(1:2:end);
1015                        s = s(1:2:end);
1016                        % generate distribution object
1017                        obj.distInfo = ...
1018                            makedist('PiecewiseLinear','x',s,'...
                                   Fx',f);
1019                        % create cdf/pdf from distribution ...
                               object.
1020                        % for debugging and vizualization.
1021                        if debug
1022                            if obj.randomVSactual == "actual"
1023                                xMix = linspace(obj.lowerLimit...
                                       ,...
1024                                    obj.upperLimit,1000);
1025                                CDF = cdf(obj.distInfo,xMix);
1026                                % numerically differentiate
1027                                PDF = zeros(1,size(CDF(1:end...
                                       -1),2));
1028                                for i = 2:size(CDF,2)-1
1029                                    dx1 = (xMix(i+1)-xMix(i-1)...
                                           );
1030                                    PDF(i) =  (CDF(i+1) - CDF(...
                                           i-1))/dx1;
1031                                end
1032                                % smooth pdf data
1033                                smoo1 = smooth(xMix(1:end-1),...
                                       PDF,0.03);
1034                                % plot cdf,pdf,smoothed-pdf
1035                                figure('Name',['Debug: ',...
1036                                    char(obj.distributionName)...
                                           ])
1037                                subplot(2,1,1)
1038                                hold on
```

```
1039                            plot(xMix,CDF,'-m')
1040                            plot(xMix(1:end-1),PDF,'-r')
1041                            plot(xMix(1:end-1),smoo1,'-b')
1042                            ylabel('$f(x) or F(x)$','...
                                   Interpreter','latex')
1043                            xlabel('x','Interpreter','...
                                   latex')
1044                            legend('cdf','pdf','smoothed-...
                                   pdf')
1045
1046                            % plot histogram for random ...
                                   sample
1047                            subplot(2,1,2)
1048                            histogram(random(obj.distInfo...
                                   ,1000,1),...
1049                                'Normalization','...
                                       probability')
1050                        end
1051                    end
1052                    %-------
1053                case 'tLocationScale'
1054                    % t-Location Scale Case Statement
1055                    % Location parameter
1056                    Mu = 4;
1057                    % Scale parameter
1058                    Sigma = .05;
1059                    % Shape parameter
1060                    Nu = 1;
1061                    % Location parameter
1062                    % Delta = 3;
1063                    % PDF Curve \
1064                    obj.distInfo = makedist(obj....
                           distributionName,...
1065                        'Mu', Mu, 'Sigma', Sigma, 'Nu', Nu...
                               );
1066                    obj.pdfCurve = pdf(obj.distInfo, obj.x...
                           );
1067                    %\
1068                    % generate random sample or actual pdf
1069                    if obj.randomVSactual == "random"
1070                        rndData = random(obj.distInfo,1,...
                               obj.Ns);
1071                    elseif obj.randomVSactual == "actual"
1072                        data = vertcat(obj.x,obj.pdfCurve)...
                               ;
1073                    end
1074                case 'Uniform'
```

```
1075                    % Uniform Case Statement
1076                    % Lower bound
1077                    Lower = 4;
1078                    % Upper Bound
1079                    Upper = 8;
1080                    % PDF Curve \
1081                    obj.distInfo = makedist(obj....
                           distributionName,...
1082                        'Lower', Lower, 'Upper', Upper);
1083                    obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                           ;
1084                    %\
1085                    % generate random sample or actual pdf
1086                    if obj.randomVSactual == "random"
1087                        rndData = random(obj.distInfo,1,...
                               obj.Ns);
1088                    elseif obj.randomVSactual == "actual"
1089                        data = vertcat(obj.x,obj.pdfCurve)...
                               ;
1090                    end
1091                case 'Uniform-Mix'
1092                    % Uniform Case Statement
1093                    % mixture weights
1094                    p1 = 0.1;
1095                    p2 = 0.6;
1096                    p3 = 1 - p1 - p2;
1097                    p = [p1,p2,p3];
1098                    % Lower bound
1099                    Lower1 = 1;
1100                    Lower2 = 3.5;
1101                    Lower3 = 7;
1102                    % Upper Bound
1103                    Upper1 = 2;
1104                    Upper2 = 5.5;
1105                    Upper3 = 9;
1106                    % PDF Curve \
1107                    % Distribution 1
1108                    distributionLabel1 = 'Uniform';
1109                    distInfo1 = makedist(distributionLabel...
                           1,...
1110                        'Lower', Lower1, 'Upper', Upper1);
1111                    pdfCurve1 = pdf(distInfo1,obj.x);
1112                    % Distribution 2
1113                    distInfo2 = makedist(distributionLabel...
                           1,...
1114                        'Lower', Lower2, 'Upper', Upper2);
1115                    pdfCurve2 = pdf(distInfo2,obj.x);
```

```
1116                    % Distribution 3
1117                    distInfo3 = makedist(distributionLabel...
                           1,...
1118                        'Lower', Lower3, 'Upper', Upper3);
1119                    pdfCurve3 = pdf(distInfo3,obj.x);
1120                    % Mixture PDF Curve
1121                    obj.pdfCurve = p(1)*pdfCurve1 +...
1122                        p(2)*pdfCurve2 + p(3)*pdfCurve3;
1123                    %\
1124                    % generate random sample or actual pdf
1125                    if obj.randomVSactual == "random"
1126                        % mixture string array flag for ...
                              mixSampling()
1127                        mixtureType = "three";
1128                        % generate n vector for mixture ...
                              samplings
1129                        n = mixSampling(obj.Ns,p,...
                              mixtureType);
1130                        % generate random sample
1131                        rndData1 = random(distInfo1,1,n(1)...
                              );
1132                        rndData2 = random(distInfo2,1,n(2)...
                              );
1133                        rndData3 = random(distInfo3,1,n(3)...
                              );
1134                        rndData = [rndData1,rndData2,...
                              rndData3];
1135                    elseif obj.randomVSactual == "actual"
1136                        data = vertcat(obj.x,obj.pdfCurve)...
                              ;
1137                    end
1138                    % CREATE DISTRIBUTION OBJECT ...
                           ------------
1139                    % mixture string array flag for ...
                           mixSampling()
1140                    mixtureType = "three";
1141                    % generate m vector for mixture ...
                           samplings
1142                    m = mixSampling(10000,p,mixtureType);
1143                    % generate random sample to create ...
                           distribution
1144                    % object
1145                    actData1 = random(distInfo1,1,m(1));
1146                    actData2 = random(distInfo2,1,m(2));
1147                    actData3 = random(distInfo3,1,m(3));
1148                    actData = [actData1,actData2,actData...
                           3];
```

```
1149                    % generate numerical cdf: f=cdf, s=x-...
                           coordinates
1150                    [f,s] = ecdf(actData);
1151                    f = f(1:2:end);
1152                    s = s(1:2:end);
1153                    % generate distribution object
1154                    obj.distInfo = ...
1155                        makedist('PiecewiseLinear','x',s,'...
                               Fx',f);
1156                    % create cdf/pdf from distribution ...
                           object.
1157                    % for debugging and vizualization.
1158                    if debug
1159                        if obj.randomVSactual == "actual"
1160                            xMix = linspace(obj.lowerLimit...
                                   ,...
1161                                obj.upperLimit,1000);
1162                            CDF = cdf(obj.distInfo,xMix);
1163                            % numerically differentiate
1164                            PDF = zeros(1,size(CDF(1:end...
                                   -1),2));
1165                            for i = 2:size(CDF,2)-1
1166                                dx1 = (xMix(i+1)-xMix(i-1)...
                                       );
1167                                PDF(i) =  (CDF(i+1) - CDF(...
                                       i-1))/dx1;
1168                            end
1169                            % smooth pdf data
1170                            smoo1 = smooth(xMix(1:end-1),...
                                   PDF,0.03);
1171                            % plot cdf,pdf,smoothed-pdf
1172                            figure('Name',['Debug: ',...
1173                                char(obj.distributionName)...
                                       ])
1174                            subplot(2,1,1)
1175                            hold on
1176                            plot(xMix,CDF,'-m')
1177                            plot(xMix(1:end-1),PDF,'-r')
1178                            plot(xMix(1:end-1),smoo1,'-b')
1179                            ylabel('$f(x) or F(x)$','...
                                   Interpreter','latex')
1180                            xlabel('x','Interpreter','...
                                   latex')
1181                            legend('cdf','pdf','smoothed-...
                                   pdf')
1182                            % plot histogram for random ...
                                   sample
```

```
1183                            subplot(2,1,2)
1184                            histogram(random(obj.distInfo...
                                   ,1000,1),...
1185                                'Normalization','...
                                       probability')
1186                        end
1187                    end
1188                    %-------
1189                case 'Weibull'
1190                    % Weibull Case Statement
1191                    % Scale parameter
1192                    a = 1;
1193                    % Shape parameter
1194                    b = 2;
1195                    % PDF Curve \
1196                    obj.distInfo = ...
1197                        makedist(obj.distributionName,'a',...
                                a, 'b', b);
1198                    obj.pdfCurve = pdf(obj.distInfo,obj.x)...
                       ;
1199                    %\
1200                    % generate random sample or actual pdf
1201                    if obj.randomVSactual == "random"
1202                        rndData = random(obj.distInfo,1,...
                               obj.Ns);
1203                    elseif obj.randomVSactual == "actual"
1204                        data = vertcat(obj.x,obj.pdfCurve)...
                               ;
1205                    end
1206                otherwise
1207                    % Warning Statement
1208                    warning('No distribution was picked')
1209            end
1210            % Create data file
1211            if obj.randomVSactual == "random"
1212                dataCreation(rndData,obj.fileName,obj....
                       precision,1)
1213            elseif obj.randomVSactual == "actual"
1214                dataCreation(data,obj.fileName,obj....
                       precision,1)
1215            end
1216            % Create folder for distribution data \\\
1217            % Define folder name
1218            folderName = sprintf(['D_', char(obj....
                   distributionName)]);
1219            % If folder already exist don't make it again
1220            if exist(folderName,'dir') == 0
```

```
1221                 mkdir(char(folderName))
1222             end
1223             %\\\\\\\\\
1224             % Move datafile to folder
1225             if exist([char(obj.fileName),'.txt'],'file') ...
                    == 2
1226                 movefile([char(obj.fileName),'.txt'] ,char...
                        (folderName));
1227             end
1228         end
1229     end
1230 end
```

C.2.3 mixSampling.m

```
 1 function n = mixSampling(N,p,mixtureType)
 2 % Probability Distribution Data Generation function
 3 % Created By: Zach D. Merino a MS candidate
 4 % Updated: 3/22/19
 5 %-------------
 6 % This function generates the size of the sample to be ...
     taken from each
 7 % individual distrbution in a mixture distribution. This ...
     method uses
 8 % random sampling from a binomial distribution. This ...
     method can easily be
 9 % generalized to any size mixture, but for practial use ...
     the option to
10 % created a mixture from 2-5 has been included.
11 %-------------
12 % n = vector of subsamples for each distribution in the ...
     mixture
13 % N = sample size to take from total mixture distribution
14 % p = vector of probability weights for each distribution ...
     in the mixture
15
16 switch mixtureType
17     case "two"
18
19         % get random sample from binomial distribution
20         n1 = binornd(N,p(1));
21         % find sample points for last distribution in the ...
              mixture
22         n2 = N - n1;
```

```
23          % save number of samples to take from each ...
               distribution
24          n = [n1,n2];
25
26      case "three"
27
28          % get random sample from binomial distribution
29          n1 = binornd(N,p(1));
30          % get random sample from binomial distribution ...
               with conditional
31          % probabilies
32          n2 = binornd(N-n1,p(2)/(p(2)+p(3)));
33          % find sample points for last distribution in the ...
               mixture
34          n3 = N - n1 - n2;
35          % save number of samples to take from each ...
               distribution
36          n = [n1,n2,n3];
37
38      case "four"
39
40          % get random sample from binomial distribution
41          n1 = binornd(N,p(1));
42          % get random sample from binomial distribution ...
               with conditional
43          % probabilies
44          n2 = binornd(N-n1,p(2)/(p(2)+p(3)+p(4)));
45          n3 = binornd(N-n1-n2,p(3)/(p(3)+p(4)));
46          % find sample points for last distribution in the ...
               mixture
47          n4 = N - n1 - n2 - n3;
48          % save number of samples to take from each ...
               distribution
49          n = [n1,n2,n3,n4];
50
51      case "five"
52
53          % get random sample from binomial distribution
54          n1 = binornd(N,p(1));
55          % get random sample from binomial distribution ...
               with conditional
56          % probabilies
57          n2 = binornd(N-n1,p(2)/(p(2)+p(3)+p(4)+p(5)));
58          n3 = binornd(N-n1-n2,p(3)/(p(3)+p(4)+p(5)));
59          n4 = binornd(N-n1-n2-n3,p(4)/(p(4)+p(5)));
60          % find sample points for last distribution in the ...
               mixture
```

```
61          n5 = N - n1 - n2 - n3 - n4;
62          % save number of samples to take from each ...
               distribution
63          n = [n1,n2,n3,n4,n5];
64
65      case "six"
66
67          % get random sample from binomial distribution
68          n1 = binornd(N,p(1));
69          % get random sample from binomial distribution ...
               with conditional
70          % probabilies
71          n2 = binornd(N-n1,p(2)/(p(2)+p(3)+p(4)+p(5)+p(6)))...
               ;
72          n3 = binornd(N-n1-n2,p(3)/(p(3)+p(4)+p(5)+p(6)));
73          n4 = binornd(N-n1-n2-n3,p(4)/(p(4)+p(5)+p(6)));
74          n5 = binornd(N-n1-n2-n3-n4,p(5)/(p(5)+p(6)));
75          % find sample points for last distribution in the ...
               mixture
76          n6 = N - n1 - n2 - n3 - n4 - n5;
77          % save number of samples to take from each ...
               distribution
78          n = [n1,n2,n3,n4,n5,n6];
79  end
```

C.2.4 dataCreation.m

```
1  function dataCreation(data,fileName,percision,dimIndex)
2  % Probability Distribution Data Generation function
3  % Created By: Zach D. Merino a MS candidate
4  % Updated: 8/3/18
5  %-------------
6  % Comments with no leading space are for diagnostic ...
      purposes.
7  % The function creates a collimated data.txt file with a ...
      specific precision
8  % from the data and fileName variables.
9  %-------------
10 % Initialize column spacing
11 num_column = '\r\n';
12 % Loop for text file set up
13 % dimIndex can vary depending on data format
14 for i =  1:size(data,dimIndex)
15     num_column = ['%.',num2str(percision),'g ' num_column...
          ];
```

```
end
% Define full file name and type
full_name = [fileName ,'.txt'];
% Generate txt with double precision error output
[file_id, msg] = fopen(full_name,'w');
if file_id < 0
    warning(['errorID: ', file_id])
    warning(['errorMSG:', msg])
end
fprintf(file_id, num_column, data);
fclose(file_id);
end
```