TABLE METHODS FOR RANDOM VARIATE GENERATION

by

İsmail Başoğlu

B.S., in Industrial Engineering, Yıldız Technical University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Industrial Engineering
Boğaziçi University

2008

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge the enthusiastic supervision of Assoc. Prof. Wolfgang Hörmann. His orientation, guidance, insightful criticisms and patient encouragement contributed very much for the realization of this thesis.

I would also like to acknowledge the support of TÜBİTAK, the unique research foundation in Turkey, with the Domestic Master Study Scholarship Program.

During this work, I have collaborated with many colleagues for whom I have great regard and I want to express that I thank much to those who have helped me with my work besides providing a stimulating and fun environment.

Finally, I want to specially thank to my favorite supporters at every time: my dear mum Mahmude Başoğlu, dear dad Hayri Başoğlu and other family members, Nalan Gürşahbaz, Nazan Uyanık, Emrah Gürşahbaz and Fatih Uyanık who have been my mentors since I started my undergraduate education in Industrial Engineering.

ABSTRACT

TABLE METHODS FOR RANDOM VARIATE GENERATION

For stochastic simulation, the generation of random variates from different distributions is a prerequisite. In certain programming languages and software, there are already random variate generation functions of standard distributions. However, for generating random variates from non-standard distributions or quasi-densities, we need universal algorithms. In this research, we come up with two universal random variate generation methods, namely the Triangular Ahrens and the Polynomial Density Inversion. We try to see if they are competitive with existing methods with respect to simplicity, speed and other performance criteria. After explaining the basics of the algorithms, we define the pseudo-codes in detail. Both of the algorithms are coded in C in a comprehensible and elegant way. Numerical results indicate that both of the algorithms execute with a successful performance. The Triangular Ahrens, which is a rejection method, has a smaller rejection constant while it requires smaller tables. The Polynomial Density Inversion, which approximates the density with piecewise polynomials, is more complicated however we obtain outstanding approximations with smaller tables. It also has a faster marginal execution time which makes the Polynomial Density Inversion a preferable method for a large number of random variates.

ÖZET

RASSAL DEĞİŞKEN ÜRETİMİ ÜZERİNE TABLO YÖNTEMLERİ

Farklı dağılımlar üzerinden rassal değişken üretimi, stokastik benzetim için önkoşul oluşturmaktadır. Belirli programlama dilleri ve yazılımlar, standart dağılımlar için rassal değişken üretimini destekleyen fonksiyonlar barındırmaktadırlar. Yine de, standart olmayan dağılımlar ve yarı dağılımlardan rassal değişken üretebilmek için otomatik algoritmalara ihtiyaç duyulmaktadır. Bu araştırmada, Üçgensel Ahrens ve Polinomlu Yoğunluk Fonksiyonunun Ters Dönüşümü adında iki adet otomatik rassal değişken üretim yöntemi sunulmuştur. Bunların mevcut yöntemlerle yalınlık, hız ve diğer performans ölçütleri üzerinden kıyaslanabilirliği incelenmiştir. Algoritmaların temel içeriği açıklandıktan sonra, sözde kodları ayrıntılarla verilmiştir. Her iki algoritma da C programlama dili kullanılarak düzenli ve anlaşılabilir bir şekilde kodlanmıştır. Sayısal sonuçlar her iki algoritmanın da başarılı bir performans sergilediğini göstermektedir. Değişken reddetme yöntemi olan Üçgensel Ahrens daha küçük tablolar yardımıyla daha düşük reddetme katsayılarına ulaşmaktadır. Olasılık yoğunluk fonksiyonunu parçalar halinde polinomlara yaklaştıran Polinomlu Yoğunluk Fonksiyonunun Ters Dönüşümü ise daha karmaşık olmasına rağmen daha küçük tablolar yardımıyla göze çarpan yaklaşıklıklara ulaşmaktadır. Ayrıca, tek rassal değişken üretim zamanının daha hızlı olması Polinomlu Yoğunluk Fonksiyonunun Ters Dönüşümü'nü daha yüksek sayıda rassal değişken üretimi için seçkin kılmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

A_f	The area below the function
A_h	The area below the hat
A_s	The area below the squeeze
C	Size of the Guide Table
G_f	Region between the density function and the x-axis
I	Number of iterations until a successful iteration
K	Order of the approximation polynomial
M	Constant hat
N	Number of subintervals in a distribution
S	Constant squeeze
T	Total execution time of a random variate generation algorithm
T_I	Execution time of a single iteration in rejection
T_S	Execution time of the setup algorithm
$U(0,1)$	Standard uniform random variate
X	Random variate
Z	Sequential vector of local extrema, cutoff and inflection points
a_c	Critical area for the region between linear hat and constant squeeze
a_p	Area of the polynomial region
a_q	Questioned area for the region between the linear hat and the constant squeeze
a_r	Area of the rectangular region
a_t	Area of the triangular region
b	Elements of the vector of subinterval boundaries
b_l	Left boundary of a bounded domain
b_r	Right boundary of a bounded domain
c	Center point of an interval

ca	Unnormalized cumulative area of the subintervals
co	Elements of a coefficient vector of the Newton Interpolation Polynomial
cs	Constant squeeze for the decomposition
d	Divided differences
f_p	Polynomial quasi-density
f_r	Rectangular quasi-density
f_t	Triangular quasi-density
g	Elements of a Guide Table
h	Hat function
lh	Linear hat for the decomposition
lhi	Intercept of a linear hat
lhs	Slope of a linear hat
ls	Linear squeeze for the decomposition
lsi	Intercept of a linear squeeze
lss	Slope of a linear squeeze
r	Relative linear approximation error in the center point
r_c	Critical relative linear approximation error in the center point
rc	Cumulative probability of rectangular region in a subinterval
s	Squeeze function
tr	Cumulative probability of triangular region in a subinterval
w	Probabilistic weight of decomposed regions
$\tilde{x}'$	Chebyshev nodes for interpolation
$\hat{x}$	Chebyshev control points for maximum absolute approximation error
α	Rejection constant
ε	Linear approximation error in the center point
ε_c	Critical linear approximation error in the center point
ε'	Polynomial error – maximum absolute approximation error over Chebyshev control points
ε'_c	Critical polynomial error – maximum absolute approximation error over Chebyshev control points

ρ_{hs}	The ratio of the area below the hat and the area below the squeeze
cdf	Cumulative Density Function
pdf	Probability Density Function
CCP	Chebyshev Control Points
CPE	Critical Polynomial Error
CRE	Critical Relative Linear Error
ECP	Equidistant Control Points
LE	Linear Error
PE	Polynomial Error
RE	Relative Linear Error

1. INTRODUCTION

Stochastic simulation is a well-known tool appreciated in many fields of research and application. To use stochastic simulation, the generation of random variates from different distributions is a prerequisite. Non-uniform random variates are generated by transforming a sequence of independent uniform random variates into a sequence of independent random variates of the desired distribution and this is legitimate under the assumption that we have a source of truly uniform, independent and identically distributed random variates available (Hörmann *et al.*, 2004).

In general, stochastic simulation requires standard distributions like Normal, Gamma, Beta, Weibull distributions. In certain programming languages and software, there are already random variate generation functions of those distributions. However, there may be cases where the user is interested in random variates generated from non-standard distributions or quasi-densities, which do not have any existing generation methods. The best solution for this problem is to build generation algorithms that can sample from any given distribution with sufficient information about it (e.g.) the density or the cumulative distribution function; often some other information like the mode of the distribution is required. These algorithms are called universal (also called automatic or black box) generators, a single program that can sample from a large family of distributions (Hörmann *et al.*, 2004).

A universal generator typically starts with a setup that computes all constants necessary for the generation. In the sampling part of the program these constants are used to generate random variates. Well-known instances for universal algorithms are Transformed Density Rejection (Hörmann, 1995) and the Ahrens Method (Ahrens, 1995). Especially, the Ahrens Method can be applied to a large class of distributions with a bounded domain.

In this research, we come up with two ideas and two algorithms for improving the Ahrens method. The first one uses linear hats and applies “the mirroring principle” of Hörmann and Leydold (2007). Thus, we hope to decrease the rejection constant and the

expected number of density function calls, which are important performance characteristic for rejection based algorithms.

The second idea is based on numerical approximation of the density and can be seen as an improvement of the first algorithm without rejection (also called “Piecewise Linear Approximation” in the literature (Hörmann and Leydold, 2007). Yet, in order to increase the precision of the approximation, we can use higher order polynomial approximations with effective interpolation tools. Then, by using decomposition and numerical inversion, we can generate random variates with simple arithmetic operations and without any density function calls.

The aim of this thesis is to develop all the details of these new algorithms, code them and find out if they are competitive with existing methods with respect to simplicity, speed and other performance criteria. The organization of the thesis is as follows: First, Chapter 2 gives general information about the basics of random variate generation. Then, the Triangular Ahrens Method is explained and computational results are shown in Chapter 3. In Chapter 4, the piecewise constant and the piecewise linear approximation methods are criticized to see why higher order polynomial approximations are necessary. Then, the basics of numerical approximation are summarized in Chapter 5. In Chapter 6, the Polynomial Density Inversion method is introduced and explained. In addition, computational results are shown and the approximation performance is examined. And finally; conclusions, Appendix and references are set at the end of the thesis, respectively.

2. BASICS ON RANDOM VARIATE GENERATION

This chapter includes some basic ideas and methods for continuous and discrete random variate generation. Many of the universal algorithms are built on the base of these methods and gain efficiency by combining these basic ideas. The new algorithms, which can be classified as table methods, also rely on the idea of three basic methods: Inversion, Rejection and Composition.

2.1. The Inversion Method

The Inversion method is based on the following theorem.

Theorem 2.1: *Let* $F(x)$ *be a continuous cumulative distribution function (cdf) and* U *a uniform random number over* $[0,1]$. *Then the random variate* $X = F^{-1}(U)$ *has the cdf* F. *Furthermore, if* X *has cdf* F, *then* $F(X)$ *is uniformly distributed.*

Proof. The cdf is known to be monotonically increasing, thus:

$$P(X \le x) = P(F^{-1}(U) \le x) = P(U \le F(X)) = F(x)$$

and this means that $F(x)$ is the cdf of X. The second statement immediately follows from;

$$P(F(x) \le u) = P(X \le F^{-1}(u)) = F(F^{-1}(u)) = u$$

which completes the proof (Hörmann *et al.*, 2004).

Applying this principle to an invertible continuous cdf function allows generating random variates from the corresponding density by using a uniform random variate generator.

Require: Inverse of the cumulative density function $F^{-1}(x)$.
Output: Random variate X with cdf F.
1: Generate $U \sim U(0,1)$.
2: $X \leftarrow F^{-1}(U)$
3: **return** X.

Figure 2.1. Inversion algorithm

2.2. The Rejection Method

2.2.1. Basic Rejection

A well-known method of generating random variates from a density f over a bounded domain $[b_l, b_r]$ is rejection from a uniform hat, which consist of generating a uniform random variate X in the bounded domain and a uniform variate Y in $[0, M]$ where M is a constant and the uniform hat of the density with the property;

$$M \geq f(x); \quad \forall x \in [b_l, b_r]$$

Thus, any (X, Y) generated with these properties is a random point which is uniformly distributed in the rectangular region $(b_l, b_r) \times (0, M)$.

The fundamental property that if a random point (X, Y) is uniformly distributed in the region G_f between the graph of the density function f and the x-axis, then X has density f, which is formulated in the theorem (Hörmann *et al.*, 2004).

Theorem 2.2: *Let $f(x)$ be an arbitrary density and α some positive constant. If the random pair (X, Y) is uniformly distributed on the set*

$$G_{\alpha f} = \{(x, y) : 0 < y \leq \alpha f(x)\}$$

Then X is a random variate with density $f(x)$.

Proof. The random pair (X,Y) has by assumption the joint density: $f(x,y)=\frac{1}{\alpha}$ for $(x,y)\in G_{af}$ and zero elsewhere. Then the marginal density $f_X(x)$ of X is given by

$$f_X(x)=\int_0^{\alpha f(x)}\frac{1}{\alpha}dt = f(x)$$

which completes the proof (Hörmann *et al.*, 2004).

Thus, any random point that falls below the density function can be accepted and the rest can be rejected. The X values of the accepted points follow the desired distribution.

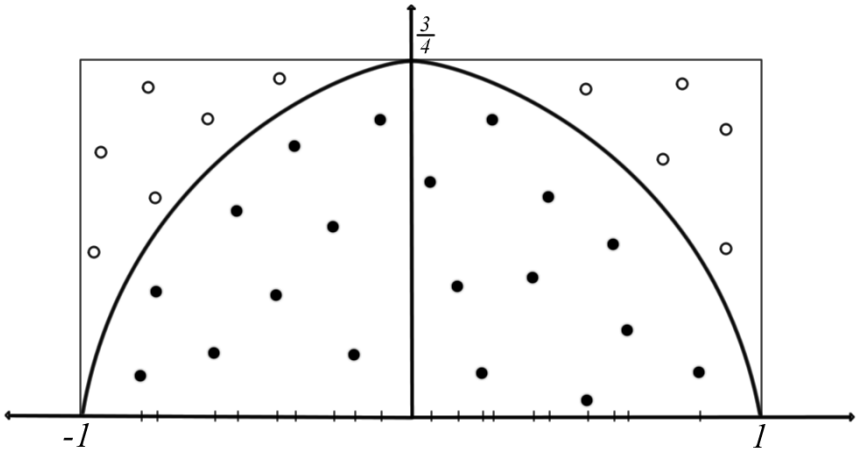


Figure 2.2. Rejection from a constant hat, density $f(x)=\frac{3}{4}(-x^2+1)$ on $[-1,1]$. 30 points are generated, 11 are rejected, the x-coordinates of the remaining points are returned

Theorem 2.2 implies that another property of the rejection method is that any multiple of the density can be used instead of the density itself. In other words, with the rejection algorithm, random variates of a density multiplied with any constant can be generated. Such a function is called quasi-density in Hörmann *et al.* (2004).

Require: Quasi-density $f(x)$ on a bounded domain $[b_l, b_r]$, upper bound (constant hat) $M \geq f(x)$.
Output: Random variate X with density proportional to f.
1: **repeat**
2: Generate $U \sim U(0,1)$.
3: $X \leftarrow b_l + (b_r - b_l)U$
4: Generate $V \sim U(0,1)$.
5: $Y \leftarrow MV$
6: **until** $Y \leq f(X)$.
7: **return** X.

Figure 2.3. Rejection from uniform hat

2.2.2. Rejection with Inversion

Instead of a uniform hat, it is possible to use a hat function $h(x)$ with $H(x)$ known. Thus, it is possible to generate random variates from the hat function by inversion. For each random variate X generated from the hat, another variate V is generated, which is uniformly distributed between zero and the corresponding hat value, $h(X)$. Those random variate pairs are uniformly distributed points in the region below the hat function. The rejection idea, then, can be applied with the original density.

Require: Quasi-density $f(x)$, hat function $h(x)$, inverse cdf $H^{-1}(x)$ of hat.
Output: Random variate X with density proportional to f.
1: **repeat**
2: Generate $U \sim U(0,1)$.
3: $X \leftarrow H^{-1}(U)$
4: Generate $V \sim U(0,1)$.
5: $Y \leftarrow Vh(X)$
6: **until** $Y \leq f(X)$.
7: **return** X.

Figure 2.4. Rejection with inversion

2.2.3. Squeeze Principle

Let f be a density on the bounded domain $[b_l, b_r]$. If f is a monotonically decreasing function, a simple upper bound (constant hat) can easily be obtained using the

density value of the left boundary, $f(b_l)$. Additionally, the density of the right boundary, $f(b_r)$, can be used as squeeze term, which allows us accepting a random point without comparing it with the corresponding density value. This enables random variate generation with less density function calls and less complicated mathematical operations. Therefore, the algorithm often executes faster.

Require: Quasi-density $f(x)$ on a bounded domain $[b_l, b_r]$, upper bound (constant hat) $M \geq f(x), \forall x \in [b_l, b_r]$, constant squeeze $S \leq f(x), \forall x \in [b_l, b_r]$.
Output: Random variate X with density proportional to f.
1: **loop**
2: Generate $U \sim U(0,1)$.
3: $X \leftarrow b_l + (b_r - b_l)U$
4: Generate $V \sim U(0,1)$.
5: $Y \leftarrow MV$
6: **If** $Y \leq S$ **then return** X.
7: **If** $Y \leq f(X)$ **then return** X.

Figure 2.5. Rejection with constant hat and constant squeeze

A constant hat on a bounded domain D can also be interpreted as the quasi-density of the uniform distribution on that domain. It can be replaced by some other appropriate quasi-density, $h(x)$, with the following property:

$$h(x) \geq f(x) \text{ for } \forall x \in D$$

Similarly, instead of using a constant squeeze, a squeeze function $s(x)$ with the following property can be used.

$$s(x) \leq f(x) \text{ for } \forall x \in D$$

These changes result in an increase of the performance, if;

- Random variates of the distribution, which has quasi-density $h(x)$, are easy to generate.
- $s(x)$ and quasi-density $h(x)$ are not expensive to evaluate.

- Rejection probability is smaller than before.

2.2.4. Performance Characteristics of the Rejection Method

It is clear that a rejection algorithm should have a small rejection probability. To evaluate the performance of the rejection algorithm, the key parameter is the acceptance probability $1/\alpha$ or its reciprocal value α called the rejection constant. It is calculated as the ratio

$$\alpha = A_h / A_f$$

of the area below the hat and the area below the density (Hörmann *et al.*, 2004). However, A_f is not equal to one for quasi-densities and might be unknown. Thus, another ratio

$$\rho_{hs} = A_h / A_s$$

of the area below the hat and the area below the squeeze can be introduced as a convenient upper bound for the rejection constant, which can be used when A_f is not known (Hörmann *et al.*, 2004).

Theorem 2.3: Let the number of iterations till acceptance in a rejection algorithm be denoted by I. Then we have:

$$P(I = i) = \frac{1}{\alpha}\left(1 - \frac{1}{\alpha}\right)^{i-1},\ E(I) = \alpha \le \rho_{hs},\ Var(I) = \alpha(\alpha - 1)$$

Denote the number of evaluations of the density f by $\#f$ and the uniform variates needed to generate one non-uniform variate by $\#U$. Then the expectations of those are given by;

$$E(\#U) = 2\alpha\,,\ E(\#\,f) = \frac{A_{h-s}}{A_f} \le \rho_{hs} - 1$$

where $A_{h-s} = A_h - A_s$.

Proof. Since the generated pairs (X,Y) are uniformly distributed on the region below the hat, the probability of acceptance is equal to the ratio of the areas A_f / A_h, which is equal to $1/\alpha$. As the uniform random numbers are assumed to be independent, then I follows a geometric distribution with parameter $1/\alpha$. Therefore, the expectation and the variance of I can be evaluated with the formulas of the geometric distribution.

As we need two uniform variates in a single iteration, we have:

$$E(\#U) = 2E(\#I) = 2\alpha$$

Two conditions should be considered to evaluate $E(\#\,f)$. Assume a random variate is finally accepted after I sequential iterations. If the generated random variate is accepted through squeeze, the density function was called $I-1$ times. If it is accepted through density comparison (in other words, if it is above the squeeze and below the density), the density function was called I times. By introducing the probability of those conditions and $E(I)$ as the expected number of iterations;

$$E(\#\,f) = (E(I)-1)\frac{A_s}{A_f} + E(I)\left(1 - \frac{A_s}{A_f}\right)$$

is obtained. Replacing $E(I)$ with A_h / A_f completes the proof (Hörmann *et al.*, 2004).

2.3. Composition

The main idea of the composition method is to decompose the domain into subintervals. For each subinterval, the region below the density can be used to evaluate the

probability that a random variate falls into that subinterval. If those subintervals are chosen with respect to behavior of the density, it is possible to speed up the generation algorithm by using computation formulas which are specific for that subinterval. Hörmann *et al.* (2004) restate this idea in a formal language by writing the target density f as with a discrete finite mixture:

$$f(x) = \sum_{i=1}^{n} w_i f_i(x)$$

In this formula, f_i are the given density functions and the w_i values form a probability vector, elements of which are larger than zero and sum up to one. In order to choose an index i, a discrete random variate I has to be generated first from that probability vector. That is possible with random variate generators for discrete distributions like the Indexed Search algorithm (See Section 2.5.2). Then the corresponding computations can be executed within subinterval i.

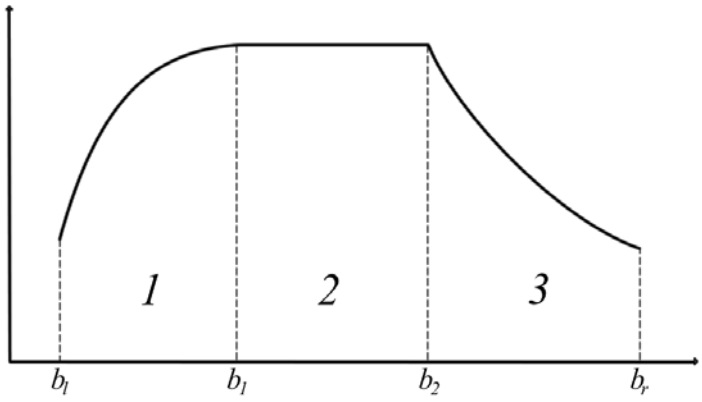


Figure 2.6. The density is decomposed into three parts

Require: Decomposition of density $f(x) = \sum_{i=1}^{n} w_i f_i(x)$; generators for each f_i.
Output: Random variate X with density f.
1: Generate discrete random variate J with probability vector (w_i).
2: Generate X with density f_J.
3: **return** X.

Figure 2.7. Composition

The composition method is especially useful when the distribution we are interested in is itself defined as a mixture. For other distributions it is rarely possible to find a simple mixture that is exactly the same as the desired distribution. Nevertheless decomposition of the area below the density into triangles and rectangles has been used to design fast but complicated generators for special distributions (Hörmann *et al.*, 2004).

2.3.1. Composition-Rejection

A very powerful application of the composition principle, however, is partitioning the domain of the distribution into subintervals. To obtain the probabilities w_i we need to know the area below the density. For the interval (b_i, b_{i+1}), computing the region below the density is possible only if both $F(b_i)$ and $F(b_{i+1})$ are available. However, for each subinterval, a hat function, $h(x)$, could be defined with $H(x)$ and $H^{-1}(x)$ known. Then we can calculate the area below the hat in the subinterval and "rejection with inversion" can be applied to the subinterval which is selected randomly with probabilities given by their respective areas below the hat.

Figure 2.8 illustrates the construction of linear hat functions. It is not necessary to construct a hat function for the second region since it already has the uniform property.

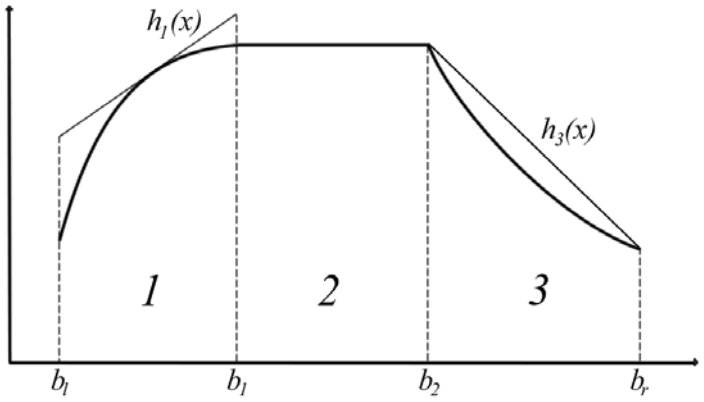


Figure 2.8. Construction of hat functions for parts of Figure 2.6

Require: Quasi-density $f(x)$ on domain (b_l, b_r); partition of domain into N intervals (b_i, b_{i+1}), $b_l = b_0 < b_{i-1} < b_i < b_N = b_r$; hat $h_i(x)$, inverse cdf of hat $H_i^{-1}(x)$, and probabilities w_i prop. to areas below h_i for each interval (b_i, b_{i+1}).
Output: Random variate X with density prop. to f.
1: **loop**
2: Generate discrete random variate J with probability vector (w_i).
/* Use Indexed Search or any other discrete RV generation method */
3: Generate $U \sim U(0,1)$.
4: $X \leftarrow H_J^{-1}(U)$
5: Generate $Y \sim U(0, h_J(X))$.
6: **if** $Y \le f(X)$ **then**
7: **return** X.

Figure 2.9. Composition – Rejection

2.3.2. Decomposition

If squeezes are added to the algorithm in Figure 2.9, this will reduce the number of evaluations of the density f but leaves the expected number of uniform random numbers unchanged. Adding a squeeze s_i in an interval (b_{i-1}, b_i) splits the region below the hat h_i into two parts. It is possible to decompose the density f further by:

$$h_i(x) = s_i(x) + (h_i(x) - s_i(x))$$

It is obvious that any random point (X, Y) generated by the algorithm that falls into the lower part below the squeeze can be accepted immediately without evaluating the density. If X is generated (by inversion) from the distribution with density proportional to the squeeze s_i, then it is not even necessary to generate Y (Hörmann *et al.*, 2004).

Figure 2.10 illustrates the decomposition of the region below the hat with constant squeezes, cs_i, and linear squeezes, ls_i. Since the second region already has the uniform property, there is no need for decomposition.

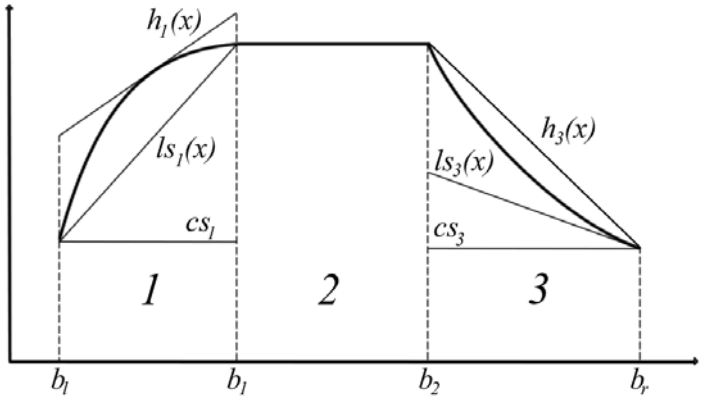


Figure 2.10. Decomposition of the regions below the hat in Figure 2.8

2.4 Rejection with Staircase-Shaped Hat Functions (Ahrens Method)

As the probability of rejection increases, the performance of the algorithm decreases. In order to improve the performance of the algorithm, a monotonically decreasing density can be divided into many subintervals and piecewise constant hats and constant squeezes can be used. Splitting points are denoted by $b_l = b_0 < b_1 < b_2 < \ldots < b_N = b_r$ and subintervals are denoted by $i = 1, 2, \ldots, N$. So the constant hat and constant squeezes can be evaluated by:

$$h_i = f(b_{i-1}) \quad for\ x \in [b_{i-1}, b_i, i = 1, \ldots, N)$$
$$s_i = f(b_i) \quad for\ x \in [b_{i-1}, b_i, i = 1, \ldots, N)$$

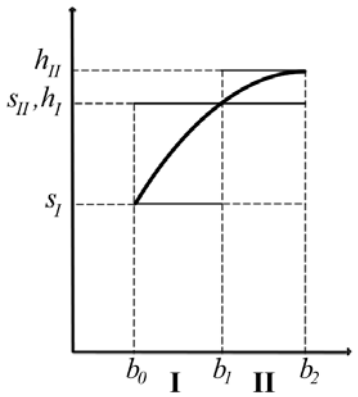


Figure 2.11. Staircase-shaped hat and squeeze functions for an increasing density

To generate a random variate, first, we have to decide on the subinterval. The probabilities of subintervals can easily be based on the areas below their constant hats. And the Indexed Search (Guide Table) Method, which is a generation method for discrete probability distributions, can be used to generate a random subinterval index (See Section 2.5.2).

After the index has been generated with a standard uniform random variate, the uniform variate can be recycled. That allows obtaining another standard uniform random variate to generate X in the chosen subinterval. This is because, if U was used to choose the subinterval I, then U is uniformly distributed over the interval $[F(I-1), F(I))$. Independent of I, we can obtain another standard uniform random variate U' by the following formula:

$$U' = \frac{U - F(I-1)}{F(I) - F(I-1)}$$

This method is called recycling of uniform random numbers in the literature. Since available pseudo-random numbers are not really continuous but have some “resolution” this technique should be used with care because of the lost precision in very short intervals (Hörmann *et al.*, 2004).

After recycling, a uniform random variate can be generated from the subinterval. With basic rejection applied, the algorithm retries until a random variate of the given quasi-density is generated.

Require: Monotonically decreasing quasi-density $f(x)$ on bounded domain $[b_l, b_r]$;
design points $b_l = b_0 < b_1 < b_2 < \ldots < b_N = b_r$.
Output: Random variate X with density proportional to f.
/* Setup */
1: Compute and store all values $f(b_i)$ for $i = 0, \ldots, N$.
2: $A_i \leftarrow f(b_{i-1})(b_i - b_{i-1})$ for $i = 1, \ldots, N$. /* area below hat in (b_i, b_{i-1}) */
3: Set $A \leftarrow \sum_{i=1}^{N} A_i$.
/* Generator */
4: **loop**
5: Generate $V \sim U(0, A)$.
/* Generate J with probability vector proportional to $(A_1, \ldots, A_N)$. */
6: $J \leftarrow \min\{J : A_1 + \ldots + A_j \geq V\}$. /* use Indexed Search algorithm */
7: $V \leftarrow \frac{1}{A_J}(V - (A_1 + \ldots + A_{J-1}))$. /* $V \sim U(0,1)$ recycle uniform RN */
8: $X \leftarrow b_{J-1} + V \cdot (b_J - b_{J-1})$. /* $X \sim U(b_{J-1}, b_J)$ */
9: Generate $Y \leftarrow U(0, f(b_{J-1}))$.
10: **if** $Y \leq f(b_J)$ **then** /* evaluate squeeze */
11: **return** X.
12: **if** $Y \leq f(X)$ **then** /* evaluate density */
13: **return** X.

Figure 2.12. Ahrens basic method

The Ahrens Method can also be applied to an arbitrary distribution with given density provided that the domain can be split into intervals where f behaves monotonically. Such intervals can easily be computed if the extrema of f are known. However, when the domain of the density is unbounded, it must be truncated in a way that the tail regions are computationally not relevant, i.e., the probability of falling into these tail regions must be so small that even in a long running simulation that is not likely to happen (Karawatzki, 2006).

2.5. Discrete Random Variate Generation Methods

2.5.1. The Sequential Search

The sequential-search method is an algorithm to generate discrete random variates with their given probability mass function. It works by searching the smallest value X for

which $F(X)$ is larger than the generated standard uniform variate. X is then the generated discrete random variate. The sequential search starts from $X = 0$.

Require: Probability mass function p_k, $k \geq 0$.
Output: Random variate X with given probability vector.
1: Generate $U \sim U(0,1)$.
2: Set $X \leftarrow 0$, $P \leftarrow p_0$.
3: **while** $U > P$ **do**
4: Set $X \leftarrow X + 1$, $P \leftarrow P + p_X$
5: **return** X.

Figure 2.13. Sequential search

2.5.2. Indexed Search (Guide Table Method)

Sequential search may be slow since the expected number of iterations can become large for long probability vectors. For example, the algorithm may search almost the whole vector until the corresponding random variate is found.

The Indexed Search Method, which has been suggested by Chen and Asau (1974), includes the improvement of starting the while loop from more appropriate points. Instead of starting with $X = 0$ like in the sequential search, we can start with a larger integer value X with $P_X \leq U$. In order to do that, a guide table of size C should be stored to obtain starting points easily. The entries of the guide table are denoted by index i and the each entry g_i is calculated with the following formula:

$$g_i = \inf\{x : P(x) \geq i/C\} \text{ for } i = 0,1,\ldots,C-1.$$

Then, with a standard uniform variate U, $g_{\lfloor UC \rfloor}$ can easily be taken from the guide table and used as the initial value for the search.

Require: Probability vector $(p_0, p_1, \ldots, p_{L-1})$; size C of guide table.
Output: Random variate X with given probability vector.
/* Setup: Computes the guide table g_i for $i = 0,1,\ldots,C-1$. */
1: Compute cumulative probabilities $P_i \leftarrow \sum_{j=0}^{i} p_j$.
2: Set $g_0 \leftarrow 0$, $i \leftarrow 0$.
3: **for** $j = 1$ to $C-1$ **do**
4: **while** $j/C > P_i$ **do**
5: Set $i \leftarrow i+1$.
6: Set $g_j \leftarrow i$.
/* Generator */
7: Generate $U \sim U(0,1)$.
8: Set $X \leftarrow g_{\lfloor UC \rfloor}$.
9: **while** $U > P_X$ **do**
10: Set $X \leftarrow X+1$.
11: **return** X.

Figure 2.14. Indexed search

Looking at the number of iterations, the Indexed Search algorithm is clearly more efficient than the sequential search algorithm. On the other hand, cache-effects in modern computers will reduce the speed-up for really large table sizes. Hörmann *et al.* (2004) recommend using a guide table that is about two times larger than the probability vector to obtain optimal speed. But this number strongly depends on the length of the probability vector, L, the computer, operating system and the compiler used.

Since the table methods for generating random variates from continuous distributions rely on the composition idea, the subintervals are denoted by an integer index. In order to decide about the region where the actual random variate will be generated, a random index must be generated first by using the cumulative probabilities of those regions. The Guide Table method is the most appropriate method to generate integer indices. This is because the sampling executes faster with less memory occupation compared to the other discrete automatic random variate generation methods.

3. TRIANGULAR AHRENS METHOD

The Ahrens Method is a simple method; however it needs a large data table in order to decrease the rejection probability. Otherwise the large rejection probability results in more density function calls and more iterations until an acceptance.

The aim of the Triangular Ahrens Method is to generate random variates with simple arithmetic operations and less calls to density functions. In order to do that, the algorithm divides the density in many subintervals and considers handling those subintervals separately. This is also the main idea of "Rejection with Staircase-Shaped Hat Functions" (Hörmann *et al.*, 2004). We call the method "Triangular Ahrens", since it is an improved version of the Ahrens Method with less subintervals. Thus, firstly, the idea of "Ahrens Method", secondly, the idea of mirroring in point-symmetric domains (Hörmann and Leydold, 2007) should be analyzed clearly, since they will also be used in the new algorithm in order to decrease the rejection probability.

Assuming that the 64-bit double precision provided by modern computers is capable of giving almost exact results for density functions with an insignificant error, Triangular Ahrens Method can generate random variates from exact densities, since it is based on rejection from the original density.

3.1. Linear PDF and the Mirroring Principle

Hörmann and Leydold (2007) proposed a new method for generating random variates form linear densities that are positive on point-symmetric domains. The method basically relies on the rejection concept. However, points which are to be rejected can actually be used to obtain another independent random variate.

Basically, for a linear density that is positive on a bounded domain $[b_l, b_r]$, a constant hat can easily be obtained depending on its slope behavior. Instead, since $[b_l, b_r]$ is a point-symmetric domain with respect to its center $(b_l + b_r)/2$, The Linear PDF method

takes $f((b_l + b_r)/2)$ in the place of the constant hat and compensates the density loss on the region above with the rejection region below, which are symmetric regions. Thus, any rejected point, by reflecting it on the center, can be transformed into an accepted point (See Figure 3.1).

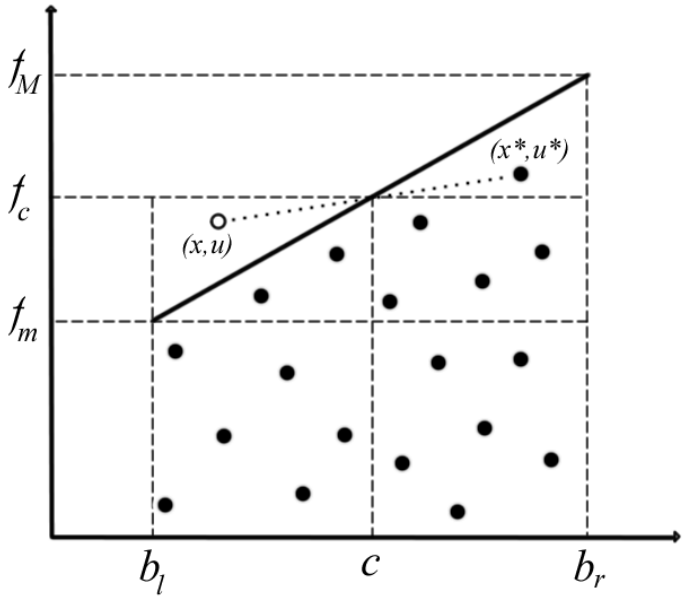


Figure 3.1. The idea of the mirroring principle

Require: Linear density $\ell(x)$ on point-symmetric domain D with center c with $\ell(x) \geq 0$ for $\forall x \in D$.

Output: Random variate X with density ℓ.

/* Setup */

1: Compute $f_c = \ell(c)$.

2: Compute $f_m = \min_{x \in D} \ell(c)$. /* constant squeeze */

/* Generator */

3: Generate X uniformly in D.

4: Generate U uniformly in $[0, f_c]$.

5: **if** $U \leq f_m$ **then** /* below squeeze */

6: **return** X.

7: **else if** $U \leq \ell(x)$ **then** /* below density */

8: **return** X.

9: **else** /* reflect point on the center */

10: **return** $X^* = 2c - X$.

Figure 3.2. Linear PDF

This mirroring principle is also applicable for multidimensional linear densities on point-symmetric domains. Besides, Hörmann and Leydold (2007) used this idea for random variate generation from concave densities over multidimensional domains. The main idea of the algorithm can be used for both convex and concave densities in dimension one. This will be explained together with the new algorithm.

3.2. The Algorithm

As previously mentioned, Rejection from Staircase-Shaped Functions (Ahrens Method) is based on the idea of dividing the density into many subintervals. For each subinterval, local maximum defines a constant hat and the minimum can be used as a constant squeeze. For monotone densities, those values are equal to the maximum and the minimum of the density values at the boundaries. The region between the hat and the squeeze is desired to be small to increase the acceptance probability. The rejection probability can be decreased if subintervals with shorter lengths are selected.

Linear PDF method (Hörmann and Leydold, 2007) is used for linear densities on point symmetric domains. It will help to decrease the rejection probability of the Ahrens Method and result in an efficient generator. In the new algorithm, a hybrid version of these two algorithms is tried to be built. For each subinterval a constant squeeze and a linear hat is defined. Optionally, a linear squeeze can also be introduced.

Since, there are basic rules to calculate the hat and squeezes for convex and concave functions, there should be no subintervals that consist of both a convex and a concave region. In order to provide that, all the inflection points of $f(x)$ must be used as subinterval boundaries. For unimodal distributions, there are often only two inflection points. In order to maintain monotonic behavior over the subintervals, local maxima and minima should also be used as subinterval boundaries. For unimodal distributions, there is only one global maximum, which is the mode, and usually two local minima, which are the cutoff points of the insignificant tails.

For both the concave and the convex case, the constant squeeze can easily be found through the local minimum over the subinterval.

Hörmann and Leydold (2007) propose a lemma to find an appropriate linear hat for concave densities to increase the acceptance probability over multidimensional domains. It is also valid for dimension one.

Lemma 3.1: For the interval $[b_l, b_r]$, let c be its center $(b_l + b_r)/2$ and f a density function that is strictly concave over the domain $[b_l, b_r]$. Then the rejection constant of a rejection algorithm based on the positive linear hat $\ell(x)$ is minimized if we choose the center c as the construction point of ℓ.

Proof. Since the area below the density on domain $[b_l, b_r]$ is constant, the rejection constant can be minimized by minimizing the region below the positive linear hat $\ell(x)$. The region is computed with the formula:

$$\int_{b_l}^{b_r} \ell(x)dx = \frac{\ell(b_l)+\ell(b_r)}{2}(b_r - b_l) = \ell\left(\frac{b_l + b_r}{2}\right)(b_r - b_l) = \ell(c)(b_r - b_l)$$

Due to the definition of the hat;

$$\ell(x) \geq f(x) \text{ for } \forall x \in [b_l, b_r]$$

$\ell(c)$ must be minimized but it must be equal or larger than $f(c)$. Since the domain is strictly concave, it is possible to reach $\ell(c) = f(c)$ with a tangent to the density on the center point c (Hörmann and Leydold, 2007).

Lemma 3.2: For the interval $[b_l, b_r]$, let c be its center $(b_l + b_r)/2$ and f a density function that is strictly convex over the domain $[b_l, b_r]$. Then the probability of immediate acceptance of a rejection algorithm based on the linear squeeze $s(x)$ is maximized if we choose boundaries b_l and b_r as the intersection points of s.

Proof. The region below the positive linear squeeze $s(x)$ must be maximized to maximize the probability of immediate acceptance with linear squeeze. The region is computed with the formula:

$$\int_{b_l}^{b_r} s(x)dx = \frac{s(b_l)+s(b_r)}{2}(b_r - b_l) = \frac{s(b_l)(b_r - b_l)}{2} + \frac{s(b_r)(b_r - b_l)}{2}$$

Due to the restriction of squeeze definition of rejection method;

$$s(x) \leq f(x) \text{ for } \forall x \in [b_l, b_r]$$

$s(b_l)$ and $s(b_r)$ can be maximized as they must be smaller or equal to $f(b_l)$ and $f(b_r)$. Since the domain is strictly concave, by the definition of concavity, the following statement holds.

$$f(\lambda b_l + (1-\lambda)b_r) \geq \lambda\, f(b_l) + (1-\lambda) f(b_r) = \lambda\, s(b_l) + (1-\lambda) s(b_r) \text{ for } \lambda \in [0,1] \text{ and } \forall x \in [b_l, b_r]$$

Then it is possible to maximize the probability of acceptance with a linear squeeze, which intersects with the density on the boundaries b_l and b_r.

So, for the concave case, the linear hat should be a tangent constructed in the center point of the subinterval. The linear squeeze is a secant, intersecting in the boundaries.

For the convex case, it can also be proven in a similar way that the linear hat should be defined like the linear squeeze for the convex case. It may be possible to maximize the immediate acceptance probability with a linear squeeze. However, within the existence of a constant squeeze, this minimization is not important and not very easy. This kind of optimization is a drawback for the desired simplicity and the speed of the automatic random variate generation algorithms. Instead, without the consideration of finding the optimal linear squeeze, the tangent constructed on the center point can be used for simplicity. Thus, the data stored for each subinterval will be the same, independent from being convex or concave:

- Slope of the linear hat function constructed in the center point of the subinterval.
- Intercept of the linear hat function constructed in the center point of the subinterval.
- Slope of the linear squeeze function that is intersecting the density in the boundaries.
- Intercept of the linear squeeze function that is intersecting the density in the boundaries.
- Constant squeeze, which is equal to the minimum of the density over the subinterval.

3.2.1. The Setup Algorithm

To run the sampling algorithm, initially a table must be created and the required constants for each subinterval must be stored in it. This is done in a setup algorithm. Below are the basic required elements of that table.

- A vector that holds the subinterval boundaries.
- Cumulative probability for each subinterval.
- Constant squeeze for each subinterval.
- Slope of the linear squeeze for each subinterval.
- Intercept of the linear squeeze in the center of the subinterval.
- Slope of the linear hat for each subinterval.
- Intercept of the linear hat in the center of the subinterval.
- A guide table to speed up the subinterval selection with the Indexed Search method.

In order to compute the constants of the table, initially the density must be divided into subintervals.

3.2.1.1. Phase I: Flexible Subinterval Creation. The main reason of creating those subintervals is to decrease the rejection probability and increase the acceptance probability. So, for each subinterval, the area of the region between the linear hat and the constant squeeze, a_q, must be less than a critical area, a_c, which is defined by the user.

Secondly, the evaluation of the linear hat will depend on the behavior of the subinterval; whether it is monotonically increasing or monotonically decreasing, convex or concave. Each subinterval must possess one single behavior. Then an easy formula can be introduced to evaluate a_q for subinterval $[b_l, b_r]$ with density function f.

$$a_q = \begin{cases} \dfrac{|f(b_r) - f(b_l)|}{2}(b_r - b_l) & \text{if convex} \\ \left(f\left(\dfrac{b_r + b_l}{2}\right) - f(b_l) \right)(b_r - b_l) & \text{if concave} \wedge \text{increasing} \\ \left(f\left(\dfrac{b_r + b_l}{2}\right) - f(b_r) \right)(b_r - b_l) & \text{if concave} \wedge \text{decreasing} \end{cases} \tag{3.1}$$

Finally, like the "Ahrens Method", our new method will require cut-off points for unbounded domains. Those must be initially selected by the user.

Require: Probability density function $f(x)$, critical area a_c for the region between linear hat and constant squeeze, a sequential set Z of all local extrema, cutoff and inflection points $(z_0, z_1, \ldots, z_m)$ from the smallest to largest, which will also help to explain the behavior of the density.

Output: A vector that holds sequential flexible subinterval boundaries $(b_0, b_1, \ldots, b_N)$, and its length $N+1$.

1: Set $i \leftarrow 0$.

2: Set $b_i \leftarrow z_i$.

3: **while** $b_i \neq z_m$ **do**

4: Compute a_q for subinterval $[b_i, z_{i+1}]$. /* use equation 3.1 */

5: **if** $a_q \leq a_c$ **then** /* use subinterval and proceed */

6: $b_{i+1} \leftarrow z_{i+1}$

7: $i \leftarrow i+1$

8: **else** **/*** divide subinterval ***/**

9: Add $\dfrac{b_i + z_{i+1}}{2}$ to set Z and enumerate the elements.

10: **return** $i+1$

Figure 3.3. Triangular Ahrens setup - flexible subinterval creation

In the end of the algorithm, the number of subintervals is returned together with the vector of subinterval boundaries. These results are used as input in the next phase.

3.2.1.2. Phase II: Data Table Creation. After subinterval boundaries were stored in a vector with a known length of $N+1$, this vector will be taken as an input and the table values will be calculated in the second phase. Phase II also requires the probability density function and its first derivative, and the distribution parameters during the table creation process. Again, a vector that holds all inflection points and local extrema will be needed. Just like in Phase I, since the evaluation of table elements will depend on the behavior of the subinterval, this vector will help to introduce easy formulas for table elements: cs_i (constant squeeze), lhs_i (slope of the linear hat), lhi_i (intercept of the linear hat), lss_i (slope of the linear squeeze), lsi_i (intercept of the linear squeeze) and ca_i (unnormalized cumulative area of the regions below the linear hats) for subinterval $i = 1,...,N$. Below, those functions are given.

$$\text{Constant Squeeze: } cs_i = \begin{cases} f(b_{i-1}) & \textit{if increases} \\ f(b_i) & \textit{if decreases} \end{cases} \tag{3.2}$$

$$\text{Slope of the Linear Hat: } lhs_i = \begin{cases} \dfrac{f(b_i) - f(b_{i-1})}{b_i - b_{i-1}} & \textit{if convex} \\ f'\left(\dfrac{b_i + b_{i-1}}{2}\right) & \textit{if concave} \end{cases} \tag{3.3}$$

$$\text{Intercept of the Linear Hat: } lhi_i = \begin{cases} \dfrac{f(b_i) + f(b_{i-1})}{2} & \textit{if convex} \\ f\left(\dfrac{b_i + b_{i-1}}{2}\right) & \textit{if concave} \end{cases} \tag{3.4}$$

$$\text{Slope of the Linear Squeeze: } lss_i = \begin{cases} f'\left(\dfrac{b_i + b_{i-1}}{2}\right) & \textit{if convex} \\ \dfrac{f(b_i) - f(b_{i-1})}{b_i - b_{i-1}} & \textit{if concave} \end{cases} \tag{3.5}$$

$$\text{Intercept of the Linear Squeeze: } lsi_i = \begin{cases} f\left(\dfrac{b_i + b_{i-1}}{2}\right) & \textit{if convex} \\ \dfrac{f(b_i) + f(b_{i-1})}{2} & \textit{if concave} \end{cases} \tag{3.6}$$

lhi_i (intercept of the linear hat) and lsi_i (intercept of the linear squeeze) are evaluated by using the center point of the subinterval as the origin. This will be helpful in

applying the mirroring principle of Linear PDF algorithm since the lhi_i value represents the y-coordinate of the symmetry center and can be directly taken from the table.

ca_i (unnormalized cumulative area of the regions below the linear hats) will be used in the Indexed Search algorithm to choose a subinterval with a uniform random variate. The cumulative area vector is still useful in its unnormalized form. But since less mathematical operations in the sampling algorithm are preferred, this vector will be normalized in the next step of phase II, by dividing all elements with its last element. After initializing ca_0 to zero, other elements of the unnormalized cumulative area vector can be evaluated with the formula:

$$ca_m = \sum_{i=1}^{m} lhi_i \left(b_i - b_{i-1}\right) = ca_{m-1} + lhi_m \left(b_m - b_{m-1}\right) \text{ for } m = 1, \ldots, N \qquad (3.7)$$

As the last step, a guide table should be created by using the normalized cumulative area vector. In our applications, this guide table will be two times longer than the number of subintervals (Hörmann *et al.*, 2004).

Require: Probability density function $f(x)$ and the corresponding parameters, first derivative of the pdf $f'(x)$, a vector that holds all local extrema, cut-off and inflection points $(z_0, z_1, \ldots, z_M)$ which will explain the behavior of the density, a vector of length $N+1$ that holds sequential subinterval boundaries $(b_0, b_1, \ldots, b_N)$.

Output: A vector that holds cumulative probability vector for subintervals $(P_0, P_1, \ldots, P_N)$, a vector that holds constant squeezes for each subinterval $(cs_1, \ldots, cs_N)$, a vector that holds the slope of linear hats for each subinterval $(lhs_1, \ldots, lhs_N)$, a vector that holds the intercept of linear hats for each subinterval $(lhi_1, \ldots, lhi_N)$, a vector that holds the slope of linear squeezes for each subinterval $(lss_1, \ldots, lss_N)$, a vector that holds the intercept of linear squeezes for each subinterval $(lsi_1, \ldots, lsi_N)$, a guide table of size $2 \times N$ for the Indexed Search for subinterval selection $(g_0, g_1, \ldots, g_{2\times N-1})$.

1: Store b_i for $i = 0, \ldots, N$.

2: Set $i \leftarrow 1$, $ca_0 \leftarrow 0$.

3: **while** $i \le N$ **do**

4: Compute and store cs_i, lhs_i, lhi_i, lss_i, lsi_i.

/* use equation (3.2, 3.2, 3.4, 3.5, 3.6) */

5: Compute ca_i. /* use equation (3.7) */

6: Set $P_i \leftarrow \dfrac{ca_i}{ca_N}$ for $i = 0, \ldots, N$. /* normalization */

7: Set $g_0 \leftarrow 1$, $i \leftarrow 1$.

8: **for** $j = 1$ **to** $2N-1$ **do** /* guide table creation for Indexed Search */

9: **while** $j/2N > P_i$ **do**

10: Set $i \leftarrow i+1$.

11: Set $g_j \leftarrow i$.

Figure 3.4. Triangular Ahrens setup - data table creation

3.2.2. The Sampling Algorithm

While the required constants are stored and ready to be used as the input, finally random variates of the corresponding density can be generated. Unlike the setup, the

sampling algorithm does not depend on the behavior of the density and just executes using the corresponding data called from the setup table.

The sampling algorithm initially must decide about which subinterval to consider. Therefore, by using the guide table $(g_0, g_1, \ldots, g_{2\times N-1})$ and cumulative probabilities of subintervals $(P_0, P_1, \ldots, P_N)$, the Indexed Search algorithm is run to generate index i. Then, another standard uniform variate should be recycled from the one which is used in the indexed search algorithm. By rescaling this uniform variate over the subinterval, a uniform random variate X of the corresponding subinterval can be generated.

Like in the rejection algorithm, another uniform random variate Y should be generated, which is uniformly distributed between zero and the value of the linear hat function on the center point, which is equal to the intercept of the linear hats for all subintervals, lhi_i.

By using the mirroring principle (Hörmann and Leydold 2007), the pair of random variates can be mirrored at the reflection point (c_i, lhc_i), if the point they represent is above the linear hat (See Section 3.1). But this computation is not necessary if the pair (X,Y) is below the constant squeeze, cs_i, and can be immediately accepted.

Actually there are three cases after generating the pair (X,Y).

- (X,Y) is below the constant squeeze and is immediately accepted.
- (X,Y) is above the constant squeeze but below the linear hat.
- (X,Y) is above the linear hat.

In the third case, the mirroring principle is applied. Then, the new uniform pair (X',Y') will obtain an equivalent status as the pair (X,Y) has in the second case.

Continuing from the second and the third case, the pair (X,Y) should be considered under three new cases.

- (X,Y) is below the linear squeeze and is accepted through squeeze comparison.
- (X,Y) is above the linear squeeze and below the density and is accepted through density comparison.
- (X,Y) is above the density, so it is rejected and the generation process starts again.

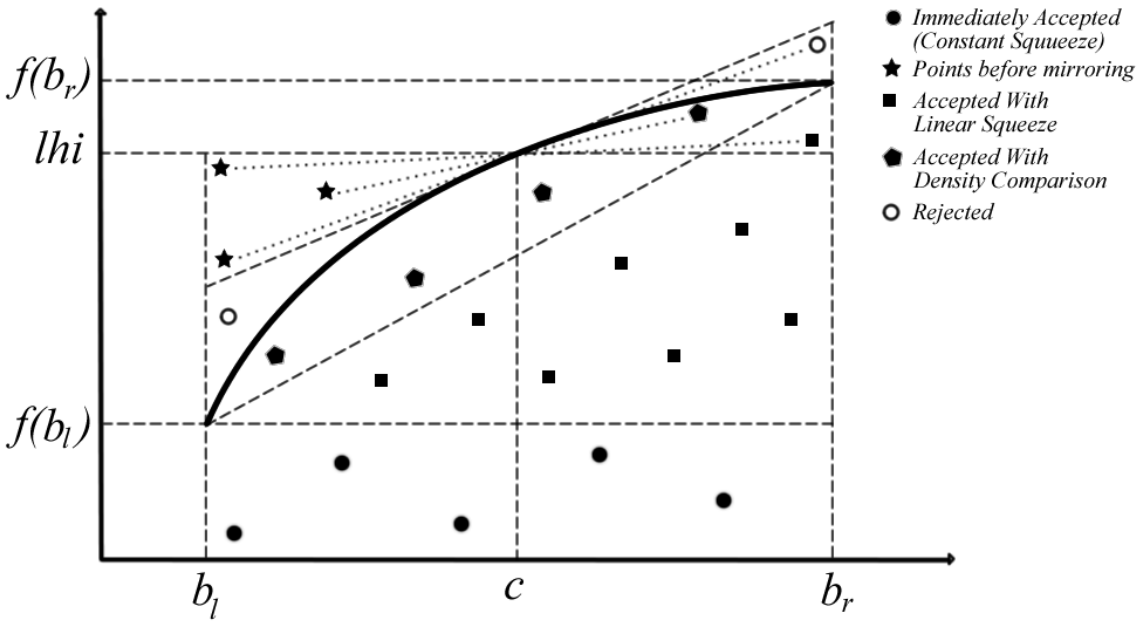


Figure 3.5. Demonstration of different cases in generation algorithm of the Triangular Ahrens Method for concave subintervals

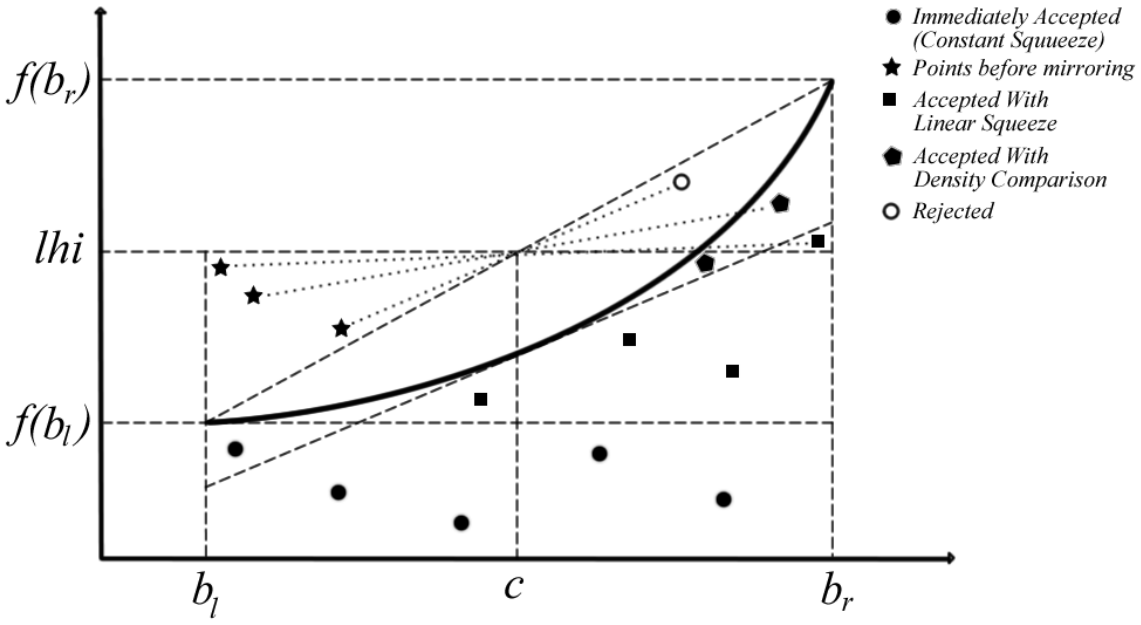


Figure 3.6. Demonstration of different cases in generation algorithm of the Triangular Ahrens Method for convex subintervals

The sampling algorithm terminates if a random variate is obtained. Otherwise, the algorithm rejects the variate and repeats from the beginning. If the rejection probability is large, this increases the execution time of the whole algorithm when a large number of random variates is required.

Require: Probability density function $f(x)$, a data table of the Triangular Ahrens Method with the corresponding density.

Output: Random variate X with density f.

1: **loop**
2: Generate $U \sim U(0,1)$.
3: Set $i \leftarrow g_{\lfloor UC \rfloor}$.
4: **while** $U > P_i$ **do** /* Indexed Search */
5: Set $i \leftarrow i+1$.
6: $U \leftarrow (U - P_{i-1})/(P_i - P_{i-1})$ /* recycling */
7: Set $X \leftarrow b_{i-1} + U(b_i - b_{i-1})$.
8: Generate $Y \sim U(0, lhi_i)$.
9: Set $c \leftarrow (b_i + b_{i-1})/2$.
10: **If** $Y \le cs_i$ **do** /* immediately acceptance */
11: **return** X.
12: **else**
13: **If** $Y > lhi_i + lhs_i(X - c)$ **do** /* mirroring */
14: Set $X \leftarrow 2c - X$, $Y \leftarrow 2lhi_i - Y$.
15: **If** $Y \le lsi_i + lss_i(X - c)$ **do** /* through linear squeeze */
16: **return** X.
17: **else if** $Y \le f(X)$ **do** /* through density */
18: **return** X.

Figure 3.7. Triangular Ahrens sampling algorithm

3.3. Computational Results and Performance Characteristics

We coded the Triangular Ahrens algorithm in "C" to evaluate some performance characteristics and measure the speed of the algorithm. Therefore, we applied the algorithm to generate random variates from particular distributions which are shown in Table 3.1.

Table 3.1. List of unimodal distributions used in the Triangular Ahrens and the Polynomial Density Inversion algorithms

Symbol	Distribution	Parameters	Left Cutoff	Left Inflection	Mode	Right Cutoff	Right Inflection
N	Normal	(0,1)	-6	-1	0	1	6
C	Cauchy	(0,1)	-640000	-0.577350	0	0.577350	640000
E	Exponential	(1)	0	0	0	0	17
G	Gamma	(3,1)	0	0.585786	2	3.414213	21
B1	Beta	(3,4)	0	0.155051	0.4	0.644948	1
B2	Beta	(30,40)	0.15	0.366049	0.426470	0.486891	0.725

3.3.1. Timing Results

The generation speed is an important indicator for the performance of the random variate generation algorithms (Hörmann *et al.*, 2004). In practice, it is interesting to see the total generation time for setup and all generated random variates. Yet, there is a great variability of those results depending on the computing environment, the compiler and the uniform pseudo-random number generator. To decrease this variability, Hörmann *et al.* (2004) suggested a term, relative generation time of an algorithm as the generation time divided by the generation time for the exponential distribution using inversion which is done by the formula:

$$X = -\log(1-U)$$

where U is a standard uniform random variate. This time has to be taken in exactly the same programming environment, using the same type of function call, etc. The relative generation time is still influenced by many factors and we should not consider differences of less than 25 per cent. Nevertheless, it can give us a crude idea about the speed of a certain random variate generation method (Hörmann *et al.*, 2007).

For the Triangular Ahrens algorithm, the relative average generation time of the whole algorithm, which includes both the setup and the sampling algorithms, is measured while 10^4 and 10^6 random variates are generated. The single subinterval creation parameter, critical area, a_c, was set to different values to see the performance of the algorithm. In order to evaluate the average generation times, each generation was repeated 1000 times in an outer loop, and then the whole execution time was divided over 1000.

Lemma 3.3: The total execution time of the Triangular Ahrens algorithm, T, for sampling a large number of random variates follows the normal distribution.

Proof. Assuming that the execution time of the setup algorithm, T_S, and the single iteration of the sampling algorithm, T_I, are constant, the variance of the total execution time is

caused by the number of iterations until a successful iteration, I, which follows the geometric distribution with a probability of success equal to $1/\alpha$ (See Theorem 2.3).

Let I_i represent the number of unsuccessful iterations for the i^{th} random variate, where $i = 1, \ldots, n$. According to Hurley and Andrews (2007), if n is sufficiently large, then $\sum_{i=1}^{n} I_i$ is approximated by a normal distribution with mean:

$$E\left(\sum_{i=1}^{n} I_i\right) = E\left(I_1 + I_2 + \ldots + I_n\right) = \frac{n}{\alpha}$$

and the variance:

$$Var\left(\sum_{i=1}^{n} I_i\right) = Var\left(I_1 + I_2 + \ldots + I_n\right) = n.Var\left(I\right) = \frac{n\left(1-\alpha\right)}{\alpha^2}$$

Thus, the total execution time of the Triangular Ahrens algorithm is normally distributed with mean:

$$E\left(T\right) = \left(\frac{n}{\alpha} + n\right)T_I + T_S$$

and the variance:

$$Var\left(T\right) = \frac{n\left(1-\alpha\right)}{\alpha^2}$$

if M is sufficiently large.

Table 3.2. Relative average generation times for the Triangular Ahrens Method with sample size $n = 10^4$

	Critical Area					
	0.05	**0.01**	**0.005**	**0.001**	**0.0005**	**0.0001**
N	1.21	1.07	1.06	1.10	1.13	1.37
C	1.21	1.15	1.10	1.16	1.16	1.46
E	1.16	1.04	1.03	1.07	1.07	1.28
G	1.70	1.30	1.30	1.55	1.82	2.81
B1	1.97	1.51	1.55	2.06	2.48	4.24
B2	2.22	1.72	1.72	2.27	2.58	4.88

Table 3.3. Relative average generation times for the Triangular Ahrens Method with sample size $n = 10^6$

	Critical Area					
	0.05	**0.01**	**0.005**	**0.001**	**0.0005**	**0.0001**
N	1.12	0.99	0.94	0.89	0.88	0.89
C	1.13	1.00	0.93	0.93	0.90	0.88
E	1.09	0.94	0.94	0.90	0.88	0.88
G	1.55	1.06	1.00	0.91	0.91	0.89
B1	1.78	1.10	1.03	0.91	0.89	0.91
B2	1.99	1.27	1.07	0.94	0.92	0.93

We can obtain the average total execution time of 10^4 and 10^6 variates in our test computer by multiplying relative average generation times respectively with 6.7×10^{-4} and 6.7×10^{-2}, which are the actual generation times in seconds, for the same number of standard exponential random variates.

3.3.2. Memory Occupation

The required table size can be evaluated if the number of subintervals, N, is known. For each subinterval we are holding the constant squeeze, the slope and the intercept of the linear squeeze, the slope and the intercept of the linear hat, the left boundary and the cumulative probability of the subinterval. Each of those values can be stored as a double type variable, which occupies 8 bytes of memory.

A guide table of size $2 \times N$ is also stored for the Indexed Search. Since the indices are integers, each element of the guide table can be stored as an integer type variable, which occupies 4 bytes of memory.

Insignificantly, cumulative probabilities and the subinterval boundaries need an additional storage since they refer to boundaries instead of subintervals. A single integer variable can be used to store the total number of subintervals for quick calls. Therefore, N subintervals will result in a table size of:

$$TS = (7N+2)b_d + (2N+1)b_i = 8\times(7N+2) + 4\times(2N+1) = 64N + 20$$

which is evaluated in bytes. Table 3.4 below shows the number of subintervals and the total size of the data tables for each distribution with different critical area parameters.

Table 3.4. Number of subintervals and the total size of the data tables for different parameters and distributions

	Critical Area											
	0,05		**0,01**		**0,005**		**0,001**		**0,0005**		**0,0001**	
N	12	788	24	1556	34	2196	76	4884	104	6676	238	15252
C	44	2836	58	3732	76	4884	134	8596	180	11540	382	24468
E	13	852	25	1620	32	2068	68	4372	94	6036	210	13460
G	11	724	25	1620	30	1940	68	4372	104	6676	220	14100
B1	10	660	22	1428	29	1876	64	4116	89	5716	187	11988
B2	13	852	25	1620	35	2260	75	4820	98	6292	232	14868

3.3.3. Performance Characteristics

Performance characteristics are important measures for rejection based algorithms. For the Triangular Ahrens Method, random variates can be accepted through constant squeeze immediately or through linear squeeze or density comparison with or without mirroring applied. To understand the performance of the algorithm, it is important to have an idea about how many times these different functions are called.

In the flexible subinterval creation phase of the setup, the critical area parameter, a_c, plays an important role to select performance characteristics. Assume that the Triangular Ahrens Method will be used to generate random variates from a density f. The critical area parameter leads to create N subintervals with a vector of boundaries $(b_0, b_1, \ldots, b_N)$. Then the total region below all the hats and the constant and the linear squeezes can be calculated with the formula:

$$A_h = \sum_{i=1}^{N} A_{h\,i} = \sum_{i=1}^{N} (b_i - b_{i-1}) \max\left\{ \frac{f(b_i) + f(b_{i-1})}{2}, f\left(\frac{b_i + b_{i-1}}{2} \right) \right\}$$

$$A_{cs} = \sum_{i=1}^{N} A_{cs\,i} = \sum_{i=1}^{N} (b_i - b_{i-1}) \min\{ f(b_i), f(b_{i-1}) \}$$

$$A_{ls} = \sum_{i=1}^{N} A_{ls\,i} = \sum_{i=1}^{N} (b_i - b_{i-1}) \min\left\{ \frac{f(b_i) + f(b_{i-1})}{2}, f\left(\frac{b_i + b_{i-1}}{2} \right) \right\}$$

Since, f is assumed to be a density, A_h will converge to one from the right as N gets larger. Then, the upper bound for the probability of immediate acceptance, $P(A_I)$, can be controlled by the critical area parameter, a_c, using the following relationship:

$$P(A_I) = \frac{A_{cs}}{A_h} \geq \frac{A_h - a_c N}{A_h} \geq 1 - a_c N$$

$1 - a_c N$ value can be accepted as a lower bound for the acceptance probability, and $1/(1 - a_c N)$ value can be accepted as an upper bound for the rejection constant. Then the expectation of the number of iterations until an acceptance, $E(I)$, has an upper bound:

$$\alpha = E(I) \leq \frac{1}{1 - a_c N} \tag{3.8}$$

Therefore, the expected number of uniform variates needed for a single generation and the expected number of density evaluations have also upper bounds in a similar way:

$$E(\#U) \leq \frac{2}{1 - a_c N} \tag{3.9}$$

$$E(\# f) \leq \frac{a_c N}{1 - a_c N} \tag{3.10}$$

Exceptionally for the Triangular Ahrens Method, a performance characteristic can be the expected number of the mirroring function calls. Exactly, for 25 percent of the pairs in the region below the hat and above the constant squeeze, the mirroring principle is

applied. Then an upper bound can also be defined for the number of mirroring function calls:

$$E(\#M) \leq \frac{a_c N}{4} \tag{3.11}$$

In order to compute those performance criteria, the Triangular Ahrens algorithm was run 500 times. In each run, the setup algorithm was executed and 10^6 random variates were generated. For each of the performance criteria, the mean and the standard deviation were evaluated.

Table 3.5. The mean and the standard deviation of the percentage of acceptance types and rejection and the calls for mirroring for different distributions and critical area parameters, $a_c = \{0.05, 0.01, 0.005\}$

			N		C		E		G		B1		B2	
Critical Area	**0,05**	**Cons. Squ. (%)**	74.30	0.043	67.23	0.044	77.46	0.042	75.00	0.042	78.32	0.039	75.70	0.041
		Lin. Squ. (%)	20.22	0.040	23.33	0.042	17.70	0.040	18.96	0.039	17.48	0.036	19.05	0.037
		Density (%)	2.10	0.014	3.34	0.018	1.35	0.012	2.17	0.014	1.98	0.014	2.05	0.014
		Rejection (%)	3.38	0.017	6.10	0.023	3.49	0.018	3.87	0.018	2.22	0.015	3.20	0.016
		Mirroring (%)	6.11	0.023	7.92	0.024	5.64	0.023	5.92	0.023	4.91	0.022	5.24	0.022
	0,01	**Cons. Squ. (%)**	88.24	0.031	82.24	0.037	90.19	0.027	89.84	0.030	90.62	0.027	88.68	0.030
		Lin. Squ. (%)	10.19	0.030	14.40	0.035	8.98	0.026	8.93	0.028	8.37	0.026	9.45	0.028
		Density (%)	0.59	0.007	1.15	0.011	0.25	0.005	0.48	0.007	0.48	0.007	0.66	0.008
		Rejection (%)	0.98	0.010	2.21	0.014	0.58	0.008	0.75	0.008	0.53	0.007	1.21	0.011
		Mirroring (%)	2.85	0.017	4.35	0.019	2.45	0.016	2.45	0.015	2.22	0.014	2.74	0.016
	0,005	**Cons. Squ. (%)**	92.10	0.026	88.57	0.030	92.64	0.025	91.80	0.025	93.01	0.023	92.24	0.025
		Lin. Squ. (%)	6.95	0.025	9.64	0.028	6.73	0.024	7.26	0.024	6.32	0.023	6.88	0.024
		Density (%)	0.37	0.006	0.57	0.007	0.18	0.004	0.36	0.006	0.32	0.006	0.33	0.006
		Rejection (%)	0.58	0.008	1.22	0.012	0.45	0.007	0.58	0.008	0.35	0.006	0.55	0.007
		Mirroring (%)	1.90	0.014	2.83	0.016	1.89	0.013	1.98	0.013	1.66	0.012	1.89	0.013

Table 3.6. The mean and the standard deviation of the percentage of acceptance types and rejection and the calls for mirroring for different distributions and critical area parameters, $a_c = \{0.001, 0.0005, 0.0001\}$

			N		C		E		G		B1		B2	
Critical Area	0,001	Cons. Squ. (%)	96.57	0.017	94.53	0.021	96.76	0.017	96.45	0.018	96.92	0.017	96.58	0.017
		Lin. Squ. (%)	3.20	0.017	4.94	0.020	3.05	0.016	3.33	0.018	2.94	0.016	3.18	0.016
		Density (%)	0.09	0.003	0.18	0.004	0.05	0.002	0.09	0.003	0.07	0.003	0.08	0.003
		Rejection (%)	0.14	0.004	0.34	0.006	0.14	0.004	0.13	0.004	0.07	0.003	0.16	0.004
		Mirroring (%)	0.84	0.009	1.35	0.011	0.81	0.009	0.87	0.009	0.75	0.008	0.84	0.009
	0,0005	Cons. Squ. (%)	97.47	0.015	96.04	0.018	97.68	0.015	97.70	0.015	97.82	0.014	97.39	0.016
		Lin. Squ. (%)	2.41	0.015	3.66	0.018	2.24	0.015	2.20	0.014	2.09	0.014	2.46	0.015
		Density (%)	0.05	0.002	0.10	0.003	0.02	0.002	0.04	0.002	0.05	0.002	0.06	0.002
		Rejection (%)	0.07	0.003	0.20	0.005	0.06	0.002	0.06	0.002	0.04	0.002	0.09	0.003
		Mirroring (%)	0.62	0.008	0.984	0.010	0.58	0.008	0.57	0.007	0.53	0.007	0.64	0.008
	0,0001	Cons. Squ. (%)	98.93	0.010	98.24	0.013	98.99	0.010	98.94	0.010	98.97	0.010	98.93	0.010
		Lin. Squ. (%)	1.04	0.010	1.68	0.012	1.00	0.010	1.03	0.009	1.01	0.010	1.04	0.010
		Density (%)	0.01	0.001	0.03	0.002	0.00	0.001	0.01	0.001	0.01	0.001	0.01	0.001
		Rejection (%)	0.02	0.001	0.05	0.002	0.01	0.001	0.02	0.001	0.01	0.001	0.02	0.001
		Mirroring (%)	0.27	0.005	0.44	0.006	0.25	0.005	0.26	0.006	0.25	0.005	0.27	0.005

By using the data shown in Table 3.5 and 3.6, we can evaluate the performance characteristics with the following formula.

$$\alpha = E(I) = \frac{100}{Cons.Squ.(\%) + Lin.Squ.(\%) + Density(\%)} \tag{3.12}$$

$$E(\#U) = \frac{200}{Cons.Squ.(\%) + Lin.Squ.(\%) + Density(\%)} \tag{3.13}$$

$$E(\#f) = \frac{100 - (Cons.Squ.(\%) + Lin.Squ.(\%))}{Cons.Squ.(\%) + Lin.Squ.(\%) + Density(\%)} \tag{3.14}$$

$$E(\#M) = \frac{Mirroring(\%)}{Cons.Squ.(\%) + Lin.Squ.(\%) + Density(\%)} \tag{3.15}$$

By using the data in Table 3.5 and 3.6 and the Equations 3.12, 3.13, 3.14 and 3.15, the performance characteristics were evaluated and shown in Table 3.7 and 3.8. Upper bounds are evaluated by using Equations 3.8, 3.9, 3.10 and 3.11 and also shown in those tables.

Table 3.7. Actual values and upper bounds of performance characteristics of the Triangular Ahrens Method for different distributions and critical area parameters,

$$a_c = \{0.05, 0.01, 0.005\}$$

			N		C		E		G		B1		B2	
Critical Area	0,05	E(I)	1.035	2.500	1.065	INF	1.036	2.857	1.040	2.222	1.023	2.000	1.033	2.857
		E(#U)	2.070	5.000	2.130	INF	2.072	5.714	2.080	4.444	2.045	4.000	2.066	5.714
		E(#f)	0.057	1.500	0.100	INF	0.050	1.857	0.063	1.222	0.043	1.000	0.054	1.857
		E(#M)	0.063	0.150	0.084	0.55	0.058	0.162	0.062	0.138	0.050	0.125	0.059	0.163
	0,01	E(I)	1.010	1.316	1.023	2.381	1.006	1.333	1.008	1.333	1.005	1.282	1.012	1.333
		E(#U)	2.020	2.632	2.045	4.762	2.012	2.667	2.015	2.667	2.011	2.564	2.024	2.667
		E(#f)	0.016	0.316	0.034	1.381	0.008	0.333	0.012	0.333	0.010	0.282	0.019	0.333
		E(#M)	0.029	0.060	0.044	0.145	0.025	0.063	0.025	0.063	0.022	0.055	0.028	0.063
	0,005	E(I)	1.006	1.205	1.012	1.613	1.004	1.190	1.006	1.176	1.003	1.170	1.006	1.212
		E(#U)	2.012	2.410	2.025	3.226	2.009	2.381	2.012	2.353	2.007	2.339	2.011	2.424
		E(#f)	0.001	0.205	0.018	0.613	0.006	0.190	0.009	0.176	0.007	0.170	0.009	0.212
		E(#M)	0.019	0.043	0.029	0.095	0.018	0.040	0.020	0.038	0.017	0.036	0.019	0.044

Table 3.8. Actual values and upper bounds of performance characteristics of Triangular Ahrens Method for different distributions and critical area parameters,

$$a_c = \{0.001, 0.0005, 0.0001\}$$

			N		C		E		G		B1		B2	
Critical Area	0,001	E(I)	1.001	1.082	1.003	1.155	1.001	1.073	1.001	1.073	1.000	1.068	1.002	1.081
		E(#U)	2.003	2.165	2.007	2.309	2.003	2.146	2.003	2.146	2.001	2.137	2.003	2.162
		E(#f)	0.002	0.082	0.005	0.155	0.002	0.073	0.002	0.073	0.001	0.068	0.002	0.081
		E(#M)	0.008	0.019	0.014	0.034	0.008	0.017	0.009	0.017	0.008	0.016	0.008	0.019
	0,0005	E(I)	1.001	1.055	1.002	1.099	1.001	1.049	1.000	1.055	1.000	1.047	1.001	1.052
		E(#U)	2.001	2.110	2.004	2.198	2.001	2.099	2.001	2.110	2.001	2.093	2.002	2.103
		E(#f)	0.001	0.055	0.003	0.099	0.001	0.049	0.001	0.055	0.001	0.047	0.001	0.052
		E(#M)	0.006	0.013	0.010	0.023	0.006	0.012	0.006	0.013	0.005	0.011	0.006	0.012
	0,0001	E(I)	1.000	1.024	1.001	1.040	1.000	1.021	1.000	1.022	1.000	1.019	1.000	1.024
		E(#U)	2.000	2.049	2.001	2.079	2.000	2.043	2.000	2.045	2.000	2.038	2.000	2.048
		E(#f)	0.000	0.024	0.001	0.040	0.000	0.021	0.000	0.022	0.000	0.019	0.000	0.024
		E(#M)	0.003	0.006	0.004	0.010	0.003	0.005	0.003	0.006	0.003	0.005	0.003	0.006

The main advantage of the Triangular Ahrens Method is the reduction of the rejection constant by using the mirroring principle. For the Ahrens Method, such rejection constants as in Table 3.7 and 3.8 can only be obtained with much more subintervals, therefore with larger tables. We have tried to build an algorithm, which is based on the Ahrens Method and as fast, yet has the advantage of storing less data, requiring less number of iterations until an acceptance and less number of density function calls.

The disadvantage is caused by the setup algorithm, which is more sophisticated than the setup of the Ahrens Method. However, this difference is not important when generating a large number of random variates.

4. APPROXIMATE RANDOM VARIATE GENERATION

Numerical results of the previous chapter (Table 3.5 and 3.6) show that the Triangular Ahrens Method can generate random variates with a small number of rejections. The Ahrens Method also has this property.

If the number of subintervals is sufficiently large, this small proportion of rejected variables should not spoil the distribution that the accepted sample follows. This idea implies the question if rejection is really necessary.

With the rejection removed, the Ahrens method becomes an approximate random variate generation method which approximates the cdf linearly. In other words, the density is approximated with piecewise constant (uniform) distributions. The Triangular Ahrens Method, without any rejection, generates from the density using piecewise linear approximations.

4.1. Piecewise Linear Approximation of the CDF

When the setup algorithm runs for the piecewise constant approximation, it only needs to store subinterval boundaries and the cumulative probabilities of subintervals. If a monotone subinterval $[b_l, b_r]$ is assumed to be uniform, it is clear that the maximum absolute approximation error can be minimized to $|f(b_r) - f(b_l)|/2$ when $(f(b_r) + f(b_l))/2$ is used as constant for that subinterval.

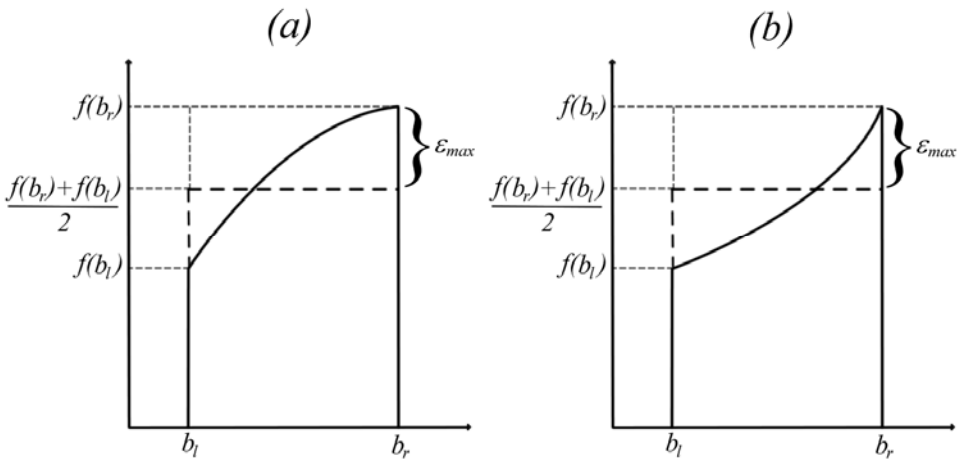


Figure 4.1. Uniform assumption (piecewise constant approximation) of subintervals for (a) the concave case and (b) the convex case

For a set of subintervals, the region below the piecewise constants can be assumed as the weights of subintervals. Therefore, the cumulative probabilities of the subintervals can easily be computed with normalization and summation. Since the only distribution in a single subinterval will be the uniform distribution, no additional data is needed to be stored other than the subinterval boundaries and the cumulative probability vector. In addition, a guide table must be stored for the Indexed Search, which is two times larger than the number of subintervals. Insignificantly, 3 more cells are required for the subinterval vector, the cumulative probability vector of the subintervals and to store the number of subintervals for quick calls. Total data to be stored in the table, then, can be computed in bytes by the following formula:

$$TS = (2N+2)b_d + (2N+1)b_i = 8\times(2N+2) + 4\times(2N+1) = 24N + 20$$

where N represents the number of subintervals and b_d and b_i respectively represent the occupied memory for double and integer variables in bytes.

4.2. Piecewise Linear Approximation of the Density

The piecewise linear approximation makes linear approximation at the subinterval borders. Thus, we use secants in each subinterval to obtain a continuous density of piecewise linear. This is also necessary prevent negative approximated densities over the tails (See Figure 4.3).

However we could do a better linear approximation by using Chebyshev nodes as interpolation points instead of using subinterval borders (See Section 5.1). But the main reason of the linear approximation is to create a simple and fast algorithm. Using Chebyshev nodes spoils the simplicity and the continuous property of the density (See Figure 4.2). It also causes to have negative approximated densities over the tails (See Figure 4.3).

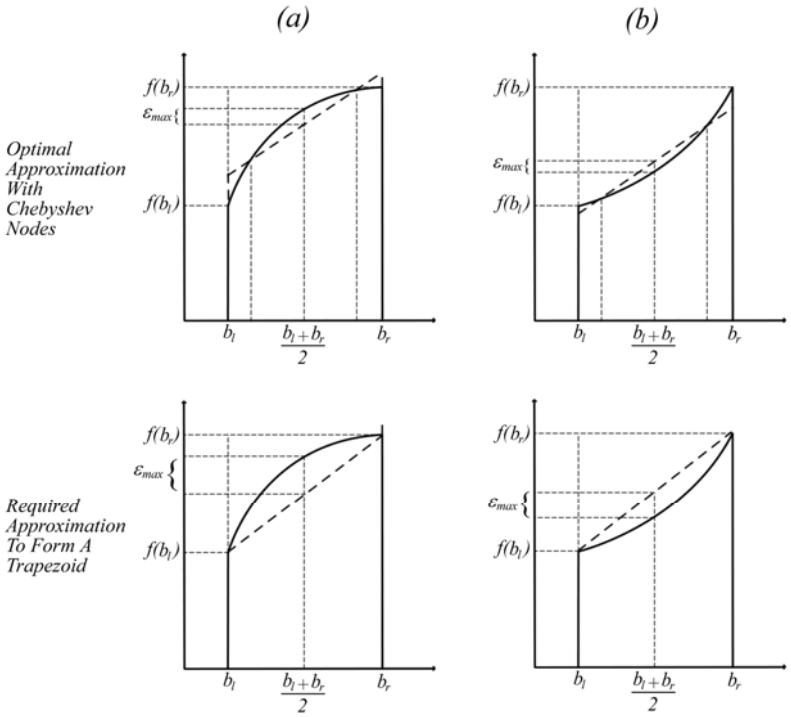


Figure 4.2. (a) The optimal linear approximation over Chebyshev nodes with scale $[0.1464466, 0.8535534]$ (b) Required approximation over the boundaries to prevent from ruining the trapezoid

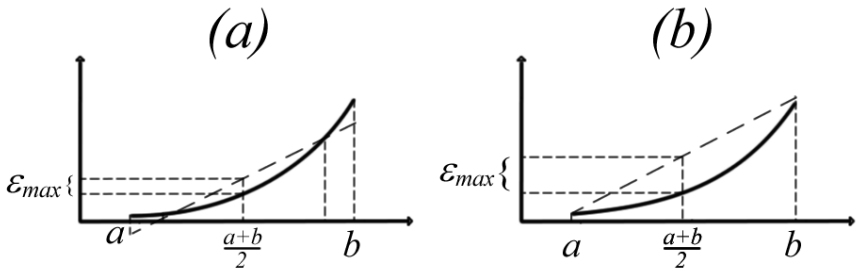


Figure 4.3. (a) Negative density is obtained over a part of the domain by using Chebyshev nodes without scaling (b) Nonnegative density is guaranteed with the secant

Such negative densities in Figure 4.3 ruin rectangular region in the subinterval. Since we apply the inversion to generate random variates, the most important computational difficulty comes as a result of having negative values over the cdf. Those conditions require additional data on the setup table, which will slow down the setup algorithm and consume from the memory. On the other hand, using the secant as the approximated density will increase the number of subintervals, since the maximum absolute approximation error is higher than the maximum absolute approximation error obtained with Chebyshev nodes in the same subinterval. Yet, both the setup and the sampling algorithm will be executed with simpler operations.

When the memory occupation is considered, the difference between the piecewise constant and the piecewise uniform approximations is the additional probability vector which holds the probabilities of the rectangular regions and an indicator vector which defines the type of the triangular region. The total size of the table then can be computed by the following formula:

$$TS = (3N+2)b_d + (3N+1)b_i = 8\times(3N+2) + 4\times(3N+1) = 36N + 20$$

where N represents the number of subintervals and b_d and b_i respectively represent the occupied memory for double and integer variables in bytes.

For both the piecewise constant and the piecewise linear approximations, flexible subintervals can be created with only one parameter. A subinterval can simply be accepted if it has a smaller maximum absolute approximation error than the critical absolute approximation error. Otherwise, it should be divided into two subintervals, both of which should be checked by the same method.

4.3. Memory Occupation and Approximation Performance

We have coded functions in R that evaluates the total table size that is used and the mean and the standard deviation for L_1 error for piecewise constant and linear approximations. These codes are shown in Appendix C.

Table 4.1 shows the number of subintervals and the total size of the tables of the piecewise constant and the piecewise linear approximations for different parameters and distributions. If the number of subintervals are higher than 10000 for a parameter value, it was assumed unnecessary to analyze the corresponding data table.

Table 4.1. Number of subintervals and the total size of tables of the piecewise constant and linear approximations for different parameters and distributions

Degree of Approx.	**Piecewise Constant Approximation**						**Piecewise Linear Approximation**							
Critical Abs. Err.	**1.E-02**		**1.E-03**		**1.E-04**		**1.E-04**		**1.E-05**		**1.E-06**		**1.E-07**	
N	58	1412	568	13652	6012	144308	140	5060	392	14132	1258	45308	4288	154388
C	80	1940	454	10916	4844	116276	170	6140	464	16724	1502	54092	4478	161228
E	76	1844	708	17012	7354	176516	106	3836	320	11540	1040	37460	3247	116912
G	40	980	401	9644	3949	94796	105	3800	331	11936	936	33716	3263	117488
B1	292	7028	2962	71108	-	-	224	8084	815	29360	2721	97976	7104	255764
B2	963	23132	8900	213620	-	-	496	17876	1719	61904	5216	187796	-	-

Table 4.2 shows L_1 error of the piecewise constant and the piecewise linear approximations for different parameters and distributions. Those values were obtained with simulation and the importance sampling with random variates of the original density and of the uniform density between cutoff points (See Section 6.1.3).

Table 4.2. Simulation results for evaluating L_1 error for the piecewise constant and linear approximations with critical absolute error, $\varepsilon_c = 10^{-4}$ (10^{-3} for B1 and B2 distributions) and 1st order approximation with critical absolute error, $\varepsilon_c = 10^{-7}$ (10^{-6} for B2 distribution)

	Piecewise Constant Approximation-1.E-04				**Piecewise Linear Approximation-1.E-07**			
	Original IS Dens.		**Uniform IS Dens.**		**Original IS Dens.**		**Uniform IS Dens.**	
	Mean L_1	**Std. D. L_1**	**Mean L_1**	**Std. D. L_1**	**Mean L_1**	**Std. D. L_1**	**Mean L_1**	**Std. D. L_1**
N	3.28E-04	5.08E-03	3.33E-04	2.86E-04	2.74E-07	2.21E-06	4.23E-07	2.70E-07
C	8.43E-03	6.77E-02	2.38E-03	1.04E-01	1.85E-04	6.07E-03	2.35E-04	3.01E-03
E	5.02E-04	1.60E-02	4.40E-04	4.42E-04	3.40E-07	3.34E-06	6.61E-07	3.98E-07
G	6.59E-04	1.45E-02	5.42E-04	4.73E-04	1.13E-06	5.55E-05	7.55E-07	4.69E-07
B1	3.76E-04	2.57E-03	3.61E-04	2.26E-04	4.27E-08	1.69E-07	4.25E-08	2.40E-08
B2	1.73E-04	1.55E-03	2.38E-04	1.48E-04	2.37E-07	6.56E-06	2.06E-07	1.31E-07

According to Table 4.1 and 4.2, the performance of the piecewise constant approximation is very bad even with the table sizes of more than 100 kilobytes. The piecewise linear approximation is better and can achieve a reasonable approximation error with a magnitude of 10^{-7} but only with a large number of subintervals.

Even though the generated random variates follow the desired distribution for both of the methods, those methods can be improved with higher order approximations and subinterval decomposition (See Chapter 6). Then, a reasonable approximation error can be achieved with a small number of subintervals, consequently with less data computed and stored in the setup.

5. BASICS ON NUMERICAL APPROXIMATION

In Chapter 6, the Polynomial Density Inversion method will be explained, which is a random variate generation method based on the approximation of the density. Polynomial approximation will be used since there are simple approximation methods and root finding algorithms, which can be used in the method in an efficient way. Therefore, in this chapter, some basics on polynomial approximation are presented.

5.1. Newton Interpolation Polynomial

There are many types of interpolation methods in the literature. The Newton Interpolation is an efficient and fast method which is applicable for any type of single variable function. The unique polynomial obtained within the method; Newton Polynomial, which is named after its inventor Isaac Newton, is the interpolation polynomial for a given set of interpolation points in the Newton form.

Newton Form is a linear combination of Newton basis polynomials (Burden and Faires, 1997). For instance, given a set of $k+1$ interpolation points $(x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$, where no two x_j are the same, the unique polynomial in Newton form can be expressed as:

$$N(x) = \sum_{j=0}^{k} a_j n_j(x)$$

with the Newton basis polynomials are defined as:

$$n_j(x) = \prod_{i=0}^{j-1} (x - x_i)$$

and the coefficients are defined as:

$$a_j = d_{j,j}$$

where

$$d_{m,n} = \begin{cases} y_m & m = 0,1,\ldots,k \text{ and } n = 0 \\ \dfrac{d_{m,n-1} - d_{m-1,n-1}}{x_m - x_{m-n}} & m = 1,\ldots,k \text{ and } n = 1,\ldots,m \end{cases} \tag{5.1}$$

is the notation for divided differences with the restriction $n \le m$. Therefore, the Newton polynomial can be written as:

$$\begin{aligned} N(x) &= d_{0,0} + d_{1,1}(x - x_0) + d_{2,2}(x - x_0)(x - x_1) \\ &+ d_{3,3}(x - x_0)(x - x_1)(x - x_2) + \ldots \\ &+ d_{k,k}(x - x_0)(x - x_1)\ldots(x - x_{k-1}) \end{aligned}$$

Nevertheless, the formulation does not yield the desired coefficients $\{c_0, c_1, \ldots, c_k\}$ of the Newton Polynomial. Yet, with a little computation by using divided differences, it is easy to obtain those coefficients under a programming environment by the following formula:

$$c_i = \sum_{j=0}^{k} D_{i,j} S_{i,j}(x_0, x_1, \ldots, x_k) \tag{5.2}$$

where

$$D_{i,j} = (-1)^j (d_{i+j,i+j})$$

and

$$S_{i,j}(x_0, x_1, \ldots, x_k) = \begin{cases} 1 & i = 0,\ldots,k \text{ and } j = 0 \\ s_{j,i+j}(x_0, x_1, \ldots, x_k) & i = 0,\ldots,k-1 \text{ and } j = 1,\ldots,k-i \end{cases}$$

Here the notation $s_{m,n}(x_0, x_1, \ldots, x_k)$ expresses the summation of the element multiplication of the "m-element subsets" of the first n elements of the vector $(x_0, x_1, \ldots, x_k)$.

$$s_{m,n}(x_0, x_1, \ldots, x_k) = \begin{cases} 0 & m > n \text{ or } m \leq 0 \\ \sum_{i=0}^{n-1} x_i & m = 1 \\ x_0 \, s_{m-1,n-1}(x_1, x_2, \ldots, x_k) + s_{m,n-1}(x_1, x_2, \ldots, x_k) & \textit{otherwise} \end{cases}$$

It is clear that, in a programming environment, coefficients of the Newton interpolation polynomial can be obtained by recursively executing sub-functions. Thus, the computation of higher order approximations becomes expensive. In approximate random variate generation methods, the density should be approximated with piecewise polynomials, which have a degree of 4 or 6 at most.

5.1.1. Choosing Interpolation Points

In numerical analysis, Runge's Phenomenon is a problem that occurs when using polynomial interpolation with higher order polynomials. It was discovered by Carl David Tolme Runge (1901) when exploring the behavior of the error when using polynomial interpolation to approximate certain functions (See Figure 5.1). If such functions are interpolated at equidistant points x_i in the interval (b_l, b_r) such that:

$$x_i = b_l + (i-1)\frac{b_r - b_l}{n}, \quad i \in \{1, 2, \ldots, n+1\}$$

with a polynomial $P_n(x)$ of degree smaller than or equal to n, the resulting interpolation oscillates towards the boundaries of the interval. Even it is expected for the maximum interpolation error to converge to zero as the degree of the polynomial increases, according to Runge's phenomenon, interpolation error tends towards infinity because of that oscillation.

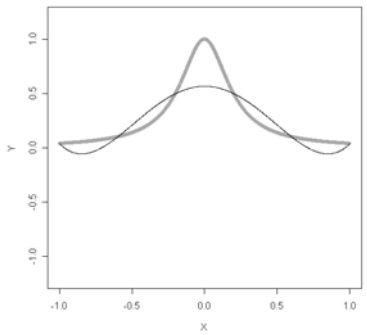
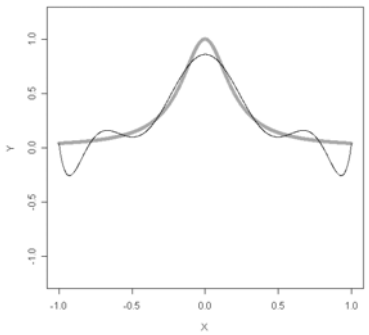
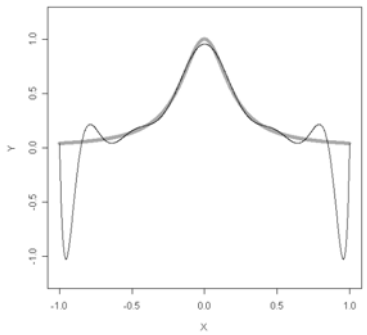
5^{th} Order Approximation	9^{th} Order Approximation	13^{th} Order Approximation
		

Figure 5.1. Illustration of the 5^{th}, 9^{th} and the13^{th} order approximations for Runge's function $f(x) = 1/(1+25x^2)$ with equidistant points

The main reason of that problem is easy to analyze. It is known that the error between the function itself and the interpolation polynomial of order N is bounded by the N^{th} derivative of the original function. Since, the magnitude of higher order derivatives of certain functions, which Runge has proposed, gets even larger, the upper bound for the error between interpolating points when using higher order polynomials becomes larger.

$$f(x) = \frac{1}{(1+25x^2)}$$

$$f'(x) = -\frac{50x}{(1+25x^2)^2} \qquad |f'(-1)| = \frac{50}{26^2} \approx 0.0740$$

$$f''(x) = \frac{5000(1+25x^2) - 50(1+25x^2)^2}{(1+25x^2)^4} \qquad |f''(-1)| = \frac{96200}{26^4} \approx 0.2105$$

Chebyshev points or Chebyshev nodes are a set of interpolation points with a particular sequence and are used for polynomial interpolation (Burden and Faires, 1997). Because the resulting interpolation polynomial minimizes the problem of Runge's Phenomenon and is almost optimal, in our algorithm, the approximate density is interpolated in Chebyshev points, which are rescaled in each subinterval. It's a fact that the oscillation occurring near the boundaries can be minimized by choosing interpolating points more frequent nearby the boundaries and less frequent in the center. Chebyshev nodes have this kind of property, such that the maximum error is guaranteed to diminish

with increasing polynomial order. Another method to cope with Runge's phenomenon is to approximate the function with piecewise polynomials. This will also be used in the Polynomial Density Rejection algorithm (See Chapter 6).

In order to obtain an interpolation polynomial with degree N in the interval $[-1,1]$, we need $N+1$ Chebyshev nodes such that:

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i \in \{0,\ldots,n\}$$

We rescale those points over an arbitrary interval $[b_l, b_r]$ by the linear transformation.

$$\tilde{x}_i = \frac{1}{2}(b_r + b_l) + \frac{1}{2}(b_r - b_l)\cos\left(\frac{2i+1}{2n+2}\pi\right), \quad i \in \{0,\ldots,n\}$$

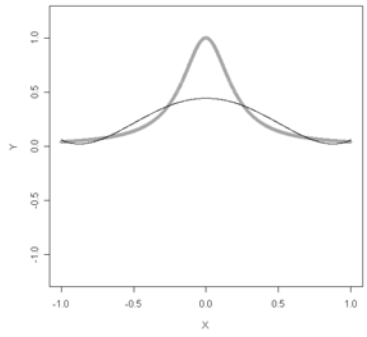
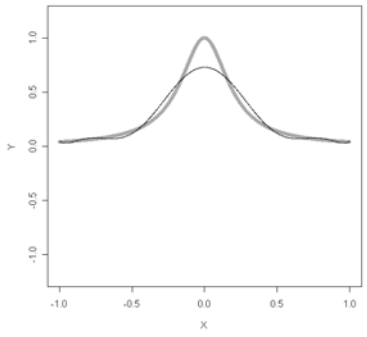
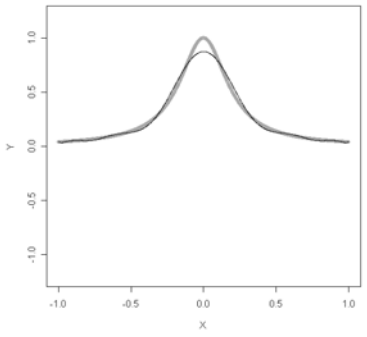
5th Order Approximation	9th Order Approximation	13th Order Approximation
		

Figure 5.2. Illustration of the 5th, 9th and the13th order approximations for Runge's function $f(x) = 1/(1+25x^2)$ with Chebyshev nodes

However, we also use the boundaries $\{b_l, b_r\}$ as interpolation points, since we want no approximation error at the interval boundaries. This is because, for the approximate random variate generation methods, the approximated density should be consistent for neighboring subintervals on their intersection points and possible negative approximated densities must be eliminated over the tails like it was done for the piecewise linear

approximation (See Figure 4.3). Therefore, we need an appropriate scaling, such that the first and the last interpolation points are equal to the boundaries. In order to do that, we need the normalized distances between interpolation points:

$$\theta_i = \sin\left(\frac{i\pi}{N+1}\right)\tan\left(\frac{\pi}{2N+2}\right), \quad i \in \{0,\ldots,n\} \tag{5.3}$$

Then, we can use these distances, which sum up to one, as a scale while calculating actual interpolation points in the interval $[b_l, b_r]$.

$$\tilde{x}_i' = b_l + (b_r - b_l)\sum_{k=0}^{i}\theta_i, \quad i \in \{0,\ldots,n\} \tag{5.4}$$

See Figure 5.3 for the illustration of the 5th, 9th and the13th order approximations for Runge's function with rescaled Chebyshev nodes.

5.1.2. Interpolation Polynomial Error Analysis

As it was mentioned before, in order to cope with Runge's Phenomenon, we need to use Chebyshev points for interpolation. If it is not enough for a reasonable approximation, we need to split the subinterval into two parts.

Needless to say, that a good approximation has a small interpolation error. To evaluate the performance of the approximation, we need to look at the maximum absolute error ε through the subinterval $[b_l, b_r]$, which is:

$$\varepsilon = \max_{x\in[b_l,b_r]} \left| f(x) - P_N(x) \right|$$

where P_N is the interpolation polynomial of order N of the original function f.

Since the function and the interpolation polynomial are continuous over the interval, the maximum error can be obtained by checking the local extrema by taking the

first derivative of the difference and equating it to zero. However, solving this equation with $N-1$ roots can be slow.

However, we can construct control points inspired by the Chebyshev points, which may not give the exact solution for the problem but still will be close to the local extrema (Burden and Faires, 1997).

For evaluating the error of interpolation polynomial of order N in the interval $[b_l, b_r]$, we need N control points such that:

$$\hat{x}_i = \frac{1}{2}(b_r + b_l) + \frac{(b_r - b_l)}{2} \frac{\cos\left(\frac{\pi(N+1-i)}{N+1}\right)}{\cos\left(\frac{\pi}{2N+2}\right)}, \quad i = 1, \ldots, N \tag{5.5}$$

Then we can find the approximated maximum absolute error between interpolation polynomial and the original function with the following formula.

$$\varepsilon' = \max_{i=1,\ldots,N} \left| f(\hat{x}_i) - P_N(\hat{x}_i) \right| \tag{5.6}$$

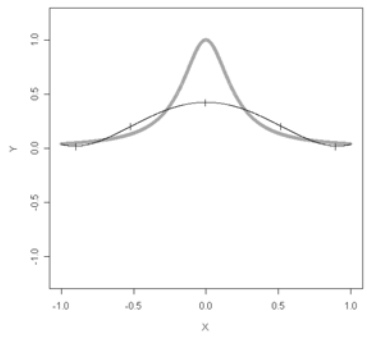
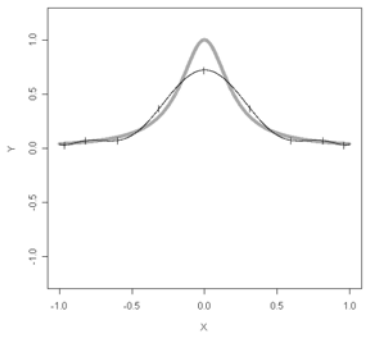
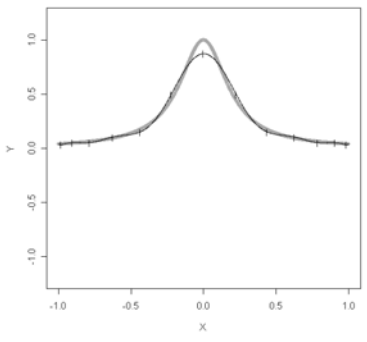
5th Order Approximation	**9th Order Approximation**	**13th Order Approximation**
		

Figure 5.3. Illustration of the 5th, 9th and the13th order approximations and control points for Runge's function $f(x) = 1/(1+25x^2)$ with rescaled Chebyshev nodes

As a result, an approximation is good if the approximated maximum absolute error ε' is smaller than or equal to a critical interpolation error, which is a parameter that can be selected before the execution of the random variate generation algorithm.

Table 5.1. Maximum absolute approximation error with rescaled Chebyshev points at Chebyshev control points (CCP) and 10000 equidistant control points (ECP) in the subinterval $[0.4, 0.5]$ for different degrees of approximations and different distributions

		Degrees of Approximation Polynomial							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
N	**CCP**	8.1024E-03	3.5923E-04	3.6746E-06	3.0121E-08	4.4483E-10	2.0273E-12	3.9857E-14	1.1102E-16
	ECP	8.1024E-03	3.5929E-04	3.6747E-06	3.0121E-08	4.4483E-10	2.0274E-12	3.9857E-14	1.6653E-16
C	**CCP**	9.8786E-03	1.8027E-04	1.0641E-05	1.1482E-07	2.7909E-09	9.9275E-11	5.9297E-13	4.0801E-14
	ECP	9.8786E-03	1.8130E-04	1.0641E-05	1.1482E-07	2.7911E-09	9.9275E-11	5.9297E-13	4.0801E-14
E	**CCP**	3.1895E-02	79720E-04	5.1507E-06	2.8712E-08	1.3434E-10	5.3590E-13	1.8874E-15	1.1102E-16
	ECP	3.1895E-02	7.9726E-04	5.1508E-06	2.8712E-08	1.3434E-10	5.3602E-13	1.9984E-15	2.2204E-16
G	**CCP**	1.1095E-02	1.6111E-04	9.1196E-06	1.2430E-07	1.0592E-09	6.6651E-12	3.3383E-14	1.5266E-16
	ECP	1.1095E-02	1.6195E-04	9.1200E-06	1.2431E-07	1.0592E-09	6.6651E-12	3.3404-14	1.9429E-16
B1	**CCP**	9.9300E-02	4.7156E-02	1.1354E-03	5.0717E-05	1.5061E-06	2.6645E-15	3.7748E-15	2.6645E-15
	ECP	9.9300E-02	4.7197E-02	1.1359E-03	5.0721E-05	1.5061E-06	5.3291E-15	5.9952E-15	6.2172E-15
B2	**CCP**	1.4391E+00	1.5762E+00	2.5628E-01	4.6967E-02	8.0515E-03	9.2686E-04	1.8114E-04	1.2375E-05
	ECP	1.4391E+00	1.6200E+00	2.5656E-01	4.7001E-02	8.0515E-03	9.2713E-04	1.8114E-04	1.2378E-05

Table 5.1 shows how the maximum absolute approximation error behaves in the subinterval $[0.4, 0.5]$ on Chebyshev control points and 10000 equidistant control points for approximations of degrees zero to seven and different distributions. It is clear that checking the error on Chebyshev control points is almost as precise as checking it on 10000 equidistant control points. Thus, it is better to use Chebyshev control points for a fast control. Also, in general, the results indicate that the maximum absolute error is roughly in the magnitude of order 10^{-2N}, where N is the degree of the approximation polynomial.

5.2. Root Finding with Brent's Method

Assume that we have a polynomial quasi-density, which is strictly positive over the domain $[b_l, b_r]$. Then it is easy to evaluate the cumulative quasi-density by integration. Since we are not able to find the inverse cdf, we apply a search algorithm to solve the equation of the inversion method. Brent's Method (1973) is a good numerical search algorithm with a guaranteed fast convergence. It is a combination of popular root-finding algorithms like bisection method, secant method and inverse quadratic interpolation. It has

the reliability of the bisection method, and the convergence is often as quick as the convergence of the other two methods (Brent, 1973).

Require: Interval boundaries $\{a,b\}$ such that only one root of the equation exists in the interval $[a,b]$ and $f(a)f(b) \geq 0$, a subroutine for continuous function f; cdf of the quasi-density.
Output: One and the unique real root of $f(x) = 0$.

Calculate $f(a)$.
Calculate $f(b)$.
if $|f(a)| \leq |f(b)|$ **then**
    Swap a and b.
Set $c \leftarrow a$.
Set $m \leftarrow 1$.
loop
    if $f(a) \neq f(c)$ **and** $f(b) \neq f(c)$ **then**
        $$s \leftarrow \frac{a\,f(b)f(c)}{(f(a)-f(b))(f(a)-f(c))} + \frac{b\,f(a)f(c)}{(f(b)-f(a))(f(b)-f(c))} + \frac{c\,f(a)f(b)}{(f(c)-f(a))(f(c)-f(b))}$$ /* inverse quadratic interpolation */
    else
        $$s \leftarrow b - f(b)\frac{b-a}{f(b)-f(a)}$$ /* secant rule */
    if s is not between $\frac{(3a+b)}{4}$ and b **or** ($m = 1$ **and** $|s-b| \geq |b-c|/2$)
    or ($m = 0$ **and** $|s-b| \geq |c-d|/2$) **then**
        $$s \leftarrow \frac{a+b}{2}$$
        Set $m \leftarrow 1$.
    else
        Set $m \leftarrow 0$.
    Calculate $f(s)$.
    Set $d \leftarrow c$.
    Set $c \leftarrow b$.
    if $f(a)f(s) < 0$ **then**
        Set $b \leftarrow s$.
    else
        Set $a \leftarrow s$.
    if $|f(a)| \leq |f(b)|$ **then**
        Swap a and b.
    if $f(b) = 0$ **or** $|b-a| \leq \delta$ **then**
        return b. /* return the root */

Figure 5.4. Brent's Method

6. POLYNOMIAL DENSITY INVERSION

Approximate methods in random variate generation, commonly use the piecewise constant (uniform) or piecewise linear (1^{st} order) approximations. These methods are based on the piecewise linear approximation of the cdf and the density. They can, thus, be seen as the Ahrens or the Triangular Ahrens Method without rejection.

Although, these methods are suggested in parts of the literature, they have a large maximum absolute approximation error or require too many subintervals. The aim of the Polynomial Density Inversion algorithm is to obtain a better approximation, without increasing the number of subintervals and influencing the speed of the algorithm.

6.1. Basics of the Algorithm

The Polynomial Density Inversion algorithm is based on the decomposition method and the polynomial approximation of the density. Since polynomials are easy to integrate and easy to invert with search algorithms, the inversion method can easily be applied to polynomial quasi-densities.

After dividing the domain of the distribution in many subintervals, for each subinterval, a trapezoid can be defined below the density. This trapezoid actually consists of a rectangular (uniform) and a triangular distribution. Since generating random variates from these distributions is easy and fast, trapezoids should cover most of the region below the density. Then the rest of the subinterval region, which is below the density and above the trapezoid, can be approximated with a polynomial. The probability or the weight of that polynomial region can easily be obtained by integration.

As short subintervals with monotone densities are chosen, the behavior of the density gets closer to linear. For such densities, smaller polynomial interpolation errors are available as the number of interpolation points increases. That allows us to obtain a maximum absolute approximation error which has a magnitude of order 10^{-12} with just five interpolation points (See Table 5.1).

Of course, to execute the sampling algorithm, we need a table of subinterval boundaries and cumulative subinterval probabilities, the areas of the rectangular and the triangular regions and the coefficients of the polynomials. As the number of interpolation points increases, the number of interpolation coefficients also increases. That results in a data table of undesirable size. So, a wise decision should be made about the number of interpolation points, which should result in a reasonable interpolation error and a small table, which will speed up the setup and occupy less memory.

Since the sampling algorithm will successfully generate a random variate of the density each time it is executed, there will be no need for retrying.

6.1.1. Handling Monotone Subintervals

For a subinterval, the probability for the polynomial part must be as small as possible. Since it can be handled only with simple arithmetic operations, generating random variates from rectangular (uniform random variates) and triangular (maximum or minimum of two uniform random variates) distributions are not expensive. The triangular distribution requires two uniform random variates. On the other hand, calling the cdf of a polynomial density and executing a search algorithm with many repetitions takes a significant time. So we need to maximize the trapezoidal region below the density.

6.1.1.1. Concave Case. It is easy to see that the maximum trapezoidal region can be obtained by a secant which intersects the density at the subinterval boundaries for concave subintervals (See Lemma 3.2).

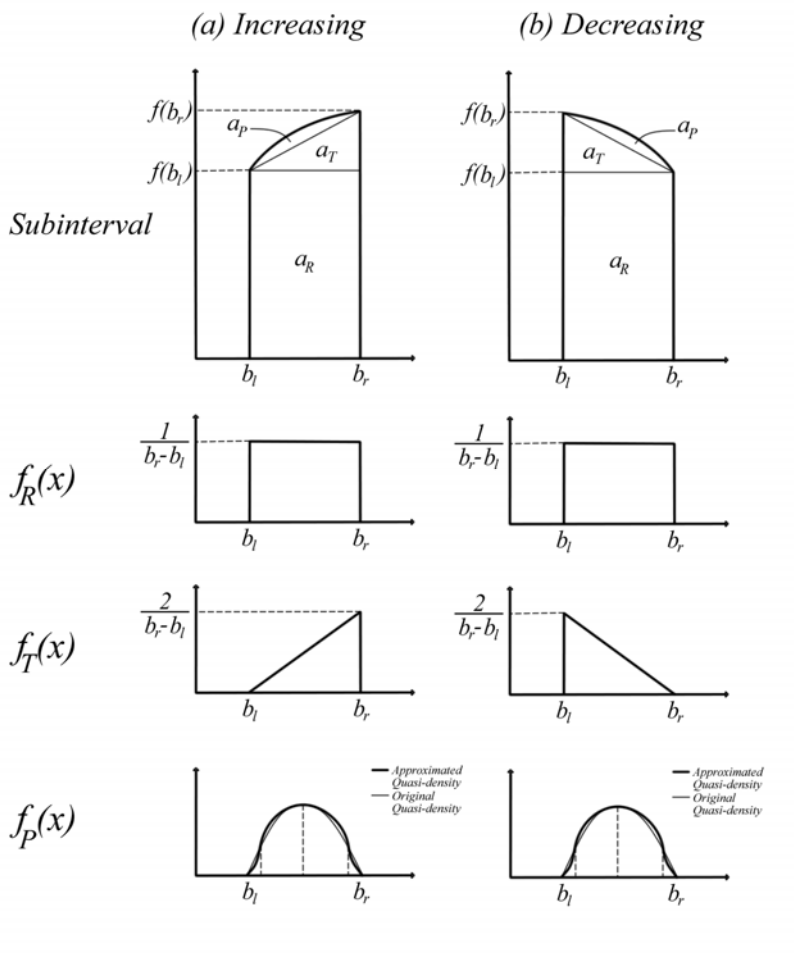


Figure 6.1. Decomposition of concave subintervals

For the subinterval $[b_l, b_r]$ with an increasing concave density, the region below the density of the left boundary $f(b_l)$ describes the uniform distribution with a probabilistic weight of its proportional area. The triangular region above is stored by its parameters (b_l, b_r, b_r). It is equivalent to the distribution that is generated by the maximum of two uniform random variates in the subinterval (See Lemma 6.2). The remaining concave region must be decomposed by subtracting the secant, and the interpolation points must be well chosen. An interpolation algorithm with $k+1$ interpolation points compute the polynomial coefficients $(c_0, c_1, \ldots, c_k)$. Integration of this polynomial within the interval $[b_l, b_r]$ is necessary to evaluate the probabilistic weight of this region.

The only difference for decreasing concave densities is caused by the triangular distribution. The upper bound for the rectangular region is now the density value of the

right boundary $f(b_r)$. On the other hand, since the triangular distribution must be decreasing as well, the parameters will change into (b_l, b_l, b_r). The distribution is actually equivalent to the minimum of two uniform random variates in the subinterval.

The concave subinterval, then, can be represented by the composition formula:

$$f(x) = w_r f_r(x) + w_t f_t(x) + w_p f_p(x)$$

where:

$$w_i = \frac{a_i}{a_r + a_t + a_p}, \quad i \in \{r, t, p\}$$

$$a_r = \begin{cases} f(b_l)(b_r - b_l) & \textit{increases} \\ f(b_r)(b_r - b_l) & \textit{decreases} \end{cases} \tag{6.1}$$

$$a_t = \frac{|f(b_r) - f(b_l)|(b_r - b_l)}{2} \tag{6.2}$$

$$a_p = \int_{b_l}^{b_r} \sum_{i=0}^{k} c_i x^i \, dx \tag{6.3}$$

$$f_r(x) = \frac{1}{b_r - b_l}$$

$$f_t(x) = \begin{cases} \dfrac{2(x - b_l)}{(b_r - b_l)^2} & \textit{increases} \\ \dfrac{2(b_r - x)}{(b_r - b_l)^2} & \textit{decreases} \end{cases}$$

$$f_p(x) = \frac{\sum_{i=0}^{k} c_i x^i}{a_p}$$

6.1.1.2. Convex Case It is desirable to find decomposition routines which are similar to each other for all types of subintervals. However, for the convex case, it is not a good idea to construct a trapezoid using a tangent to the density in the center point of the interval.

For instance, in the tail parts of the normal distribution, a tangent constructed in the center point intersects with the x-axis. This is undesirable since it ruins the trapezoidal shape and it would require additional constants.

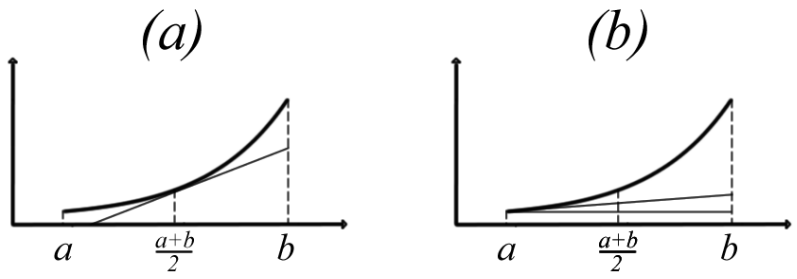


Figure 6.2. (a) Tangent in the center point ruins the trapezoid. (b) Tangent in the boundary with the smaller density value

Lemma 6.1: Let f be a density function that is strictly convex and monotone over the domain $[b_l, b_r]$. In order to construct a trapezoid below the density, the tangent for the trapezoid must be constructed in the boundary, which has the minimum density value.

Proof. Let f be a monotonic density. Then any tangent constructed in a point X will not intersect with x-axis in the subinterval $[b_l, b_r]$, if the tangent has the following property:

$$g_X(x) \geq g_X(X) = f(X) \quad for\ \forall x \in [b_l, b_r]$$

That is guaranteed if X is chosen as the boundary, which has the minimum density value.

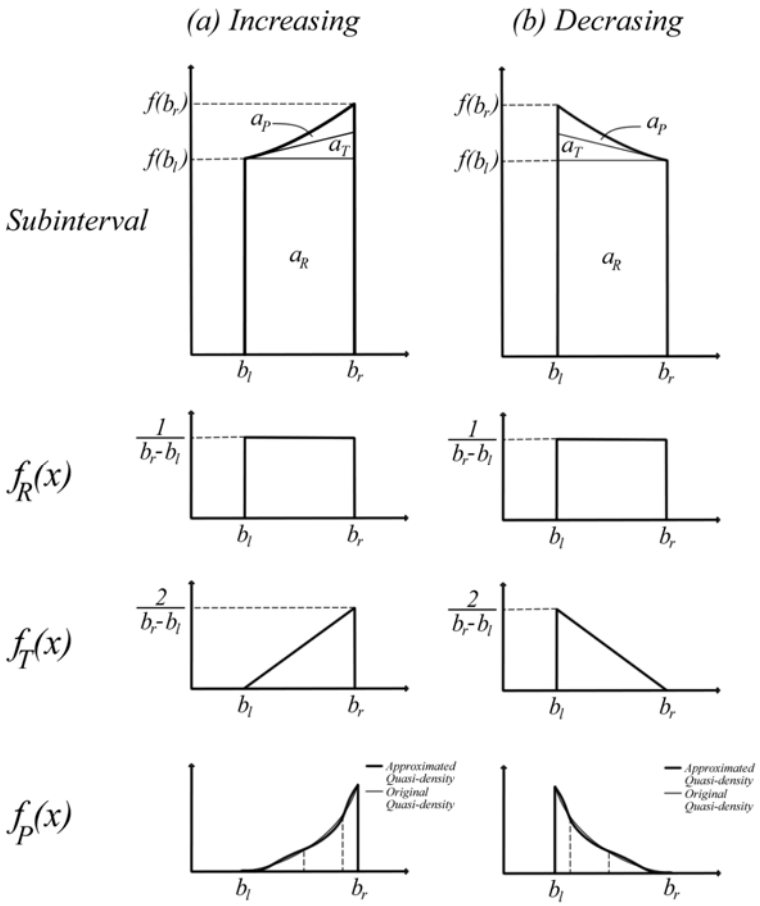


Figure 6.3. Decomposition of convex subintervals

The difference in the decomposition of increasing convex subintervals is the triangular region; it is below the tangent, which is constructed in the boundary that has the smaller density value. Even if the shape of the polynomial region changes into convex densities, the idea remains the same. Again, the decomposition of decreasing convex densities has the same modifications as the decomposition of decreasing concave densities.

Then the composition formula for the concave case, Equation 6.1, 6.2 and 6.3, can also be used for convex subintervals, only Equation 6.2 has to be replaced by:

$$a_t = \begin{cases} \dfrac{f'(b_l)(b_r - b_l)^2}{2} & \textit{increases} \\ -\dfrac{f'(b_r)(b_r - b_l)^2}{2} & \textit{decreases} \end{cases} \tag{6.4}$$

6.1.2. Monotone Triangular Distribution

The triangular distribution is a continuous probability distribution with lower limit a, mode c and upper limit b [e].

$$f(x;a,b,c)=\begin{cases}\dfrac{2(x-a)}{(c-a)(b-a)} & \text{for } a\le x\le c\\ \dfrac{2(b-x)}{(b-a)(b-c)} & \text{for } c\le x\le b\\ 0 & \text{otherwise}\end{cases}$$

In our algorithm, the monotonic triangular distribution is used as one of the limits and the mode are the same. In that case, we have a new density function with two parameters only; the lower and the upper limit, and an index to express whether it is an increasing or a decreasing density.

$$f_I(x;a,b)=\begin{cases}\dfrac{2(x-a)}{(b-a)^2} & a\le x\le b\\ 0 & \text{otherwise}\end{cases}$$

$$f_D(x;a,b)=\begin{cases}\dfrac{2(b-x)}{(b-a)^2} & a\le x\le b\\ 0 & \text{otherwise}\end{cases}$$

Lemma 6.2: The maximum of two independent and identically distributed uniform random variates over the interval $[a,b]$ is distributed with the density of an increasing triangular distribution. Then the minimum of two independent and identically distributed uniform random variates over the interval follows a decreasing triangular distribution.

Proof. Let x be a certain random variate obtained by taking the maximum of two independent and identically distributed standard uniform random variates, u_1 and u_2, over the domain $[a,b]$. The probability of obtaining x can be evaluated by considering two cases.

$$P(x) = P(u_1 = x)P(u_2 \le x) + P(u_2 = x)P(u_1 \le x)$$
$$P(x) = \frac{1}{b-a}\frac{x-a}{b-a} + \frac{1}{b-a}\frac{x-a}{b-a} = \frac{2(x-a)}{(b-a)^2} = f_I(x;a,b)$$

If the minimum of those uniform random variates is considered, then the probability of obtaining x can be evaluated by the following equation:

$$P(x) = P(u_1 = x)P(u_2 \ge x) + P(u_2 = x)P(u_1 \ge x)$$
$$P(x) = \frac{1}{b-a}\frac{b-x}{b-a} + \frac{1}{b-a}\frac{b-x}{b-a} = \frac{2(b-x)}{(b-a)^2} = f_D(x;a,b)$$

There are other ways of generating random variates from monotone triangular distributions. Of course the inversion method can be used. Since the cdf of the monotone triangular distribution is a second order polynomial and easy to invert using a quadratic equation, a random variate can be generated with only one uniform random variate.

$$F_I(x;a,b) = \begin{cases} \dfrac{(x^2 - 2ax)}{(b-a)^2} & a \le x \le b \\ 0 & otherwise \end{cases}$$
$$F_D(x;a,b) = \begin{cases} \dfrac{(2bx - x^2)}{(b-a)^2} & a \le x \le b \\ 0 & otherwise \end{cases}$$

It is easy to show that, for the increasing triangular distribution, solving the quadratic equation of the cdf with a uniform random variate is equivalent to taking the square root of that uniform random variate and rescale it over the interval. On the other hand, for the decreasing triangular distribution, it is equivalent to take the square root of the uniform variate and rescale it over the subinterval after subtracting it from one. This alternative method does not require an additional uniform random variate but requires a call to the square root function. It must be tested whether the computation of the square root or generating another random variate is expensive.

6.1.3. Polynomial Approximation and Approximated Density Error Analysis with Importance Sampling

The piecewise approximations may result in the best possible performance by using Newton interpolation and Chebyshev nodes for a fixed subinterval. Still the approximation performance may differ over the subintervals depending on the density. In order to evaluate that performance, we need to integrate the absolute error over the whole density. We use the L_1 error defined by:

$$L_1 - error = \int_{b_{l_i}}^{b_{r_i}} \left| \tilde{f}(x) - f(x) \right| dx$$

Here, $\tilde{f}(x)$ denotes the approximated density function. The approximation simply can be done without dividing the subinterval into rectangular and triangular regions, but it can be done afterwards too. It is hard to compute the L_1 error in closed form. Yet, it is possible to evaluate it by simulation.

The problem can be handled by using importance sampling. Since there are other methods of generating random variates from certain densities without approximation (e.g. the Triangular Ahrens Method), we can generate random variates from an appropriate importance sampling distribution and calculate the absolute approximation error of the Polynomial Density Inversion method for each of them. After dividing each absolute error by its corresponding importance sampling density, the mean and the variance of those will give an insight about the L_1 error. Following integration defines the L_1 error with the importance sampling density $g(x)$.

$$L_1 - error = \int_{b_{l_i}}^{b_{r_i}} \left| \tilde{f}(x) - f(x) \right| dx = \int_{b_{l_i}}^{b_{r_i}} \frac{\left| \tilde{f}(x) - f(x) \right|}{g(x)} g(x) dx = \int_{b_{l_i}}^{b_{r_i}} q(x) g(x) dx$$

It is possible to show that the variance of the results is minimized for the importance sampling density $g^*(x)$:

$$g^*(x) = \frac{|q(x)f(x)|}{\int |q(x)f(x)|\,dx}$$

In practice we cannot use this result, since the denominator is unknown. But we should remember the rule that the importance sampling density $g(x)$ should mimic the behavior of $q(x)f(x)$ (Hörmann, 2007).

Evaluating the L_1 error, yet, is too expensive for testing the acceptance of piecewise approximations. Because a shorter setup time is preferred, it is appropriate to use some heuristics in order to test the error of piecewise approximations in the setup. These heuristics are difficult to interpret but easier to calculate.

The heuristics for creating flexible subintervals will be explained in the setup algorithm. L_1 error obtained with these heuristics over different densities will be examined after the whole algorithm is explained.

6.2. The Algorithm

The Polynomial Density Inversion algorithm consists of two parts. Initially, a setup algorithm must calculate the necessary tables and then random variates can be generated by the sampling algorithm.

6.2.1. The Setup Algorithm

To run the sampling algorithm, initially a table must be created and the required constants for each subinterval must be stored in it. This is done in a setup algorithm. Below are the basic required elements of that table.

- A vector that holds the subinterval boundaries.
- Cumulative distribution probabilities for each subinterval.
- Cumulative distribution probabilities of rectangular and triangular regions for each subinterval.

- A table that holds the coefficients of approximation polynomials of order K, for each subinterval.
- A guide table to speed up discrete subinterval selection with the Indexed Search method (two times larger than the number of subintervals).

In order to compute the constants of the table, initially the density must be divided into reasonable subintervals with certain rules.

6.2.1.1. Phase I: Flexible Subinterval Creation. The aim of creating those subintervals is to decrease the approximation error and probability of the polynomial region. For each monotone interval $[b_l, b_r]$, three different values are computed in order to compare them with critical values of subinterval acceptance.

Linear Error (ε): It is the difference between the density and the secant on the center point of the subinterval.

$$\varepsilon = f\left(\frac{b_r + b_l}{2}\right) - \frac{f(b_r) + f(b_l)}{2} \tag{6.5}$$

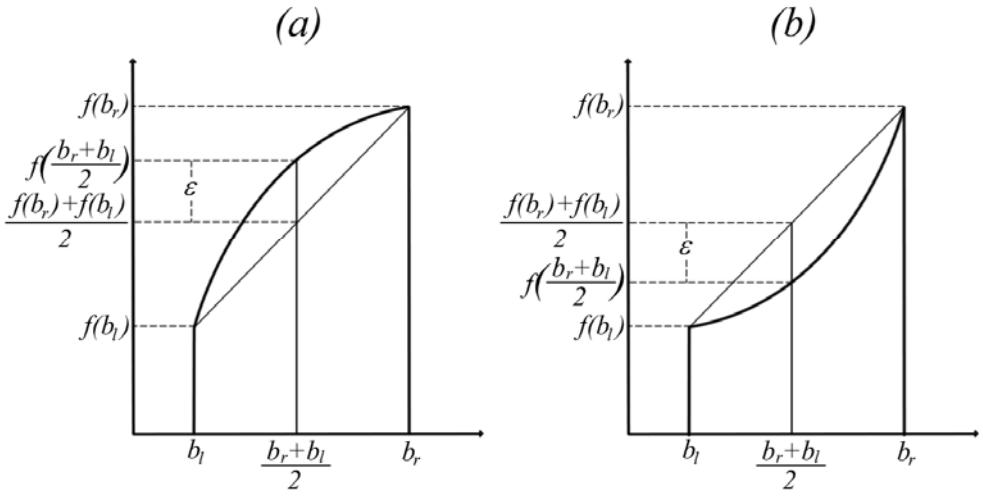


Figure 6.4. (a) Linear error in concave subintervals (b) Linear error in convex subintervals

As this value gets smaller, the density behaves closer to linear. Thus, a better approximation can be obtained and the rectangular and the triangular regions become

relatively larger. That decreases the probability of generating from the polynomial region in the sampling algorithm. So, it is necessary to compute ε and compare it with a critical value ε_c which is defined by the user. That is also a subinterval acceptance criterion for the piecewise linear approximation algorithm, which approximates all subintervals as a single trapezoid with only the rectangular and the triangular regions (See Section 4.2).

Relative Linear Error (r): In the tails of unimodal distributions, even if we have a small linear error, the convex behavior of the density may not allow us to obtain a good polynomial approximation. If the local minimum of the subinterval is much smaller than the local maximum, it may not be possible to obtain a good approximation. Therefore, we need to calculate a relative error, r, which should be smaller than the critical value, r_c.

$$r = \frac{\varepsilon}{f\left(\frac{b_r + b_l}{2}\right)} \tag{6.6}$$

In the tails of unimodal distributions, rectangular and triangular regions will still have a high probability and it will be possible to obtain good polynomial approximation, if r_c is selected small enough.

However, there is still need for the linear error check since the relative error check is not sufficient to create subintervals over the main part of the distribution. Therefore, the linear error and the relative error check complete each other in order to create flexible subintervals in an efficient way.

Polynomial Error (ε´): Although the linear error and the relative error checks are sufficient to create flexible subintervals, they do not give a clue about the approximation errors, which should have reasonably small values.

The polynomial error is hard to calculate. It can only be calculated after calculating interpolation of the approximate polynomial. Therefore, it should only be computed after the linear and the relative linear errors were computed and seemed acceptable. It is

necessary to use the formulas (Equation 4.1, 4.2, 4.3 and 4.4) in Chapter 4 to calculate polynomial error, ε', using the Chebyshev control points, $\hat{x}_i$. Then we need to compare the polynomial error with its critical value, ε'_c, defined by the user.

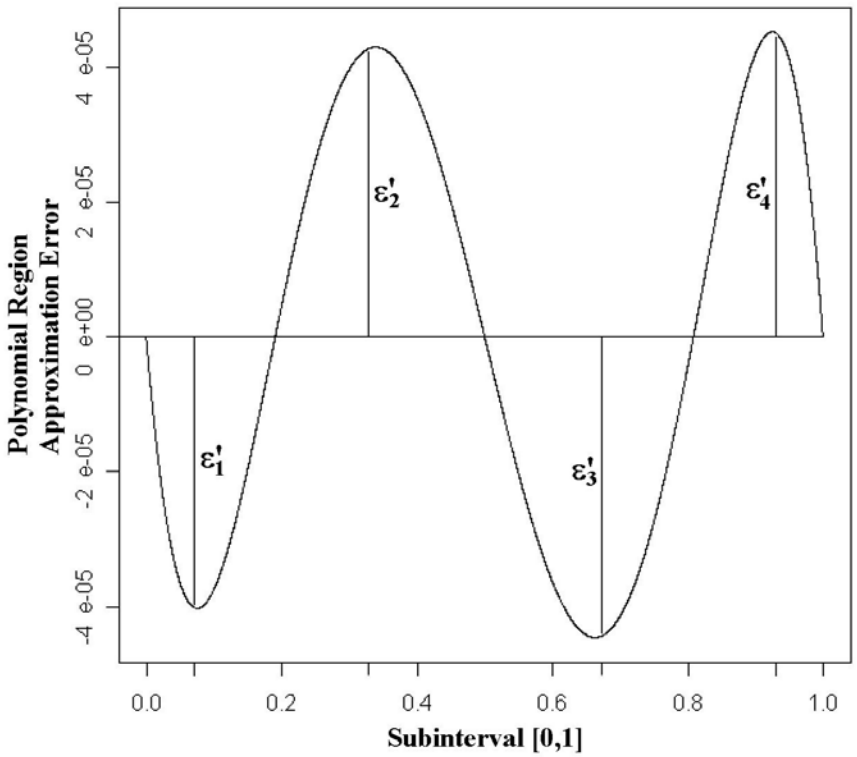


Figure 6.5. Polynomial region 4th order approximation error of standard normal distribution over the subinterval $[0,1]$ with Chebyshev control points

$$\{0.0746746, 0.3375402, 0.6624598, 0.9253254\}$$

As it was previously mentioned, the polynomial error should be small enough to get a good approximation of the density. Then the generated random variates really follow the distribution of the given density.

The logic of flexible subinterval creation is simple. Considering local extrema, cutoff and inflection points as an initial set of subinterval boundaries, we need to question the acceptance of each subinterval by their ε, r values first, and then by the ε' value.

Require: Probability density function $f(x)$, the first derivative of the pdf $f'(x)$, critical value ε_c for the difference between the density and the secant on the center point, critical value r_c for the relative proportion of the density-secant error over the density, critical value ε_c' for the maximum interpolation error, order of interpolation polynomials K, a sequential set Z of all local extrema, cutoff and inflection points $(z_0, z_1, \ldots, z_m)$ from the smallest to largest, which will also help to explain the behavior of the density.

Output: A vector that holds sequential flexible subinterval boundaries $(b_0, b_1, \ldots, b_k)$, and its length.

1: Set $i \leftarrow 0$.

2: Set $b_i \leftarrow z_i$

2: **while** $b_i \neq z_m$ **do**

3: Compute ε for subinterval $[b_i, z_{i+1}]$. /* use Equation 6.5 */

4: Compute r for subinterval $[b_i, z_{i+1}]$. /* use Equation 6.6 */

5: **if** $\varepsilon \leq \varepsilon_c$ **and** $r \leq r_c$ **then**

6: Interpolate $K+1$ Chebyshev nodes over $[b_i, z_{i+1}]$.

/* use Equation 4.1, 4.2, 4.3 and 4.4 and obtain coefficients */

7: Compute ε' for subinterval $[b_i, z_{i+1}]$. /* use Equation 4.5 and 4.6 */

8: **if** $\varepsilon' \leq \varepsilon_c'$ **then** /* use subinterval and proceed */

9: $b_{i+1} \leftarrow z_{i+1}$

10: $i \leftarrow i+1$

11: **else** /* divide subinterval */

12: Add $\frac{b_i + z_{i+1}}{2}$ to set Z and enumerate the elements.

13: **else** /* divide subinterval */

14: Add $\frac{b_i + z_{i+1}}{2}$ to set Z and enumerate the elements.

15: **return** $i+1$.

Figure 6.6. Polynomial Density Inversion setup - flexible subinterval creation

In the end of the algorithm, the number of subintervals is returned together with the vector of subinterval boundaries. These results are used as input in the next phase.

6.2.1.2. Phase II: Data Table Creation. After defining the boundaries of reasonable subintervals in Phase I, it is now possible to use them to compute the required data table for the sampling algorithm. Phase II also requires the probability density function and its first derivative for the table creation process. Again, a vector that holds all inflection points and the local extrema will be needed to analyze the behavior of the density, since the calculation of table elements will depend on the behavior of the subinterval. This vector will help to introduce easy to evaluate functions for table elements ca_i (unnormalized cumulative area of the subinterval) and $co_i^{(0)}, co_i^{(1)}, \ldots, co_i^{(K)}$ (coefficients of approximated polynomials of order K) for subinterval $i = 1, ..., N$.

ca_i (unnormalized cumulative area of the subinterval) will help to calculate the cumulative probability vector which will be used in the Indexed Search algorithm to choose a subinterval and a region with a uniform random variate. Actually, the cumulative area vector is also useful while it is in the unnormalized form. But since less mathematical operations in the generation algorithm are preferred, this vector is normalized in the next step of phase II, by dividing all elements with its last element. After initializing ca_0 to 0, other values of cumulative areas can be evaluated with the formula;

$$ca_m = \sum_{i=1}^{m} \left(a_{r\,i} + a_{t\,i} + a_{p\,i} \right) = ca_{m-1} + a_{r\,m} + a_{t\,m} + a_{p\,m} \qquad m = 1, \ldots, N \qquad (6.7)$$

where index i shows that the corresponding value is related with subinterval $[b_{i-1}, b_i]$.

As the last step, a guide table should be created by using the normalized cumulative area vector. In our applications, this guide table will be two times larger than the number of subintervals (Hörmann *et al.*, 2004).

Require: Probability density function $f(x)$, the first derivative of the pdf $f'(x)$, a vector that holds cut-off points, all local extrema and inflection points $(z_0, z_1, \ldots, z_M)$ which will explain the behavior of the density, a vector of length $N+1$ that holds sequential subinterval boundaries $(b_0, b_1, \ldots, b_N)$, order of interpolation polynomials K.

Output: A vector that holds the cumulative probability of each subinterval $(P_0, P_1, \ldots, P_N)$, a vector that holds cumulative probability of rectangular regions in each subinterval $(rc_1 \ldots, rc_N)$, a vector that holds cumulative probability of triangular regions in each subinterval $(tr_1, \ldots, tr_N)$, a vector that holds $K+1$ coefficients for each polynomial of order K in each subinterval $\left(co_1^{(k)}, co_2^{(k)}, \ldots, co_N^{(k)}\right)$ for $k = 0,1,\ldots,K$, a guide table of size $2 \times N$ for the indexed search in subinterval selection $(g_0, \ldots, g_{2\times N-1})$.

1: Store b_i for $i = 0, \ldots, N$

2: Set $i \leftarrow 1$, $ca_0 \leftarrow 0$.

3: **while** $i \le N$ **do**

4: Compute a_{ri} and a_{ti}. /* use Equation 6.1, 6.2 and 6.4 * /

5: Interpolate $K+1$ Chebyshev nodes over $[b_{i-1}, b_i]$.

/* use Equation 4.1, 4.2, 4.3 and 4.4 and obtain coefficients */

6: Store $co_i^{(k)}$ for $k = 0,1,\ldots,K$

7: Compute a_{pi}. /* use equation 6.3 * /

8: Store $rc_i = \dfrac{a_{ri}}{a_{ri} + a_{ti} + a_{pi}}$ and $tr_i = \dfrac{a_{ri} + a_{ti}}{a_{ri} + a_{ti} + a_{pi}}$.

9: Compute ca_i. /* use Equation 6.7 * /

10: Store $P_i \leftarrow \dfrac{ca_i}{ca_N}$ for $i = 0, \ldots, N$. /* normalization */

11: Set $g_0 \leftarrow 1$, $i \leftarrow 1$.

12: **for** $j = 1$ **to** $2N-1$ **do** /* guide table creation */

13: **while** $j/2N > P_i$ **do**

14: Set $i \leftarrow i+1$.

15: Set $g_j \leftarrow i$.

Figure 6.7. Polynomial Density Inversion setup - data table creation

6.2.2. The Sampling Algorithm

After the required data are stored in the table, finally random variates of the corresponding density can be generated. Different from the setup, the sampling algorithm only needs to know whether the density is increasing or decreasing, since the type of the triangular distribution depends on that. It is possible to use the mode for unimodal distributions to analyze the density but we could use indicator variables, which could optionally be added in the table to define the property of the subinterval in the setup algorithm.

The sampling algorithm initially must decide about which subinterval to consider. Therefore, by using the guide table $(g_0, g_1, \ldots, g_{2\times N-1})$ and cumulative probabilities of subintervals $(P_0, P_1, \ldots, P_N)$, the Indexed Search algorithm is run to generate index i. Then another standard uniform variate should be recycled from the one which was used for the Indexed Search algorithm.

With this recycled uniform random variate, the region that will be used in the subinterval must be selected by using the cumulative probabilities of the regions in the subinterval. It is possible to recycle this uniform variate once again over the cumulative region probabilities. Then, a uniform random variate X of the corresponding subinterval can be generated.

If the chosen region is the rectangular region, X is uniformly distributed over the corresponding interval and can be immediately returned. If it is the triangular region, another uniform variate over the subinterval must be generated and depending on the type of the triangular distribution, the maximum or the minimum is returned after the comparison (See Section 6.1.2).

However, the most difficult algorithm is necessary for the polynomial region. The uniform variate, which is recycled for a second time, can be used on the inverse cdf of the polynomial, which is $P^{-1}(x)$. By using Brent's method (1973), the unique solution of the inverse cdf over the subinterval is returned as random variate for the polynomial region.

$$P_i(x) = \int \sum_{j=0}^{K} co_i{}^{(j)} x^j \, dx = \sum_{j=1}^{K+1} \left(\frac{co_i{}^{(j)} x^j}{j} \right) \tag{6.8}$$

Require: Probability density function $f(x)$, a vector that holds indicator variables $(I_1, I_2, \ldots, I_N)$ which will explain the behavior of the density, a data table of the Polynomial Density Inversion method with the corresponding density.

Output: Random variate with density f.

1: **loop**
2: Generate $U \sim U(0,1)$.
3: Set $i \leftarrow g_{\lfloor U C \rfloor}$.
4: **while** $U > P_i$ **do**
5: Set $i \leftarrow i+1$.
6: $U = (U - P_{i-1})/(P_i - P_{i-1})$ /* recycling */
7: **if** $U \le rc_i$ **then**
8: $U \leftarrow U/rc_i$ /* second recycling */
9: $X \leftarrow b_{i-1} + U(b_i - b_{i-1})$
10: **return** X.
11: **else if** $U \le tr_i$ **then**
12: $U \leftarrow (U - rc_i)/(tr_i - rc_i)$ /* second recycling */
13: $X \leftarrow b_{i-1} + U(b_i - b_{i-1})$
14: Generate $X_2 \sim U(b_{i-1}, b_i)$.
15: **if** subinterval i increasing $(I_i = 1)$ **then**
16: **return** $\max(X, X_2)$.
17: **else**
18: **return** $\min(X, X_2)$.
19: **else**
20: $U \leftarrow (U - tr_i)/(1 - tr_i)$ /* second recycling */
21: **return** $P^{-1}(U \times a_{p_i})$.
/* use Equation 6.8 and Brent's Method */

Figure 6.8. Polynomial Density Inversion sampling algorithm

6.3. Computational Results and Approximation Performance

We have applied the Polynomial Density Inversion algorithm in C for a list of unimodal distributions given in Table 3.1. We used the necessary input (the mode, the cutoff and the inflection points) which is also given in Table 3.1. We have also coded a

function in R that evaluates the total table size that is used and the mean and the standard deviation for L_1 error for piecewise polynomial approximations. This code is shown in Appendix C.

6.3.1. Timing Results

For the Polynomial Density Inversion algorithm, the relative execution time (See Section 3.3.1) of the total algorithm, which includes both the setup and the sampling algorithms, is measured while 10^4 and 10^6 random variates are generated. For the subinterval creation parameters, linear error (LE), relative error (RE) and the polynomial error (PE), different values were used to find efficient settings for the algorithm. In order to evaluate relative average generation times, each generation was repeated 1000 times in an outer loop, and then the total execution time was divided by 1000.

Table 6.1. Relative average generation times for the Polynomial Density Inversion method with sample size $n = 10^4$

		LE	0.01			0.001			0.0001			0.00001		
		PE	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12
RE	0.1	N	1.75	2.30	5.16	1.67	2.22	5.07	2.16	2.34	4.67	4.01	4.01	4.73
		C	2.37	3.46	6.67	2.28	3.39	6.70	2.49	3.03	6.21	3.85	4.13	5.58
		E	1.66	1.79	3.04	1.51	1.72	2.97	1.79	1.87	2.78	3.21	3.24	3.60
		G	3.09	5.21	11.07	2.87	5.00	10.85	4.21	4.73	9.94	9.96	10.01	11.54
		B1	4.25	9.18	17.20	6.12	7.96	15.81	12.66	12.68	14.03	41.64	41.64	41.79
		B2	6.57	16.27	30.90	10.30	13.88	28.06	26.42	27.31	31.34	86.72	87.01	88.66
	0.01	N	2.41	2.72	5.16	2.37	2.69	5.12	2.84	2.88	4.78	4.58	4.58	4.90
		C	3.10	3.93	6.70	3.09	3.88	6.78	3.34	3.73	6.39	4.58	4.60	5.81
		E	1.73	2.97	2.96	1.60	1.79	2.91	1.90	1.99	2.73	3.30	3.30	3.60
		G	4.22	5.82	11.12	4.22	5.78	11.09	5.37	5.64	10.40	10.91	10.91	12.07
		B1	6.87	11.49	18.96	8.66	10.45	18.06	14.93	14.93	16.12	43.28	43.13	43.28
		B2	9.55	16.87	29.85	13.13	15.07	27.61	28.66	28.51	31.34	87.52	88.36	88.96

We can obtain the average total execution time of 10^4 and 10^6 variates in our test computer by multiplying relative average generation times respectively with 6.7×10^{-4} and 6.7×10^{-2}.

Table 6.2. Relative average generation times for the Polynomial Density Inversion method with sample size $n = 10^6$

		LE	0.01			0.001			0.0001			0.00001		
		PE	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12	1.E-08	1.E-10	1.E-12
RE	0.1	N	1.01	0.93	0.94	1.04	0.94	0.93	0.93	0.91	0.96	0.91	0.91	0.91
		C	1.22	0.97	0.94	1.16	0.97	0.97	1.09	1.02	0.97	1.00	0.96	0.94
		E	1.21	0.96	0.88	1.06	0.93	0.90	0.90	0.90	0.90	0.90	0.90	0.91
		G	0.99	0.94	0.99	0.96	0.96	0.99	0.94	0.94	0.99	0.99	0.97	0.97
		B1	1.00	0.97	1.03	0.96	0.96	1.03	0.99	1.00	1.01	1.28	1.28	1.28
		B2	1.03	1.04	1.18	0.99	1.00	1.15	1.13	1.15	1.18	1.81	1.81	1.82
	0.01	N	0.96	0.90	0.91	0.97	0.90	0.90	0.88	0.90	0.91	0.90	0.90	0.91
		C	1.09	0.96	0.96	1.12	0.99	0.96	0.99	0.96	0.94	0.94	0.94	0.94
		E	1.21	0.91	0.90	1.03	0.94	0.90	0.91	0.91	0.90	0.91	0.90	0.91
		G	0.99	0.94	0.97	0.97	0.93	0.98	0.94	0.93	0.97	1.00	0.94	0.97
		B1	0.99	0.97	1.06	0.96	0.96	1.04	1.01	1.00	1.03	1.30	1.30	1.33
		B2	1.00	1.04	1.16	1.00	1.03	1.15	1.16	1.16	1.18	1.81	1.81	1.82

6.3.2. Memory Occupation

The required table size can simply be evaluated while the number of subintervals is known. For each subinterval we are holding the $K+1$ coefficients of the approximation polynomial of order K, the cumulative probability of rectangular and triangular regions, the left boundary and the cumulative probability of the subinterval. Each of those values can be stored as a double variable, which occupies 8 bytes of memory.

A guide table of size $2 \times N$ should also be stored for the Indexed Search. Since the indices are integers, each element of the guide table can be stored as an integer variable, which occupies 4 bytes of memory.

Insignificantly, cumulative probabilities and the subinterval boundaries need an additional storage since they refer to boundaries instead of subintervals. A single integer variable can be used to store the total number of subintervals for quick calls. We have to include the vector of indicator variables as a table element, since in practice we prefer to generate variates with only one input which holds all the information we need. Yet, we can handle the indication by the mode for unimodal distributions as a double variable. Therefore, N subintervals will result in a table size of:

$$TS = \big((5+K)N+3\big)b_d + \big(2N+1\big)b_i = 8\times\big((5+K)N+2\big) + 4\times\big(2N+1\big) = \big(48+8K\big)N+20$$

which is evaluated in bytes. Table 6.3 below shows the number of subintervals and the total size of the data tables for each distribution with different flexible subinterval creation parameters.

Table 6.3. Number of subintervals and the total size of the setup tables of 4^{th} order polynomial density inversion for different parameters and distributions

		Critical RE:	0.1								0.01							
		Critical LE:	0.01		0.001		0.001		0.0001		0.01		0.001		0.001		0.0001	
Critical Polynomial Error	1.E-08	N(0,1)	80	6428	80	6428	166	13308	412	32988	188	15068	188	15068	260	20828	496	39708
		C(0,1)	162	12988	164	13148	232	18588	516	41308	356	28508	356	28508	420	33628	688	55060
		E(1)	48	3868	58	4668	122	9788	331	26508	71	5708	81	6508	143	11468	348	27860
		G(3,1)	62	4988	64	5148	124	9948	343	27468	121	9708	121	9708	169	13548	378	30260
		B(3,4)	59	4748	104	8348	236	18908	821	65708	120	9628	156	12508	279	22348	853	68260
		B(30,40)	99	7948	186	14908	505	40428	1722	137788	168	13468	245	19628	552	44188	1757	140580
	1.E-10	N(0,1)	138	11068	138	11068	182	14588	416	33308	218	17468	218	17468	262	20988	496	39700
		C(0,1)	304	24348	304	24348	310	24828	538	43068	458	36668	458	36668	464	37148	692	55380
		E(1)	89	7148	89	7148	132	10588	333	26668	104	8348	104	8348	147	11788	348	27860
		G(3,1)	121	9708	121	9708	138	11068	344	27548	162	12988	162	12988	176	14108	378	30260
		B(3,4)	127	10188	129	10348	237	18988	821	65708	181	14508	181	14508	280	22428	853	68260
		B(30,40)	229	18348	236	18908	519	41548	1726	138108	263	21068	270	21628	553	44268	1757	140580
	1.E-12	N(0,1)	378	30268	378	30268	378	30268	476	38108	422	33788	422	33788	422	33788	520	41620
		C(0,1)	674	53948	674	53948	674	53948	722	57788	786	62908	786	62908	786	62908	834	66740
		E(1)	201	16108	201	16108	211	16908	365	29228	207	16588	207	16588	217	17388	371	29700
		G(3,1)	268	21468	268	21468	268	21468	382	30588	296	23708	296	23708	296	23708	406	32500
		B(3,4)	235	18828	235	18828	254	20348	822	65788	281	22508	281	22508	297	23788	854	68340
		B(30,40)	425	34028	425	34028	573	45868	1748	139868	441	35308	441	35308	589	47148	1764	141140

According to the Table 6.3, if setup time is not important, for the critical linear error it is better to use the values 0.01 and 0.001 and for the critical relative linear error, it is better to use the value 0.1. Known that the main parameter to define an upper bound for the maximum absolute approximation error is the critical polynomial error (See Section 6.3.4), it is clear that, the small values of other parameters create unnecessary subintervals and increase the memory occupation and result in a slow setup.

6.3.3. Efficiency

For the Polynomial Density Inversion method, random variates can be sampled from three regions: rectangular, triangular and polynomial regions. Since sampling from the polynomial region is more expensive, the probability of sampling from the polynomial regions should be small.

Table 6.4, 6.5, 6.6 and 6.7 show the average percentages of sampling types for different flexible subinterval creation parameters. In order to compute those performance criteria, the Polynomial Density Inversion algorithm was run 500 times. In each run, the setup algorithm is executed and 10^6 random variates are generated. For the percentage of each type of sampling, the mean and the standard deviation were evaluated.

Table 6.4. Average percentage of sampling types for different distributions, critical linear error, $\varepsilon_c = 0.01$, and different critical relative (CRE) and polynomial error (CPE) values

CRE	CPE	Regions	N		C		E		G		B1		B2	
0.1	1.E-08	Rect. (%)	94.18	0.023	93.01	0.026	91.27	0.029	92.99	0.025	94.57	0.024	96.33	0.019
		Tri. (%)	5.52	0.022	6.37	0.025	8.14	0.027	6.52	0.024	5.23	0.023	3.54	0.019
		Poly. (%)	0.30	0.006	0.62	0.008	0.59	0.008	0.49	0.007	0.20	0.004	0.13	0.004
	1.E-10	Rect. (%)	97.14	0.016	97.37	0.015	96.31	0.019	97.34	0.017	98.14	0.014	98.58	0.011
		Tri. (%)	2.80	0.016	2.52	0.015	3.59	0.019	2.59	0.016	1.83	0.014	1.40	0.011
		Poly. (%)	0.06	0.002	0.11	0.003	0.10	0.101	0.07	0.003	0.03	0.002	0.02	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.29	0.013	98.91	0.010	99.07	0.009	99.24	0.009
		Tri. (%)	0.90	0.010	1.07	0.010	1.69	0.013	1.08	0.010	0.92	0.009	0.75	0.009
		Poly. (%)	0.01	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.01	0.001
0.01	1.E-08	Rect. (%)	94.61	0.023	94.30	0.023	91.36	0.028	94.21	0.025	94.79	0.023	96.44	0.019
		Tri. (%)	5.18	0.022	5.36	0.022	8.07	0.027	5.57	0.024	5.05	0.023	3.45	0.018
		Poly. (%)	0.21	0.004	0.34	0.006	0.57	0.007	0.22	0.005	0.16	0.004	0.11	0.003
	1.E-10	Rect. (%)	97.14	0.016	97.46	0.016	96.31	0.019	97.35	0.017	98.17	0.014	98.58	0.011
		Tri. (%)	2.80	0.016	2.45	0.015	3.59	0.019	2.58	0.016	1.80	0.013	1.40	0.011
		Poly. (%)	0.06	0.002	0.09	0.003	0.10	0.003	0.07	0.003	0.03	0.002	0.02	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.29	0.013	98.91	0.010	99.07	0.009	99.24	0.009
		Tri. (%)	0.90	0.010	1.07	0.010	1.69	0.013	1.08	0.010	0.92	0.009	0.75	0.009
		Poly. (%)	0.001	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.01	0.001

Table 6.5. Average percentage of sampling types for different distributions, critical linear error, $\varepsilon_c = 0.001$, and different critical relative (CRE) and polynomial error (CPE) values

CRE	CPE	Regions	N		C		E		G		B1		B2	
0.1	1.E-08	Rect. (%)	94.18	0.023	93.33	0.025	94.43	0.023	93.51	0.026	96.80	0.018	98.26	0.013
		Tri. (%)	5.52	0.022	6.10	0.024	5.28	0.023	6.14	0.025	3.15	0.018	1.71	0.013
		Poly. (%)	0.30	0.006	0.57	0.008	0.29	0.005	0.35	0.006	0.05	0.002	0.03	0.002
	1.E-10	Rect. (%)	97.14	0.016	97.37	0.015	96.31	0.020	97.34	0.017	98.15	0.014	98.68	0.011
		Tri. (%)	2.80	0.016	2.52	0.015	3.59	0.020	2.59	0.016	1.82	0.014	1.31	0.011
		Poly. (%)	0.06	0.002	0.11	0.003	0.10	0.003	0.07	0.003	0.03	0.002	0.01	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.29	0.013	98.91	0.010	99.07	0.009	99.24	0.009
		Tri. (%)	0.90	0.010	1.07	0.010	1.69	0.013	1.08	0.010	0.92	0.009	0.75	0.009
		Poly. (%)	0.01	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.01	0.001
0.01	1.E-08	Rect. (%)	94.61	0.023	94.30	0.023	94.53	0.022	94.39	0.024	96.82	0.018	98.28	0.013
		Tri. (%)	5.18	0.022	5.36	0.022	5.21	0.022	5.41	0.024	3.14	0.018	1.70	0.013
		Poly. (%)	0.21	0.004	0.34	0.006	0.26	0.005	0.20	0.004	0.04	0.002	0.02	0.001
	1.E-10	Rect. (%)	97.14	0.016	97.46	0.016	96.31	0.019	97.35	0.017	98.17	0.014	98.68	0.011
		Tri. (%)	2.80	0.016	2.45	0.015	3.59	0.019	2.58	0.016	1.80	0.013	1.31	0.011
		Poly. (%)	0.06	0.002	0.09	0.003	0.10	0.003	0.07	0.003	0.03	0.002	0.01	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.29	0.013	98.91	0.010	99.07	0.009	99.24	0.009
		Tri. (%)	0.90	0.010	1.07	0.010	1.69	0.013	1.08	0.010	0.92	0.009	0.75	0.009
		Poly. (%)	0.01	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.01	0.001

Table 6.6. Average percentage of sampling types for different distributions, critical linear error, $\varepsilon_c = 0.0001$, and different critical relative (CRE) and polynomial error (CPE) values

CRE	CPE	Regions	N		C		E		G		B1		B2	
0.1	1.E-08	Rect. (%)	97.67	0.015	96.13	0.018	98.04	0.014	97.16	0.017	98.87	0.011	99.34	0.008
		Tri. (%)	2.29	0.015	3.56	0.017	1.91	0.014	2.77	0.016	1.12	0.011	0.66	0.008
		Poly. (%)	0.04	0.002	0.31	0.006	0.05	0.002	0.07	0.003	0.01	0.001	0.00	0.001
	1.E-10	Rect. (%)	97.89	0.014	97.45	0.015	98.11	0.014	97.65	0.015	98.91	0.010	99.36	0.008
		Tri. (%)	2.08	0.014	2.44	0.015	1.85	0.014	2.30	0.015	1.08	0.010	0.64	0.008
		Poly. (%)	0.03	0.002	0.11	0.003	0.04	0.002	0.05	0.002	0.01	0.001	0.00	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.52	0.012	98.91	0.010	99.12	0.009	99.44	0.007
		Tri. (%)	0.90	0.010	1.07	0.010	1.46	0.012	1.08	0.010	0.87	0.009	0.56	0.007
		Poly. (%)	0.01	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.00	0.001
0.01	1.E-08	Rect. (%)	97.71	0.015	96.76	0.017	98.07	0.014	97.23	0.017	98.87	0.011	99.34	0.008
		Tri. (%)	2.26	0.015	3.10	0.017	1.89	0.014	2.72	0.016	1.12	0.011	0.66	0.008
		Poly. (%)	0.03	0.002	0.15	0.004	0.04	0.002	0.05	0.003	0.01	0.001	0.00	0.001
	1.E-10	Rect. (%)	97.89	0.014	97.54	0.016	98.11	0.014	97.65	0.015	98.91	0.010	99.36	0.008
		Tri. (%)	2.08	0.014	2.38	0.015	1.85	0.014	2.30	0.015	1.08	0.010	0.64	0.008
		Poly. (%)	0.03	0.002	0.08	0.003	0.04	0.002	0.05	0.002	0.01	0.001	0.00	0.001
	1.E-12	Rect. (%)	99.09	0.010	98.91	0.010	98.52	0.012	98.91	0.010	99.12	0.009	99.44	0.007
		Tri. (%)	0.90	0.010	1.07	0.010	1.46	0.012	1.08	0.010	0.87	0.009	0.56	0.007
		Poly. (%)	0.01	0.001	0.02	0.001	0.02	0.001	0.01	0.001	0.01	0.001	0.00	0.001

Table 6.7. Average percentage of sampling types for different distributions, critical linear error, $\varepsilon_c = 0.00001$, and different critical relative (CRE) and polynomial error (CPE) values

CRE	CPE	Regions	N		C		E		G		B1		B2	
0.1	1.E-08	Rect. (%)	99.12	0.009	98.35	0.013	99.35	0.008	99.04	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.87	0.009	1.54	0.012	0.64	0.008	0.95	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.01	0.001	0.11	0.004	0.01	0.001	0.01	0.001	0.00	0.000	0.00	0.000
	1.E-10	Rect. (%)	99.12	0.009	98.54	0.012	99.35	0.008	99.04	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.87	0.009	1.40	0.012	0.64	0.008	0.95	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.01	0.001	0.06	0.002	0.01	0.001	0.01	0.001	0.00	0.000	0.00	0.000
	1.E-12	Rect. (%)	99.28	0.009	98.99	0.010	99.37	0.008	99.23	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.72	0.009	0.99	0.010	0.62	0.008	0.77	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.00	0.001	0.02	0.001	0.01	0.001	0.00	0.001	0.00	0.000	0.00	0.000
0.01	1.E-08	Rect. (%)	99.12	0.009	98.59	0.012	99.35	0.008	99.05	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.87	0.009	1.36	0.011	0.64	0.008	0.94	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.01	0.001	0.05	0.002	0.01	0.001	0.01	0.001	0.00	0.000	0.00	0.000
	1.E-10	Rect. (%)	99.12	0.009	98.62	0.011	99.35	0.008	99.05	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.87	0.009	1.34	0.011	0.64	0.008	0.94	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.01	0.001	0.04	0.002	0.01	0.001	0.01	0.001	0.00	0.000	0.00	0.000
	1.E-12	Rect. (%)	99.28	0.009	99.00	0.010	99.37	0.008	99.23	0.009	99.65	0.006	99.80	0.004
		Tri. (%)	0.72	0.009	0.98	0.010	0.62	0.008	0.77	0.009	0.35	0.006	0.20	0.004
		Poly. (%)	0.00	0.001	0.02	0.001	0.01	0.001	0.00	0.001	0.00	0.000	0.00	0.000

6.3.4. Approximation Performance of Heuristic Subinterval Creation Method

In Section 5.1, it was suggested that the degree of the approximation polynomial should be chosen around 5. We have observed how the error behaves over the domain for normal distribution and evaluated L_1 error for different distributions under different flexible subinterval creation parameters.

For different values of critical linear, relative linear and polynomial error parameters, different subinterval-decompositions were obtained. As the number of subintervals increases, of course the size of the setup table also increases. However, both L_1 error and the maximum approximation error get smaller. Also, sampling from the polynomial region has a small probability.

The Figure 6.9 and 6.10 show, how the absolute approximation error of a fourth order interpolation is distributed for the standard normal distribution with different critical error parameters. They show the absolute approximation error values for 12001 equidistant

points between cutoff points $\{-6,6\}$. It can be easily seen that the maximum absolute approximation error is always smaller than the critical polynomial error ε_c' and, in general, larger in the center than in the tails.

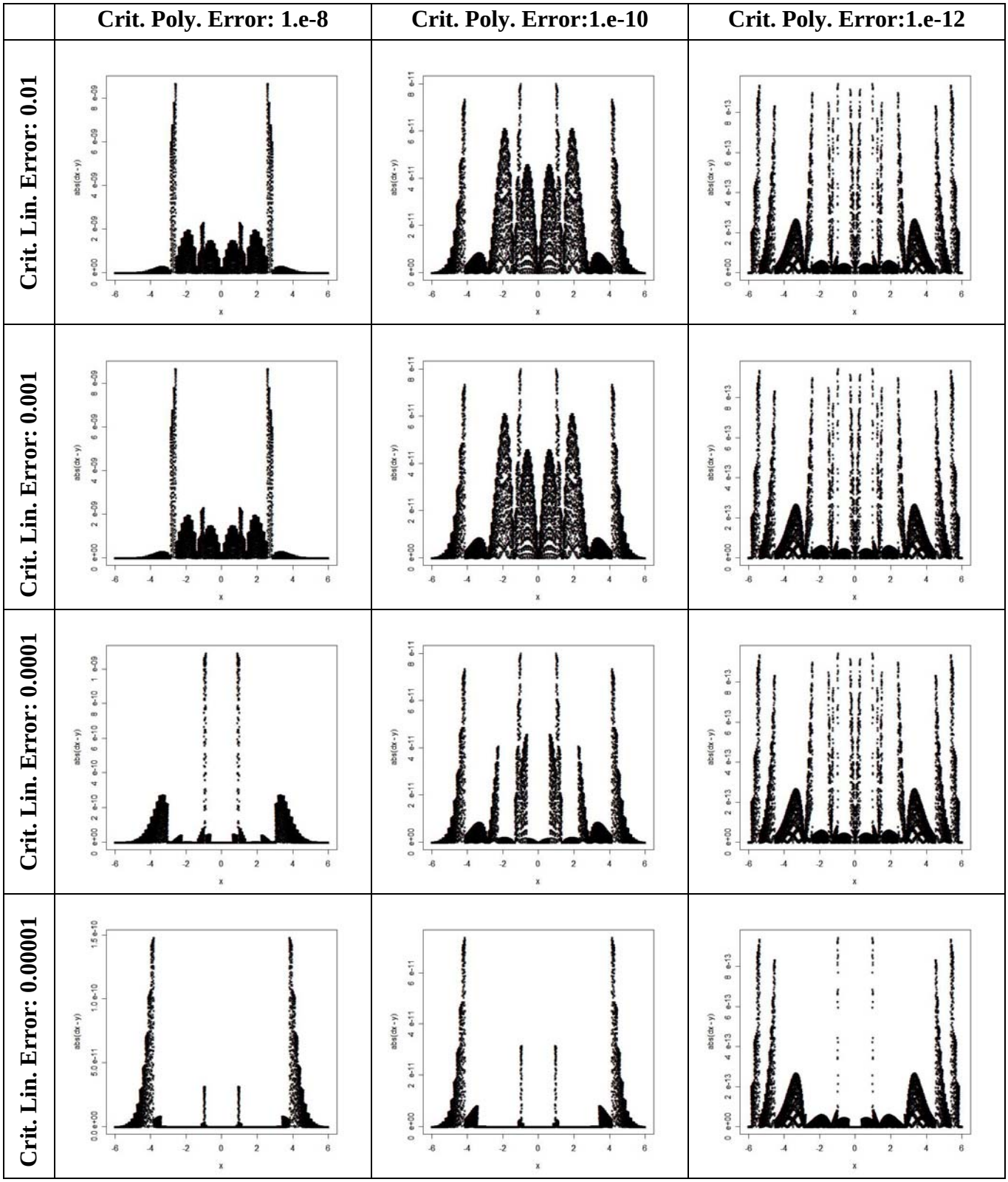


Figure 6.9. Distribution of absolute approximation error of a fourth order interpolation for the standard normal distribution with critical relative error, $r_c = 0.1$, and different critical linear and polynomial error values

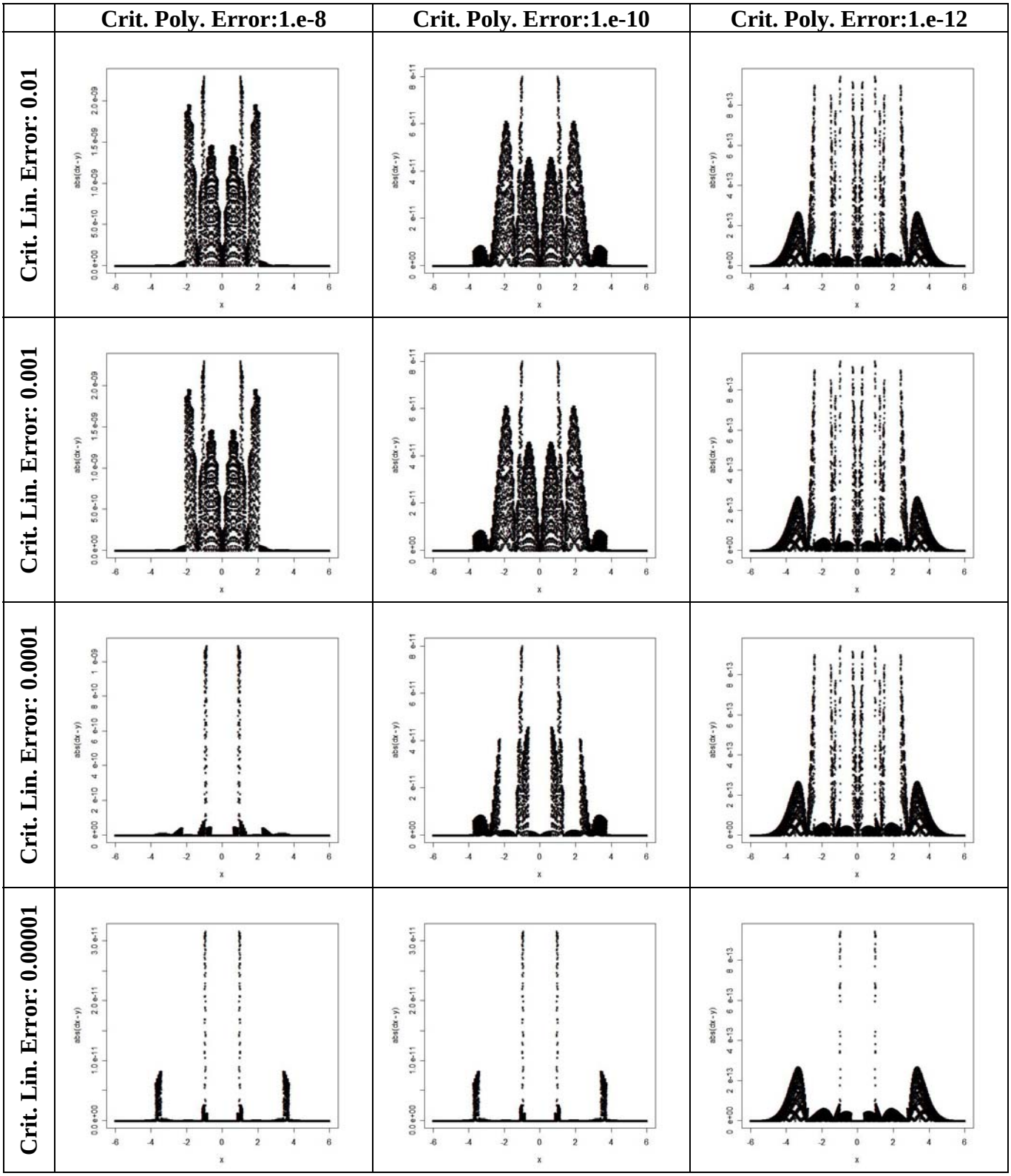


Figure 6.10. Distribution of absolute approximation error of a fourth order interpolation for the standard normal distribution with critical relative error, $r_c = 0.01$, and different critical linear and polynomial error values

Looking at the graphs, it is now possible to analyze the complicated construction of the flexible subinterval creation heuristics. Clearly the main critical error parameter which defines the maximum absolute approximation error is the critical polynomial error parameter itself. The critical relative error is the parameter which decreases the absolute approximation error in the tails. On the other hand, the critical linear error is the parameter which decreases the average absolute approximation error, when the critical polynomial error is fixed.

Also, the critical linear error is not so effective in creating subintervals while the critical polynomial error takes smaller values. As it is seen in the graphs, while the critical polynomial error is 10^{-12}, the critical linear error can only change the average absolute approximation error when it takes the value of 0.00001.

Tables 6.8, 6.9, 6.10 and 6.11 show the mean and the standard deviation of the simulation results in order to evaluate the L_1 error for different distributions. To evaluate those values, importance sampling (See Section 6.1.3) has been used. For importance sampling densities, the original distribution, and the uniform distribution between the cutoff points are used. 10000 random variates have been used for each run.

Table 6.8. Simulation results for evaluating L_1 error with critical linear error, $\varepsilon_c = 0.01$, and different critical relative and polynomial error values

Critical Polynomial Error			Critical Relative Error=0.1				Critical Realtive Error=0.01			
			Original IS Dens.		Uniform IS Dens.		Original IS Dens.		Uniform IS Dens.	
			Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1
	1.E-08	N	7.71E-09	5.36E-08	7.36E-09	1.01E-10	3.52E-09	4.94E-09	3.46E-09	4.25E-11
		C	2.98E-07	2.06E-06	1.82E-07	9.15E-06	2.26E-08	3.61E-08	2.21E-09	1.44E-07
		E	1.12E-08	5.81E-08	1.13E-08	2.27E-08	6.53E-09	9.66E-09	6.63E-09	1.92E-08
		G	1.48E-08	2.80E-08	1.51E-08	2.89E-08	8.19E-09	1.06E-07	8.27E-09	2.37E-08
		B1	1.46E-09	1.10E-09	1.45E-09	1.42E-09	1.40E-09	8.60E-10	1.38E-09	1.47E-09
		B2	7.30E-10	2.44E-09	8.00E-10	1.16E-09	6.80E-10	1.88E-09	6.58E-10	1.22E-09
	1.E-10	N	1.84E-10	4.99E-09	1.79E-10	1.43E-12	1.41E-10	2.89E-10	1.41E-10	1.36E-12
		C	2.79E-08	6.30E-07	4.44E-08	1.52E-06	1.95E-09	1.14E-08	4.41E-10	3.11E-08
		E	2.38E-10	9.90E-09	2.06E-10	3.10E-10	1.63E-10	1.40E-09	1.48E-10	2.71E-10
		G	3.21E-10	7.22E-09	2.29E-10	3.07E-10	1.57E-10	4.46E-10	1.56E-10	2.64E-10
		B1	7.57E-12	5.56E-11	7.44E-12	7.60E-12	6.29E-12	2.03E-11	6.52E-12	7.43E-12
		B2	6.42E-12	6.37E-11	8.78E-12	9.94E-12	6.73E-12	5.90E-11	7.17E-12	9.23E-12
	1.E-12	N	1.02E-12	7.34E-12	1.71E-12	1.57E-14	1.16E-12	1.38E-11	1.09E-12	1.27E-14
		C	5.18E-09	2.88E-07	5.32E-09	9.12E-08	4.78E-10	5.48E-09	1.47E-10	4.84E-09
		E	4.05E-12	2.63E-10	2.85E-12	3.11E-12	1.60E-12	1.82E-11	2.28E-12	3.01E-12
		G	2.75E-12	1.40E-11	4.13E-12	4.25E-12	2.51E-12	1.27E-11	3.44E-12	4.06E-12
		B1	2.94E-13	6.64E-12	2.47E-13	2.42E-13	2.25E-13	1.14E-12	2.29E-13	2.39E-13
		B2	4.54E-14	1.83E-12	6.21E-13	6.46E-13	6.19E-13	5.92E-12	5.39E-13	6.31E-13

Table 6.9. Simulation results for evaluating L_1 error with critical linear error, $\varepsilon_c = 0.001$, and different critical relative and polynomial error values

Critical Polynomial Error			Critical Relative Error=0.1				Critical Realtive Error=0.01			
			Original IS Dens.		Uniform IS Dens.		Original IS Dens.		Uniform IS Dens.	
			Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1
	1.E-08	N	7.52E-09	4.26E-08	7.23E-09	1.00E-10	3.38E-09	4.75E-09	3.42E-09	4.24E-11
		C	2.74E-07	1.89E-06	4.97E-07	3.24E-05	2.21E-08	3.68E-08	4.34E-10	1.88E-08
		E	6.68E-09	5.60E-08	6.75E-09	1.47E-08	2.38E-09	6.47E-09	2.29E-09	6.44E-09
		G	1.27E-08	2.46E-08	1.24E-08	2.66E-08	7.18E-09	1.06E-08	7.22E-09	2.21E-08
		B1	2.26E-10	6.55E-10	2.29E-10	6.80E-10	2.51E-10	7.01E-10	2.34E-10	7.02E-10
		B2	7.55E-11	2.00E-09	1.14E-10	1.87E-10	2.55E-11	1.34E-10	2.47E-11	5.84E-11
	1.E-10	N	1.38E-10	2.83E-10	1.77E-10	1.40E-12	1.42E-10	3.53E-10	1.40E-10	2,00E+00
		C	2.18E-08	3.81E-07	2.63E-08	7.31E-07	1.80E-09	1.03E-08	1.27E-10	3.94E-09
		E	1.86E-10	3.91E-09	2.14E-10	3.13E-10	1.44E-10	1.18E-09	1.44E-10	2.68E-10
		G	1.88E-10	3.68E-09	2.25E-10	3.03E-10	1.67E-10	5.04E-10	1.58E-10	2.66E-10
		B1	7.95E-12	3.78E-11	7.36E-12	7.70E-12	6,47E-12	2.15E-11	6.44E-12	7.37E-12
		B2	5.58E-12	5.25E-11	8.14E-12	9.29E-12	7.28E-12	1.43E-10	6.45E-12	8.44E-12
	1.E-12	N	9.83E-13	5.76E-12	1.70E-12	1.57E-14	1.06E-12	1.24E-11	1.09E-12	1.27E-14
		C	5.55E-09	2.07E-07	5.58E-09	7.32E-08	4.93E-10	5.40E-09	1.05E-09	4.77E-08
		E	2.68E-12	1.03E-10	2.80E-12	3.10E-12	5.78E-12	3.02E-10	2.33E-12	3.06E-12
		G	2.86E-12	1.62E-11	4.23E-12	4.25E-12	3.58E-12	9.09E-11	3.48E-12	2.66E-10
		B1	2.30E-13	1.05E-12	2.47E-13	2.43E-13	2.20E-13	9.32E-13	2.29E-13	2.37E-13
		B2	4.98E-13	3.04E-12	6.22E-13	6.49E-13	5.24E-13	4.21E-12	5.27E-13	6.16E-13

Table 6.10. Simulation results for evaluating L_1 error with critical linear error, $\varepsilon_c = 0.0001$, and different critical relative and polynomial error values

			Critical Relative Error=0.1				Critical Realtive Error=0.01			
			Original IS Dens.		Uniform IS Dens.		Original IS Dens.		Uniform IS Dens.	
			Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1
Critical Polynomial Error	1.E-08	N	4.40E-10	7.62E-09	5.65E-10	1.02E-11	2.19E-10	7.38E-10	2.19E-10	8.75E-12
		C	2.84E-07	2.09E-06	1.52E-08	5.43E-07	7.63E-09	2.71E-08	7.81E-09	7.58E-07
		E	1.36E-09	2.85E-08	1.73E-09	4.66E-09	2.29E-10	2.10E-09	2.18E-10	6.03E-10
		G	1.52E-09	1.49E-08	1.34E-09	3.32E-09	4.42E-10	1.31E-09	4.38E-10	2.44E-09
		B1	2.44E-12	1.18E-11	2.48E-12	1.22E-11	2.31E-12	1.10E-11	2.10E-12	1.05E-11
		B2	1.47E-12	1.93E-11	2.96E-11	7.67E-11	2.92E-12	7.13E-11	2.79E-12	1.17E-11
	1.E-10	N	6.84E-11	3.76-E10	1.04E-10	1.17E-12	6.43E-11	1.93E-10	6.33E-11	9.48E-13
		C	3.38E-08	6.98E-07	3.08E-08	8.02E-07	1.95E-09	1.09E-08	1.09E-09	5.77E-08
		E	9.20E-11	1.23E-09	1.61E-10	2.75E-10	7.47E-10	1.00E-10	9.26E-11	2.10E-10
		G	1.73E-10	5.28E-09	1.68E-10	2.30E-10	1.04E-10	4.88E-10	1.02E-10	1.59E-10
		B1	1.15E-12	5.19E-12	1.12E-12	3.37E-12	1.08E-12	3.60E-12	1.15E-12	3.54E-12
		B2	8.70E-13	2.78E-11	3.78E-12	7.41E-12	3.51E-12	1.47E-10	2.12E-12	4.79E-12
	1.E-12	N	1.20E-12	1.28E-11	1.73E-12	1.59E-14	1.17E-12	1.03E-11	1.07E-12	1.26E-14
		C	2.35E-09	5.21E-08	4.25E-09	6.23E-08	4.54E-10	4.93E-09	5.05E-10	2.77E-08
		E	1.48E-12	1.98E-11	2.67E-12	3.00E-12	4.11E-12	2.62E-10	2.17E-12	2.94E-12
		G	2.64E-12	1.37E-11	4.09E-12	4.20E-12	2.81E-12	1.39E-11	3.35E-12	4.00E-12
		B1	2.32E-13	2.19E-12	2.27E-13	2.51E-13	2.38E-13	1.73E-12	2.13E-13	2.47E-13
		B2	3.73E-13	4.66E-12	4.60E-13	5.11E-13	3.47E-13	2.89E-12	3.75E-13	4.37E-13

Table 6.11. Simulation results for evaluating L_1 error with critical linear error, $\varepsilon_c = 0.00001$, and different critical relative and polynomial error values

			Critical Relative Error=0.1				Critical Realtive Error=0.01			
			Original IS Dens.		Uniform IS Dens.		Original IS Dens.		Uniform IS Dens.	
			Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1	Mean L_1	Std. D. L_1
Critical Polynomial Error	1.E-08	N	5.12E-11	4..45E-09	9.11E-11	1.74E-12	4.98E-12	1.09E-10	5.96E-12	2.06E-13
		C	1.37E-07	1.49E-06	2.01E-07	8.91E-06	2.26E-09	1.61E-08	1.34E-08	1.33E-06
		E	1.86E-10	9.94E-09	2.09E-10	6.01E-10	1.69E-11	5.44E-10	2.07E-11	5.60E-11
		G	3.30E-11	1.36E-09	1.26E-10	3.22E-10	2.18E-11	3.50E-10	1.91E-11	6.31E-11
		B1	3.53E-14	2.62E-13	3.83E-14	2.59E-13	3.58E-14	2.41E-13	3.87E-14	2.50E-13
		B2	4.24E-12	3.92E-10	4.69E-12	1.58E-11	2.86E-13	8.78E-13	3.89E-13	8.05E-13
	1.E-10	N	9.99E-11	9.39E-09	4.63E-11	8.74E-13	7.68E-12	1.86E-10	5.96E-12	2.06E-13
		C	2.73E-08	5.89E-07	3.42E-08	1.23E-06	1.51E-09	1.12E-08	3.78E-10	1.86E-08
		E	2.38E-11	6.48E-10	8.55E-11	2.01E-10	1.26E-11	4.48E-10	1.95E-11	5.31E-11
		G	2.43E-10	9.46E-09	8.88E-11	2.15E-10	1.74E-11	2.81E-10	1.91E-11	6.50E-11
		B1	3.90E-14	6.11E-13	4.12E-14	2.68E-13	3.50E-14	2.50E-13	4.22E-14	2.62E-13
		B2	2.87E-13	1.58E-12	1.84E-12	5.76E-12	2.73E-13	9.35E-13	3.70E-13	7.42E-13
	1.E-12	N	4.43E-13	8.56E-12	1.14E-12	1.26E-14	4.06E-13	8.41E-12	4.92E-13	5.74E-15
		C	2.35E-09	6.37E-08	6.02E-09	8.32E-08	4.47E-10	5.12E-09	1.87E-10	4.84E-09
		E	1.11E-12	2.29E-11	2.05E-12	2.77E-12	7.92E-13	1.56E-11	1.53E-12	2.52E-12
		G	2.78E-12	1.57E-10	2.78E-12	3.75E-12	2.99E-12	1.24E-10	2.05E-12	3.28E-12
		B1	1.67E-14	1.12E-13	2.02E-14	6.79E-14	3.74E-14	1.75E-12	2.00E-14	6.47E-14
		B2	3.68E-13	6.13E-12	4.12E-13	5.35E-13	5.11E-13	1.72E-11	3.15E-13	4.40E-13

The tables show that the L_1 error is small enough to speak of a very good approximation. An accurate result of the L_1 error is always obtained when the uniform distribution between the cutoff points is used as importance sampling density.

As an approximate random variate generation method, the polynomial approximation can reach very small approximation errors with less subintervals than piecewise constant or piecewise linear approximation of the density. Therefore, the algorithm requires smaller tables. Also, the sampling algorithm does not require any density function calls.

The algorithm has the advantage of a faster sampling than the Triangular Ahrens Method. This is because of the reasons:

- The Triangular Ahrens needs at least two uniform random variates to generate a random variate.
- There is a rejection probability.
- It needs to call density function in the sampling algorithm.

On the other hand, the Polynomial Density Inversion is based only on decomposition-inversion and needs only one uniform random number. It never calls the density function in the sampling algorithm. It is also guaranteed to generate a random variate in each time the sampling algorithm is run. Yet, it needs a larger data table to be stored, especially for higher order approximations.

Nevertheless, the setup algorithm is slower than the setup algorithm of the Triangular Ahrens and more complicated. Therefore, the Polynomial Density Inversion method can be an appropriate method when a larger number of random variates from a fixed distribution are needed.

7. CONCLUSIONS

Both of the Triangular Ahrens and the Polynomial Density Inversion algorithms were coded in “C”. Numerical results show both of the algorithms work well for a particular list of unimodal distributions and have advantages when compared to existing universal random variate generation methods (See Table 7.1). Both of the methods are not so easy to analyze. Due to the miscellaneous functions and sub-functions (See Appendix A and B) used both in the setup and sampling algorithms, it was hard to code the algorithms in a comprehensible and elegant way. On the other hand, defining general heuristics for the decomposition of the density (especially in the Polynomial Density Inversion method) was a long study with many trials of implementations. However, we believe that those heuristics are sufficient to obtain a decomposition, which is close to the optimal, for a user who understands the heuristics and can decide about appropriate critical parameters.

Table 7.1. Comparison of universal random variate generation methods

Performance Criteria	Ahrens Method	Transformed Density Rejection	Polynomial Inversion	Triangular Ahrens	Polynomial Density Inversion
Simplicity	**X**				
Small table		**X**		**X**	**X**
No function calls in the sampling			**X**		**X**
Speed	**X**	**X**	**X**	**X**	**X**

After the implementation, by applying the chi-square test, we saw that the generated sample follows the original distribution and the algorithms generate as fast as existing universal random variate generation methods. When we get into the details, we criticize and compare the characteristics of this new two methods and Ahrens, Transformed Density Rejection and Polynomial Inversion.

The Triangular Ahrens is the modification of the Ahrens Method with linear hats and the mirroring principle. Adding mirroring principle decreases the rejection constant. But different from the rejection with inversion, it generates random points without calling the hat function, like it is in the Ahrens Method. It needs a smaller table and has an acceptable marginal execution time when compared with the Ahrens Method. As it is seen

in Table 7.1, the Triangular Ahrens Method is similar to Transformed Density Rejection with respect to performance criteria.

The Polynomial Density Inversion uses piecewise polynomial approximations instead of the original density. Thus, it does not need any density function calls in the sampling algorithm. Since it also uses decomposition, it has a low probability of using polynomial inversion, which is relatively slow due to the expensive root finding algorithms. It achieves a good approximation of unimodal densities with 4^{th} order polynomials and result in a table size of 50 kilobytes at most. Compared with the Triangular Ahrens, it needs a longer and more complicated setup, but has a simpler sampling algorithm and a faster marginal execution time. The method is, then, preferable when it is needed to generate a very large number of random variates. It is similar to polynomial inversion with respect to performance criteria, only a bit faster and with smaller tables.

We have explained both of the algorithms with explicit pseudo codes which could be applied within any programming environment. We also added the “C” codes for both of the algorithms and “R” codes for evaluating L_1 error for constant, linear and the 4^{th} order approximations of the densities in the appendix.

APPENDIX A: TRIANGULAR AHRENS C CODES

Following codes are the c application of the Triangular Ahrens algorithm. The algorithm is coded as a main function. Also there is a header file that includes the setup, the sampling and all other helping functions.

Main Function:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "trilib.h"
/*1.000.000.000 repeat*/

#define OUTER 1         /*outer repetition increase the accuracy of time
length, repeats the setup and generation*/
#define RVS 1000000
/*number of random variates to be generated after setup*/
#define CLASSES 1000
/*number of classes for chi-square test, sqrt(RVS) is desirable*/
#define PRINTFLAG 1
/*prints out the setup table for each outer loop*/
#define CHI2RESULT 1
/*makes a chi-square test for random variates in each outer loop*/
#define COUNTFLAG 1     /*counts and prints the random variates according
to the generation types*/

#define ACRIT 0.05
/*heuristic subinterval creation parameter, area between error*/
#define DENSITY Normal        /*distribution type to be generated*/
#define DERIVATIVE Der1Normal /*first derivative of the distribution*/
#define CUMULATIVE CdfNormal  /*cumulative distribution for chi-square
test*/
#define P1 0      /*parameter 1*/
#define P2 1      /*parameter 2*/
#define LEFTCUTOFF -6
/*NORMAL(0,1) -6*/                  /*CAUCHY(0,1) -640000*/
/*EXPO(1,NULL) 0*/                  /*GAMMA(3,1) 0*/
/*BETA(3,4) 0*/                     /*BETA(30,40) 0*/
#define LEFTINFLECTION -1
/*NORMAL(0,1) -1*/                  /*CAUCHY(0,1) -0.577350269189625*/
/*EXPO(1,NULL) 0*/                  /*GAMMA(3,1) 0.585786437626905*/
/*BETA(3,4) 0.15505102572168*/      /*BETA(30,40) 0.366049994532764*/
#define MODE 0
/*NORMAL(0,1) 0*/                   /*CAUCHY(0,1) 0*/
/*EXPO(1,NULL) 0*/                  /*GAMMA(3,1) 2*/
/*BETA(3,4 0.4)*/                   /*BETA(30,40) 0.426470588235294*/
#define RIGHTINFLECTION 1
/*NORMAL(0,1) 1*/                   /*CAUCHY(0,1) 0.577350269189625*/
/*EXPO(1,NULL) 0*/                  /*GAMMA(3,1) 3.414213562373095*/
```

```
/*BETA(3,4) 0.6449489742783*/      /*BETA(30,40) 0.486891181937824*/
#define RIGHTCUTOFF 6
/*NORMAL(0,1) 6*/                  /*CAUCHY(0,1) 640000*/
/*EXPO(1,NULL) 17*/                /*GAMMA(3,1) 21*/
/*BETA(3,4) 1*/                    /*BETA(30,40) 0.8*/

int main()
{
      int anykey;

      int j=0,i,subints,total;
      time_t genstart,genend;
      double gendif,rv;            /*variables that hold timing result
and generated RVs*/
      double typevec[5]={0};       /*a vector that holds the average
percentage of sampling types*/
      double sumsquare[5]={0};     /*a vector that holds the std.
deviation of the percentage of sampling types*/
      double typemat[5][OUTER];    /*a matrix that holds the percentage
of sampling types for each run*/

      time (&genstart); /*this is where the timing starts*/
      double sum=0;     /*an arbitrary operation variable to be used with
each random variate*/
      for(j=0;j<OUTER;j++){
            /*SETUP PART*/
            LPAGEN *lpagen;   /*a struct that will store the table*/
            lpagen=Setup(LEFTCUTOFF,LEFTINFLECTION,MODE,RIGHTINFLECTION,
                        RIGHTCUTOFF,ACRIT,DENSITY,DERIVATIVE,P1,P2);
            if (PRINTFLAG!=0){
                  PrintLPATable(lpagen);
                  /*command that prints out the table*/
            }
            subints=lpagen->n;
            /*SETUP ENDS*/
            /*SAMPLING PART*/
            int counter[5]={0};    /*count vector to count three types
of generation and recursion*/
            long freq[CLASSES]={0}; /*frequency vector that will store
F(rv) in #CLASSES equidistant intervals*/
            for(i=0;i<RVS;i++){
                  rv=Generator(lpagen,DENSITY,P1,P2,counter,COUNTFLAG);
                  sum+=rv;    /*an arbitrary insignificant (not
expensive) operation*/
                  if (CHI2RESULT!=0){
                        freq[(int)(CLASSES*CUMULATIVE(rv,P1,P2))]++;
                  }
            }
            total=counter[0]+counter[1]+counter[2]+counter[3];
            typevec[0]+=typemat[0][j]=(double)counter[0]*100/total;
      /*assigments for efficiency characteristics*/
            typevec[1]+=typemat[1][j]=(double)counter[1]*100/total;
      /*assigments for efficiency characteristics*/
            typevec[2]+=typemat[2][j]=(double)counter[2]*100/total;
      /*assigments for efficiency characteristics*/
            typevec[3]+=typemat[3][j]=(double)counter[3]*100/total;
      /*assigments for efficiency characteristics*/
            typevec[4]+=typemat[4][j]=(double)counter[4]*100/total;
      /*assigments for efficiency characteristics*/
            /*SAMPLING ENDS*/
```

```
            if (CHI2RESULT!=0){
                  Chi2Test(freq,CLASSES,RVS,CHI2RESULT);
            }
            if (COUNTFLAG!=0){     /*prints out number of random
variates generated with three types of generation*/
                  printf("\nTotal # of RVs                    : %d",
                        total);
                  printf("\n# of RVs immediatly accepted      : %d"
                        "\n# of RVs accepted through squeeze : %d"
                        "\n# of RVs accepted through density : %d"
                        "\n# of RVs rejected                 : %d"
                        "\n# of mirroring applied RVs        : %d\n",
                        counter[0],counter[1],counter[2],counter[3],
                        counter[4]);
            }
            FreeGen(lpagen);  /*frees the memory*/
      }
      printf("\nMean: %f\n",sum/(OUTER*RVS));  /*the result of the
arbitrary operations*/
      time (&genend);   /*this is where the timing ends*/

      gendif=difftime(genend,genstart);

      for (i=0;i<=4;i++){
            for(j=0;j<OUTER;j++){
                  sumsquare[i]+=(typemat[i][j]-
(typevec[i]/OUTER))*(typemat[i][j]-(typevec[i]/OUTER));
            }
      }

      printf("\nMEMORY OCCUPATION:");    /*prints out the number of
subintervals in the table*/
      printf("\n# of subintervals: %d\n",subints);
      printf("\nTIMING RESULTS:");
      printf("\nProcessing time : %f",gendif); /*prints out the whole
execution time*/
      printf("\nSingle loop time: %f\n",(double)gendif/OUTER);   /*prints
out the average of single loop (setup and generation) times*/
      printf("\nPERFORMANCE CHARACTERISTICS:");      /*prints out the
mean and the std. dev. of the percentage of sampling types*/
      printf("\nAverage percentage of RVs immediately accepted
: %f",typevec[0]/OUTER);
      printf("\nStd.Dev. of the percentage of RVs immediately accepted
: %f",sqrt(sumsquare[0]/OUTER));
      printf("\nAverage percentage of RVs accepted through squeeze
: %f",typevec[1]/OUTER);
      printf("\nStd.Dev. of the percentage of RVs accepted through
squeeze: %f",sqrt(sumsquare[1]/OUTER));
      printf("\nAverage percentage of RVs accepted through density
: %f",typevec[2]/OUTER);
      printf("\nStd.Dev. of the percentage of RVs accepted through
density: %f",sqrt(sumsquare[2]/OUTER));
      printf("\nAverage percentage of RVs rejected
: %f",typevec[3]/OUTER);
      printf("\nStd.Dev. of the percentage of RVs rejected
: %f",sqrt(sumsquare[3]/OUTER));
      printf("\nAverage percentage of mirroring applied RVs
: %f",typevec[4]/OUTER);
      printf("\nStd.Dev. of the percentage of mirroring applied RVs
: %f\n",sqrt(sumsquare[4]/OUTER));
```

```
      printf("\nExpected # of iterations                          :
%f",100/((typevec[0]/OUTER)+(typevec[1]/OUTER)+(typevec[2]/OUTER)));
      printf("\nUpperbound for expected # of iterations           :
%f",1/(1-subints*ACRIT));
      printf("\nExpected # of uniform variates used               :
%f",200/((typevec[0]/OUTER)+(typevec[1]/OUTER)+(typevec[2]/OUTER)));
      printf("\nUpperbound for expected # of uniform variates used :
%f",2/(1-subints*ACRIT));
      printf("\nExpected # of density function calls              :
%f",(100-(typevec[0]/OUTER)-
(typevec[1]/OUTER))/((typevec[0]/OUTER)+(typevec[1]/OUTER)+(typevec[2]/OU
TER)));
      printf("\nUpperbound for expected # of density function calls:
%f",(subints*ACRIT)/(1-subints*ACRIT));
      printf("\nExpected # of density mirrorings applied          :
%f",(typevec[4]/OUTER)/((typevec[0]/OUTER)+(typevec[1]/OUTER)+(typevec[2]
/OUTER)));
      printf("\nUpperbound for expected # of mirrorings applied   :
%f\n",subints*ACRIT/4);

      printf("\nPress X, then Enter to exit...");
      scanf("%d",&anykey);
      return 0;
}
```

Trilib.h:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAXTABLESIZE 400     /*temporary subinterval vector storage
size*/
#define SEED 1    /*initial seed for uniform random variate generator*/

#define MAXIT 100       /*CDF BETA*/
#define EPS 3.0e-7      /*CDF BETA*/
#define FPMIN 1.0e-30   /*CDF BETA*/

typedef struct{
      int n;            /*number of subintervals*/
      double *sivec;    /*pointer to the array holding subinterval
boundaries*/
      double *cavec;    /*pointer to the array holding subinterval
cumulative probabilities*/
      double *csvec;    /*pointer to the array holding constant squeeze
value*/
      double *lsvec;    /*pointer to the array holding linear squeeze
slopes*/
      double *lsconvec; /*pointer to the array holding linear squeeze
constants*/
      double *lhvec;    /*pointer to the array holding linear hat
slopes*/
      double *lhconvec; /*pointer to the array holding linear hat
constants*/
      int *guidevec;    /*pointer to the array holding indexed search
guide table*/
```

```
} LPAGEN;

void PrintLPATable(LPAGEN *gen)    /*function that prints data table and
guide table by using the pointer*/
{
      int i;
      printf("Ind   Lbound   Ubound   Cumprob  Conssqu  Squgrad  Squcons
Hatgrad  Hatcons\n");
      for (i=0;i<gen->n;i++){
            printf("%3d%9.5f%9.5f%9.5f%9.5f%9.5f%9.5f%9.5f%9.5f\n",
                  i,gen->sivec[i],gen->sivec[i+1],gen->cavec[i+1],gen-
>csvec[i],
                  gen->lsvec[i],gen->lsconvec[i],
                  gen->lhvec[i],gen->lhconvec[i]);
      }
      printf("\nGuide Table:");
      for (i=0;i<2*gen->n;i++){
            if (i%20==0){
                  printf("\n");
            }
            printf("%3d",gen->guidevec[i]);
      }
      printf("\n");
}

double Normal(double x,double mu,double sigma) /*PDF of normal
distribution*/
{
      if (sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            return exp(-(x-mu)*(x-
mu)*0.5/(sigma*sigma))/(2.506628274630963*sigma);
      }
}

double Der1Normal(double x,double mu,double sigma)   /*first derivative
of normal distribution*/
{
      if (sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            return (mu-x)*exp(-(x-mu)*(x-
mu)*0.5/(sigma*sigma))/(2.506628274630963*sigma*sigma*sigma);
      }
}

double CdfNormal(double x,double mu,double sigma)    /*CDF of normal
distribution*/
{
      if(sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            int  help;
```

```
    double xx,x2,x3,x4,x5,x6,x7,x8,zahler,nenner;

    static double
  ep0=242.66795523053175,ep1=21.979261618294152,
  ep2=6.9963834886191355,ep3= -3.5609843701815385e-2,
  eq0=215.0588758698612,eq1=91.164905404514901,
  eq2=15.082797630407787,/*eq3=1,*/
  zp0=300.4592610201616005,zq0=300.4592609569832933,
  zp1=451.9189537118729422,zq1=790.9509253278980272,
  zp2=339.3208167343436870,zq2=931.3540948506096211,
  zp3=152.9892850469404039,zq3=638.9802644656311665,
  zp4=43.16222722205673530,zq4=277.5854447439876434,
  zp5=7.211758250883093659,zq5=77.00015293522947295,
  zp6=0.5641955174789739711,zq6=12.78272731962942351,
  zp7= -1.368648573827167067e-7,/*zq7=1,*/
  dp0= -2.99610707703542174e-3,dq0=1.06209230528467918e-2,
  dp1= -4.94730910623250734e-2,dq1=1.91308926107829841e-1,
  dp2= -2.26956593539686930e-1,dq2=1.05167510706793207,
  dp3= -2.78661308609647788e-1,dq3=1.98733201817135256,
  dp4= -2.23192459734184686e-2/*,dq4=1*/;

    x=(x-mu)/sigma;
    if(x<0)
          help=0;
    else
          help=1;
    x=fabs(x)/sqrt(2.);
    if(x<0.5){
          x2=x*x;
          x4=x2*x2;
          x6=x4*x2;
          zahler=ep0+ep1*x2+ep2*x4+ep3*x6;
          nenner=eq0+eq1*x2+eq2*x4+x6;
          if(help)
                return(0.5*(1+x*zahler/nenner));
          else
                return(0.5*(1-x*zahler/nenner));
    }
    else if(x<4){
          x2=x*x;
          x3=x2*x;
          x4=x3*x;
          x5=x4*x;
          x6=x5*x;
          x7=x6*x;

zahler=zp0+zp1*x+zp2*x2+zp3*x3+zp4*x4+zp5*x5+zp6*x6+zp7*x7;
          nenner=zq0+zq1*x+zq2*x2+zq3*x3+zq4*x4+zq5*x5+zq6*x6+x7;
          if(help)
                return(0.5*(2-exp(-x2)*zahler/nenner));
          else
                return(0.5*exp(-x2)*zahler/nenner);
    }
    else if(x<50){
          xx=x*x;
          x2=1/xx;
          x4=x2*x2;
          x6=x4*x2;
          x8=x6*x2;
          zahler=dp0+dp1*x2+dp2*x4+dp3*x6+dp4*x8;
```

```
                  nenner=dq0+dq1*x2+dq2*x4+dq3*x6+x8;
                  if(help)
                        return(1-0.5*(exp(-
xx)/x)*(1/sqrt(3.141592653589793)+zahler/(xx*nenner)));
                  else
                        return(0.5*(exp(-
xx)/x)*(1/sqrt(3.141592653589793)+zahler/(xx*nenner)));
            }
            else if(help)
                  return(1.);
            else
                  return(0.);
      }
}

double FGamma(double x)       /*gamma function*/
{
      double
            p0=1.000000000190015,p1=76.18009172947146,
            p2=-86.50532032941677,p3=24.01409824083091,
            p4=-1.231739572450155,p5=1.208650973866179e-3,
            p6=-5.395239384953e-6;
      double
sum=p0+(p1/(x+1))+(p2/(x+2))+(p3/(x+3))+(p4/(x+4))+(p5/(x+5))+(p6/(x+6));

      return sum*sqrt(2*3.141592653589793)*pow((x+5.5),(x+0.5))*exp(-
1*(x+5.5))/x;
}

double FBeta(double x,double y)          /*beta function*/
{
      return FGamma(x)*FGamma(y)/FGamma(x+y);
}

double FactorDiv(double x,double y)      /*factorial division*/
{
      double res=1;
      for (x;x>y;x--){
            res=res*x;
      }
      return res;
}

double IncGamma(double alfa,double xbeta)      /*incomplete gamma
function*/
{
      double prop=1,res=0;
      double n=0;

      while (prop>1e-12){
            prop=pow(xbeta,alfa)*exp(-
1*xbeta)*pow(xbeta,n)/FactorDiv(alfa+n,alfa-1);
            res=res+prop;
            n++;
      }
      return res;
}

double BetaCF(double x,double a,double b)      /*evaluates continued
fraction for incomplete beta function*/
```

```
{
	int m,m2;
	double aa,c,d,del,h,qab,qam,qap;

	qab=a+b;
	qap=a+1.0;
	qam=a-1.0;
	c=1.0;
	d=1.0-qab*x/qap;
	if(fabs(d)<FPMIN){
		d=FPMIN;
	}
	d=1.0/d;
	h=d;
	for(m=1;m<MAXIT;m++){
		m2=2*m;
		aa=m*(b-m)*x/((qam+m2)*(a+m2));
		d=1.0+aa*d;
		if(fabs(d)<FPMIN){
			d=FPMIN;
		}
		c=1.0+aa/c;
		if(fabs(c)<FPMIN){
			c=FPMIN;
		}
		d=1.0/d;
		h*=d*c;
		aa=-(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
		d=1.0+aa*d;
		if(fabs(d)<FPMIN){
			d=FPMIN;
		}
		c=1.0+aa/c;
		if(fabs(c)<FPMIN){
			c=FPMIN;
		}
		d=1.0/d;
		del=d*c;
		h*=del;
		if(fabs(del-1.0)<EPS){
			break;
		}
	}
	return h;
}

double IncBeta(double x,double a,double b)		/*incomplete beta
function*/
{
	double bt;
	if(x<0){
		return 0;
	}
	if(x>1){
		return 1;
	}
	if (x==0.0 || x==1.0){
		bt=0.0;
	}
	else {
```

```
            bt=exp(log(FGamma(a+b))-log(FGamma(a))-
log(FGamma(b))+a*log(x)+b*log(1.0-x));
      }
      if (x<(a+1.0)/(a+b+2.0)){
            return bt*BetaCF(x,a,b)/a;
      }
      else{
            return 1.0-bt*BetaCF(1.0-x,b,a)/b;
      }
}

double Gamma(double x,double alfa,double beta)       /*PDF of gamma
distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return pow(x,alfa-1)*pow(beta,alfa)*exp(-
1*beta*x)/FGamma(alfa);
      }
}

double Der1Gamma(double x,double alfa,double beta)               /*first
derivative of gamma distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return pow(beta,alfa)*exp(-1*beta*x)*pow(x,alfa-2)*(alfa-1-
beta*x)/FGamma(alfa);
      }
}

double CdfGamma(double x,double alfa,double beta)                /*CDF of
gamma distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return IncGamma(alfa,x*beta)/FGamma(alfa);
      }
}

double Beta(double x,double alfa,double beta)        /*PDF of beta
distribution*/
```

```
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0 || x>1){
            return 0;
      }
      else{
            return pow(x,alfa-1)*pow(1-x,beta-1)/FBeta(alfa,beta);
      }
}

double CdfBeta(double x,double alfa,double beta)           /*CDF of beta
distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else if (x>1){
            return 1;
      }
      else {
            return IncBeta(x,alfa,beta);
      }
}

double Der1Beta(double x,double alfa,double beta)                /*first
derivative of beta distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<=0 || x>=1){
            return 0;
      }
      else {
            return pow(x,alfa-2)*pow(1-x,beta-2)*(alfa-1-x*(alfa+beta-
2))/FBeta(alfa,beta);
      }
}

double Expo(double x,double lambda,double null)            /*PDF of
exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      else{
            return lambda*exp(-1*lambda*x);
      }
}
```

```
double CdfExpo(double x,double lambda,double null)               /*CDF of
exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return 1-exp(-1*lambda*x);
      }
}

double Der1Expo(double x,double lambda,double null)              /*first
derivative of exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return -1*pow(lambda,2)*exp(-1*lambda*x);
      }
}

double Cauchy(double x,double x0,double gamma)             /*PDF of
cauchy distribution*/
{
      if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 1/(3.141592653589793*gamma*(1+pow((x-x0)/gamma,2)));
}

double CdfCauchy(double x,double x0,double gamma)          /*CDF of
cauchy distribution*/
{
      if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 0.5+(atan((x-x0)/gamma)/3.141592653589793);
}

double Der1Cauchy(double x,double x0,double gamma)               /*first
derivative of cauchy distribution*/
{
            if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 2*(x0-x)*gamma/(3.141592653589793*pow((pow(x0,2)-
2*x0*x+pow(x,2)+pow(gamma,2)),2));
}
```

```
double U01()           /*standard uniform random variate generator*/
{
      static unsigned long int seed=SEED;
      seed=(69069*(seed)+1);
      return(seed/4.294967296e9);
}

double Dfchia(double x,long n)           /*approximation*/
{
      double chix,y;
      y=2./(9.*n);
      chix=(exp(log(x/n)/3)-1+y)/sqrt(y);
      return(CdfNormal(chix,0,1));
}

double Chi2Test(long b[],                      /*obsereved frequencies*/
                long l,                        /*number of classes*/
                       long wid,                     /*sample size*/
                       int printflag)                /*0..no
output,1..little,2..more*/
{
      long int i;
      double chi2=0.,erw,pval;

      erw=(double)wid/l;
      if (erw<5.){
            printf("Error Chi2Test: expected frequency smaller than
5\n");
      }
      for (i=0;i<l;++i){
            if (printflag==2){
                  printf("%4ld:%4ld;",i,b[i]);
            }
            chi2+= (b[i]-erw)*(b[i]-erw);
      }
      chi2/=erw;
      pval=1.-Dfchia(chi2,l-1);
      if (printflag>=1)
      {
            printf("\nChi2-test: samplesize=%ld  number of classes= %ld
\n",wid,l);
            printf("Chi2-value %f   Approximate P-Value:
%.15f\n",chi2,pval);
      }
      return(chi2);
}

int IndexSearch(double u,                /*std. uniform RV*/
                        int guide[],           /*guide table*/
                        int gwidth,            /*length of guide table*/
                        double cdfvec[])       /*cumulative probability
values*/
{
      int sind=u*gwidth;
      int x;
      x=guide[sind];
      while (u>cdfvec[x]){
            x++;
      }
```

```
      return x-1;
}

int FlexSubint(double lcutoff,     /*left cutoff point of the unbounded
density*/
                  double linflect,  /*left inflection point*/
                  double mode,      /*global maxima of the density*/
                  double rinflect,  /*right inflection point*/
                  double rcutoff,   /*right cutoff point of the unbounded
density*/
                  double acrit,     /*max. area above constant squeeze
for each subinterval*/
                  double *subints,  /*pointer to the vector that will
store subintervals*/
                  double (*dens)(double a,double b,double c),/*pointer to
corresponding distribution density function*/
                  double p1,        /*parameter 1*/
                  double p2)        /*parameter 2*/
/*creates flexible subintervals with an heuristic method*/
{
      double initsivec[5]={lcutoff,linflect,mode,rinflect,rcutoff};
      int ninit=4;
      double last=initsivec[ninit]; /*upper boundary of questioned
subinterval for dividing*/
      double start=initsivec[0];          /*lower boundary of questioned
subinterval for dividing*/
      int k=ninit-1;
      double aim[MAXTABLESIZE];           /*vector that holds sequenced
upper boundaries of subintervals to be questioned*/
      while (k>=0){
            aim[k]=initsivec[ninit-k];

            k--;
      }

      k=ninit-1;
      int i=0;
      double aquest;    /*the questioned area*/

      subints[i]=initsivec[0];
      while (subints[i]!=last){
            if (aim[k]<linflect || aim[k]>rinflect || start<linflect ||
            start>rinflect){
                  aquest=(aim[k]-start)*fabs((*dens)(aim[k],p1,p2)-
                  (*dens)(start,p1,p2))/2;
            }
            else {
                  if ((*dens)(aim[k],p1,p2)<(*dens)(start,p1,p2)){
                        aquest=(aim[k]-
                        start)*((*dens)((aim[k]+start)/2,p1,p2)-
                        (*dens)(aim[k],p1,p2));
                  }
                  else{
                        aquest=(aim[k]-
                        start)*((*dens)((aim[k]+start)/2,p1,p2)-
                        (*dens)(start,p1,p2));
                  }
            }
            if (aquest<=acrit){     /*accept subinterval*/
                  i++;
```

```
                  subints[i]=aim[k];
                  start=subints[i];
                  k--;
            }
            else {      /*reject and divide subinterval*/
                  k++;
                  aim[k]=(start+aim[k-1])/2;
            }
      }
      return i;
}

void GuideTable(int length,double *cumvec,int *outputvec){ /*produces a
guide table, double size of subinterval vector*/
      outputvec[0]=0;
      int i=0;
      int j;

      for (j=1;j<(2*length);j++){
            while (((double)j*cumvec[length-1]/(2*length))>cumvec[i]){
                  i++;
            }
            outputvec[j]=i;
      }
}

LPAGEN *Setup(double lcutoff,            /*left cutoff point of the
unbounded density*/
              double linflect,     /*left inflection point*/
              double mode,         /*global maxima of the density*/
              double rinflect,     /*right inflection point*/
              double rcutoff,      /*right cutoff point of the unbounded
density*/
              double acrit,        /*max. area above constant squeeze
for each subinterval*/
              double (*dens)(double a,double b,double c),  /*pointer to
the corresponding distribution density function*/
              double (*der1)(double d,double e,double f),  /*pointer to
the first derivative of distribution density function*/
              double p1,      /*parameter 1*/
              double p2)      /*parameter 2*/
/*function returns a pointer to the data table*/
{
      double subints[MAXTABLESIZE]; /*vector that records approved
subintervals*/
      int i;
      i=FlexSubint(lcutoff,linflect,mode,rinflect,rcutoff,acrit,subints,(
      *dens),p1,p2);

      LPAGEN *gen;      /*a structure that holds data table*/
      gen=(LPAGEN *)malloc(sizeof(LPAGEN));
      gen->sivec=(double *)malloc(sizeof(double)*(i+1));   /*allocation
for subintervals of the data table*/
      gen->cavec=(double *)malloc(sizeof(double)*(i+1));   /*allocation
for cumulative probabilities of the data table*/
      gen->csvec=(double *)malloc(sizeof(double)*i);       /*allocation
for constant squeezes of the data table*/
      gen->lsvec=(double *)malloc(sizeof(double)*i);       /*allocation
for linear squeeze slopes of the data table*/
```

```
      gen->lsconvec=(double *)malloc(sizeof(double)*i);    /*allocation
for linear squeeze constants of the data table*/
      gen->lhvec=(double *)malloc(sizeof(double)*i);       /*allocation
for linear hat slopes of the data table*/
      gen->lhconvec=(double *)malloc(sizeof(double)*i);    /*allocation
for linear hat constants of the data table*/
      gen->guidevec=(int *)malloc(sizeof(int)*2*i);        /*allocation
for the guide table of indexed search*/
      gen->n=i;
      gen->sivec[0]=subints[0];
      gen->cavec[0]=0;

      int j,concave=1;
      double sum=0;
      double midpoint,intlngth,lborval,rborval;


      for(j=1;j<=i;j++){/*depending on the property of the subinterval
(convexity, concavity, increasing or decreasing) calculates table
values*/

            midpoint=(subints[j]+subints[j-1])/2;
            intlngth=subints[j]-subints[j-1];
            lborval=(*dens)(subints[j-1],p1,p2);
            rborval=(*dens)(subints[j],p1,p2);

            gen->sivec[j]=subints[j];
            if (lborval<=rborval){
                  gen->csvec[j-1]=lborval;
            }
            else {
                  gen->csvec[j-1]=rborval;
            }
            if (subints[j-1]<linflect || subints[j-1]>rinflect ||
            subints[j]<linflect || subints[j]>rinflect){
                  gen->lhvec[j-1]=(rborval-lborval)/intlngth;
                  gen->lhconvec[j-1]=(rborval+lborval)/2;
                  gen->lsvec[j-1]=(*der1)(midpoint,p1,p2);
                  gen->lsconvec[j-1]=(*dens)(midpoint,p1,p2);
            }
            else {
                  gen->lhvec[j-1]=(*der1)(midpoint,p1,p2);
                  gen->lhconvec[j-1]=(*dens)(midpoint,p1,p2);
                  gen->lsvec[j-1]=(rborval-lborval)/intlngth;
                  gen->lsconvec[j-1]=(rborval+lborval)/2;
            }
      gen->cavec[j]=sum+intlngth*gen->lhconvec[j-1];
      sum=gen->cavec[j];
      }

      for (j=1;j<=gen->n;j++){     /*cumulative vector normalized*/
            gen->cavec[j]=gen->cavec[j]/gen->cavec[gen->n];
      }

      GuideTable(gen->n,gen->cavec,gen->guidevec);
      return gen;
}
int FreeGen(LPAGEN *gen){    /*after termination, the function frees
structure address*/
      free(gen->sivec);
```

```
      free(gen->cavec);
      free(gen->csvec);
      free(gen->lsvec);
      free(gen->lsconvec);
      free(gen->lhvec);
      free(gen->lhconvec);
      free(gen->guidevec);
      free(gen);
      return 0;
}

double Generator(LPAGEN *lpagen,
                        double (*dens)(double a,double b,double c),
                        double p1,
                        double p2,
                        int *counter,    /*a vector that holds accepted
variate types*/
                        int countflag)   /*other than zero, it counts*/
{     /*Generates a random variate through the data table*/
      double u1,u2,urc;
      int index,accept=0;
      double x,u,c,height,sqheight,length,lb,ub;

      while(accept==0){
            u1=U01();
            u2=U01();
            index=IndexSearch(u1,lpagen->guidevec,2*lpagen->n,lpagen-
            >cavec);    /*Subinterval index is determined with u1*/
            lb=lpagen->sivec[index];
            ub=lpagen->sivec[index+1];
            length=ub-lb;
            height=lpagen->lhconvec[index];
            urc=(u1-lpagen->cavec[index])/(lpagen->cavec[index+1]-lpagen-
            >cavec[index]);
            x=lpagen->sivec[index]+length*urc;  /*RV is determined with
recycled RV*/
            u=u2*height;      /*a corresponding ordinate is determined
with u2*/
            c=(lpagen->sivec[index]+lpagen->sivec[index+1])/2;   /*center
point is calculated*/
            if (u<=lpagen->csvec[index]){      /*below the constant
squeeze, immediately acceptance*/
                  if (countflag!=0){counter[0]++;}
                  accept=1;
            }
            else{
                  if (u>(lpagen->lhconvec[index]+lpagen->lhvec[index]*(x-
c))){ /*above the linear hat, must be mirrored*/
                        if (countflag!=0){counter[4]++;}
                        x=2*c-x;
                        u=2*height-u;
                  }
                  if (u<=(lpagen->lsconvec[index]+lpagen-
>lsvec[index]*(x-c))){  /*below the linear squeeeze, accepted*/
                        if (countflag!=0){counter[1]++;}
                        accept=1;
                  }
                  else if (u<=(*dens)(x,p1,p2)){     /*below the density,
accepted*/
                        if (countflag!=0){counter[2]++;}
```

```
                        accept=1;
                    }
                    else{       /*above the density rejected*/
                        if (countflag!=0){counter[3]++;}
                    }
                }
            }
            return x;
}
```

APPENDIX B: POLYNOMIAL DENSITY INVERSION C CODES

Following codes are the c application of the Polynomial Density Inversion algorithm. The algorithm is coded as a main function. Also there is a header file that includes the setup, the sampling and all other helping functions.

Main Function:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "pdilib.h"

#define OUTER 1         /*outer repetition increase the accuracy of time
length, repeats the setup and generation*/
#define RVS 1000000     /*number of random variates to be generated after
setup*/
#define CLASSES 1000    /*number of classes for chi-square test,
sqrt(RVS) is desirable*/
#define PRINTFLAG 1     /*prints out the setup table for each outer
loop*/
#define CHI2RESULT 1    /*makes a chi-square test for random variates in
each outer loop*/
#define COUNTFLAG 1     /*counts and prints the random variates according
to the generation types*/

#define ERROR 0.01      /*heuristic subinterval creation parameter,
critical linear error*/
#define RERROR 0.1      /*heuristic subinterval creation parameter,
critical relative error*/
#define PERROR 1.e-8    /*heuristic subinterval creation parameter,
critical polynomial error*/
#define DENSITY Normal  /*distribution type to be generated*/
#define DERIVATIVE Der1Normal       /*first derivative of the
distribution*/
#define CUMULATIVE CdfNormal        /*cumulative distribution for chi-
square test*/
#define P1 0      /*parameter 1*/
#define P2 1      /*parameter 2*/
#define LEFTCUTOFF -6
/*NORMAL(0,1) -6*/                  /*CAUCHY(0,1) -640000*/
/*EXPO(1,NULL) 0*/                  /*GAMMA(3,1) 0.001*/
/*BETA(3,4) 0.001*/                 /*BETA(30,40) 0.15*/
#define LEFTINFLECTION -1
/*NORMAL(0,1) -1*/                  /*CAUCHY(0,1) -0.577350269189625*/
/*EXPO(1,NULL) 1.e-12*/             /*GAMMA(3,1) 0.585786437626905*/
/*BETA(3,4) 0.15505102572168*/      /*BETA(30,40) 0.366049994532764*/
#define MODE 0
/*NORMAL(0,1) 0*/                   /*CAUCHY(0,1) 0*/
/*EXPO(1,NULL) 2.e-12*/             /*GAMMA(3,1) 2*/
```

```
/*BETA(3,4) 0.4*/                   /*BETA(30,40) 0.426470588235294*/
#define RIGHTINFLECTION 1
/*NORMAL(0,1) 1*/                   /*CAUCHY(0,1) 0.577350269189625*/
/*EXPO(1,NULL) 3.e-12*/             /*GAMMA(3,1) 3.414213562373095*/
/*BETA(3,4) 0.6449489742783*/       /*BETA(30,40) 0.486891181937824*/
#define RIGHTCUTOFF 6
/*NORMAL(0,1) 6*/                   /*CAUCHY(0,1) 640000*/
/*EXPO(1,NULL) 17*/                 /*GAMMA(3,1) 21*/
/*BETA(3,4) 0.999*/                 /*BETA(30,40) 0.725*/

int main()
{
      int anykey;

      int j=0,i,subints;
      time_t genstart,genend;
      double gendif,rv;       /*variables that hold timing result and
generated RVs*/
      double typevec[3]={0};  /*a vector that holds the average
percentage of sampling types*/
      double sumsquare[3]={0};/*a vector that holds the std. deviation of
the percentage of sampling types*/
      double typemat[3][OUTER];/*a matrix that holds the percentage of
sampling types for each run*/

      time (&genstart);       /*this is where the timing starts*/
      double sum=0;           /*an arbitrary operation variable to be
used with each random variate*/
      for(j=0;j<OUTER;j++){   /*beginning of the outer loop*/
            /*SETUP PART*/
            PDIGEN *pdigen;   /*a struct that will store the table*/
            pdigen=Setup(LEFTCUTOFF,LEFTINFLECTION,MODE,RIGHTINFLECTION,
            RIGHTCUTOFF,ERROR,RERROR,PERROR,DENSITY,DERIVATIVE,P1,P2);
            if (PRINTFLAG!=0){
                  PrintPDITable(pdigen);  /*command that prints out the
table*/
            }
            subints=pdigen->n;
            /*SETUP ENDS*/
            /*SAMPLING PART*/
            int counter[3]={0};     /*count vector to count three types
of generation*/
            long freq[CLASSES]={0}; /*frequency vector that will store
F(rv) in #CLASSES equidistant intervals*/
            for (i=0;i<RVS;i++){
                  rv=Generator(pdigen,counter,COUNTFLAG);
                  sum+=rv;    /*an arbitrary insignificant (not
expensive) operation*/
                  if (CHI2RESULT!=0){
                        freq[(int)(CLASSES*CUMULATIVE(rv,P1,P2))]++;
      /*counts F(rv)as plus 1 in its interval*/
                  }
            }
            typevec[0]+=typemat[0][j]=(double)counter[0]*100/RVS;
      /*assigments for efficiency characteristics*/
            typevec[1]+=typemat[1][j]=(double)counter[1]*100/RVS;
      /*assigments for efficiency characteristics*/
            typevec[2]+=typemat[2][j]=(double)counter[2]*100/RVS;
      /*assigments for efficiency characteristics*/
            /*SAMPLING ENDS*/
```

```
            if (CHI2RESULT!=0){
                  Chi2Test(freq,CLASSES,RVS,CHI2RESULT);    /*calls chi-
square test with frequency vector, tests if it is uniform*/

            }
            if (COUNTFLAG!=0){      /*prints out number of random
variates generated with three types of generation*/
                  printf("\nTotal # of RVs: %d",RVS);
                  printf("\n# of RVs returned through uniform
distribution   : %d"
                        "\n# of RVs returned through triangular
distribution: %d"
                        "\n# of RVs returned through polynomial inversion
: %d\n",counter[0],counter[1],counter[2]);
            }
            FreeGen(pdigen);  /*frees the memory*/
      }
      printf("\nMean: %f\n",sum/(OUTER*RVS));  /*the result of the
arbitrary operations*/
      time (&genend);   /*this is where the timing ends*/

      gendif=difftime(genend,genstart);

      for (i=0;i<=2;i++){
            for(j=0;j<OUTER;j++){
                  sumsquare[i]+=(typemat[i][j]-
                  (typevec[i]/OUTER))*(typemat[i][j]-(typevec[i]/OUTER));
            }
      }

      printf("\nMEMORY OCCUPATION:");     /*prints out the number of
subintervals in the table*/
      printf("\n# of subintervals: %d\n",subints);
      printf("\nTIMING RESULTS:");
      printf("\nProcessing time: %f",gendif);  /*prints out the whole
execution time*/
      printf("\nSingle loop time: %f\n",(double)gendif/OUTER);   /*prints
out the average of single loop (setup and generation) times*/
      printf("\nEFFICIENCY:");      /*prints out the mean and the std.
dev. of the percentage of sampling types*/
      printf("\nAverage percentage of RVs returned through uniform
distribution     : %f",typevec[0]/OUTER);
      printf("\nStd.Dev. of the percentage RVs returned through uniform
distribution: %f",sqrt(sumsquare[0]/OUTER));
      printf("\nAverage percentage of RVs returned through triangular
distribution  : %f",typevec[1]/OUTER);
      printf("\nStd.Dev. of the percentage RVs returned through uniform
distribution: %f",sqrt(sumsquare[1]/OUTER));
      printf("\nAverage percentage of RVs returned through polynomial
distribution  : %f",typevec[2]/OUTER);
      printf("\nStd.Dev. of the percentage RVs returned through uniform
distribution: %f\n",sqrt(sumsquare[2]/OUTER));

      printf("\nPress X, then Enter to exit...");
      scanf("%d",&anykey);
      return 0;
}
```

Pdilib.h:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define PTS             5     /*number of interpolation points in each
subinterval, all coefficients are not printed if PTS>5*/
#define ERR             1.e-12      /*critical error for root-finding
algorithm*/
#define MAXTABLESIZE    900   /*temporary subinterval vector storage
size*/
#define SEED            1     /*initial seed for uniform random variate
generator*/

#define MAXIT 100       /*CDF BETA*/
#define EPS 3.0e-7      /*CDF BETA*/
#define FPMIN 1.0e-30   /*CDF BETA*/

typedef struct{
      int n;            /*number of subintervals*/
      double mode;      /*mode of the distribution*/
      double *sivec;    /*pointer to the array holding subintervals*/
      double *pcvec;    /*pointer to the array holding coefficient
matrix*/
      double *cavec;    /*pointer to the array holding subinterval
areas*/
      double *rcvec;    /*pointer to the array holding rectangle areas*/
      double *trvec;    /*pointer to the array holding triangle areas*/
      int *guidevec;    /*pointer to the array holding indexed search
guide table*/
} PDIGEN;

void PrintPDITable(PDIGEN *gen)     /*function that prints the table and
guide table by using the pointer*/
{
      int i;
      printf("Ind Lbound Ubound Cumprob    Rect    Tri     x0    x1    x2
x3    x4\n");
      for (i=0;i<gen->n;i++){

      printf("%3d%7.3f%7.3f%8.4f%8.4f%8.4f%6.2f%6.2f%6.2f%6.2f%6.2f\n",
                  i,gen->sivec[i],gen->sivec[i+1],
                  gen->cavec[i+1],gen->rcvec[i],gen->trvec[i],
                  gen->pcvec[5*i],gen->pcvec[5*i+1],
                  gen->pcvec[5*i+2],gen->pcvec[5*i+3],
                  gen->pcvec[5*i+4]);
      }
      printf("\nGuide Table:");
      for (i=0;i<2*gen->n;i++){
            if (i%20==0){
                  printf("\n");
            }
            printf("%3d",gen->guidevec[i]);
      }
      printf("\n");
}

double Normal(double x,double mu,double sigma) /*PDF of normal
distribution*/
```

```
{
      if (sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            return exp(-(x-mu)*(x-
mu)*0.5/(sigma*sigma))/(2.506628274630963*sigma);
      }
}

double Der1Normal(double x,double mu,double sigma)   /*first derivative
of normal distribution*/
{
      if (sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            return (mu-x)*exp(-(x-mu)*(x-
mu)*0.5/(sigma*sigma))/(2.506628274630963*sigma*sigma*sigma);
      }
}

double CdfNormal(double x,double mu,double sigma)    /*CDF of normal
distribution*/
{
      if(sigma<=0){
            printf("error\n");
            return 0;
      }
      else{
            int  help;
            double xx,x2,x3,x4,x5,x6,x7,x8,zahler,nenner;

            static double
        ep0=242.66795523053175,ep1=21.979261618294152,
        ep2=6.9963834886191355,ep3= -3.5609843701815385e-2,
        eq0=215.0588758698612,eq1=91.164905404514901,
        eq2=15.082797630407787,/*eq3=1,*/
        zp0=300.4592610201616005,zq0=300.4592609569832933,
        zp1=451.9189537118729422,zq1=790.9509253278980272,
        zp2=339.3208167343436870,zq2=931.3540948506096211,
        zp3=152.9892850469404039,zq3=638.9802644656311665,
        zp4=43.16222722205673530,zq4=277.5854447439876434,
        zp5=7.211758250883093659,zq5=77.00015293522947295,
        zp6=0.5641955174789739711,zq6=12.78272731962942351,
        zp7= -1.368648573827167067e-7,/*zq7=1,*/
        dp0= -2.99610707703542174e-3,dq0=1.06209230528467918e-2,
        dp1= -4.94730910623250734e-2,dq1=1.91308926107829841e-1,
        dp2= -2.26956593539686930e-1,dq2=1.05167510706793207,
        dp3= -2.78661308609647788e-1,dq3=1.98733201817135256,
        dp4= -2.23192459734184686e-2/*,dq4=1*/;

            x=(x-mu)/sigma;
            if(x<0)
                  help=0;
            else
                  help=1;
            x=fabs(x)/sqrt(2.);
```

```
            if(x<0.5){
                  x2=x*x;
                  x4=x2*x2;
                  x6=x4*x2;
                  zahler=ep0+ep1*x2+ep2*x4+ep3*x6;
                  nenner=eq0+eq1*x2+eq2*x4+x6;
                  if(help)
                        return(0.5*(1+x*zahler/nenner));
                  else
                        return(0.5*(1-x*zahler/nenner));
            }
            else if(x<4){
                  x2=x*x;
                  x3=x2*x;
                  x4=x3*x;
                  x5=x4*x;
                  x6=x5*x;
                  x7=x6*x;

      zahler=zp0+zp1*x+zp2*x2+zp3*x3+zp4*x4+zp5*x5+zp6*x6+zp7*x7;
                  nenner=zq0+zq1*x+zq2*x2+zq3*x3+zq4*x4+zq5*x5+zq6*x6+x7;
                  if(help)
                        return(0.5*(2-exp(-x2)*zahler/nenner));
                  else
                        return(0.5*exp(-x2)*zahler/nenner);
            }
            else if(x<50){
                  xx=x*x;
                  x2=1/xx;
                  x4=x2*x2;
                  x6=x4*x2;
                  x8=x6*x2;
                  zahler=dp0+dp1*x2+dp2*x4+dp3*x6+dp4*x8;
                  nenner=dq0+dq1*x2+dq2*x4+dq3*x6+x8;
                  if(help)
                        return(1-0.5*(exp(-
xx)/x)*(1/sqrt(3.141592653589793)+zahler/(xx*nenner)));
                  else
                        return(0.5*(exp(-
xx)/x)*(1/sqrt(3.141592653589793)+zahler/(xx*nenner)));
            }
            else if(help)
                  return(1.);
            else
                  return(0.);
      }
}

double FGamma(double x)       /*gamma function*/
{
      double
            p0=1.000000000190015,p1=76.18009172947146,
            p2=-86.50532032941677,p3=24.01409824083091,
            p4=-1.231739572450155,p5=1.208650973866179e-3,
            p6=-5.395239384953e-6;
      double
sum=p0+(p1/(x+1))+(p2/(x+2))+(p3/(x+3))+(p4/(x+4))+(p5/(x+5))+(p6/(x+6));

      return sum*sqrt(2*3.141592653589793)*pow((x+5.5),(x+0.5))*exp(-
1*(x+5.5))/x;
```

```
}

double FBeta(double x,double y)          /*beta function*/
{
      return FGamma(x)*FGamma(y)/FGamma(x+y);
}

double FactorDiv(double x,double y)      /*factorial division*/
{
      double res=1;
      for (x;x>y;x--){
            res=res*x;
      }
      return res;
}

double IncGamma(double alfa,double xbeta)      /*incomplete gamma
function*/
{
      double prop=1,res=0;
      double n=0;

      while (prop>1e-12){
            prop=pow(xbeta,alfa)*exp(-
1*xbeta)*pow(xbeta,n)/FactorDiv(alfa+n,alfa-1);
            res=res+prop;
            n++;
      }
      return res;
}

double BetaCF(double x,double a,double b)      /*evaluates continued
fraction for incomplete beta function*/
{
      int m,m2;
      double aa,c,d,del,h,qab,qam,qap;

      qab=a+b;
      qap=a+1.0;
      qam=a-1.0;
      c=1.0;
      d=1.0-qab*x/qap;
      if(fabs(d)<FPMIN){
            d=FPMIN;
      }
      d=1.0/d;
      h=d;
      for(m=1;m<MAXIT;m++){
            m2=2*m;
            aa=m*(b-m)*x/((qam+m2)*(a+m2));
            d=1.0+aa*d;
            if(fabs(d)<FPMIN){
                  d=FPMIN;
            }
            c=1.0+aa/c;
            if(fabs(c)<FPMIN){
                  c=FPMIN;
            }
            d=1.0/d;
            h*=d*c;
```

```
            aa=-(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
            d=1.0+aa*d;
            if(fabs(d)<FPMIN){
                  d=FPMIN;
            }
            c=1.0+aa/c;
            if(fabs(c)<FPMIN){
                  c=FPMIN;
            }
            d=1.0/d;
            del=d*c;
            h*=del;
            if(fabs(del-1.0)<EPS){
                  break;
            }
      }
      return h;
}

double IncBeta(double x,double a,double b)           /*incomplete beta
function*/
{
      double bt;
      if(x<0){
            return 0;
      }
      if(x>1){
            return 1;
      }
      if (x==0.0 || x==1.0){
            bt=0.0;
      }
      else {
            bt=exp(log(FGamma(a+b))-log(FGamma(a))-
log(FGamma(b))+a*log(x)+b*log(1.0-x));
      }
      if (x<(a+1.0)/(a+b+2.0)){
            return bt*BetaCF(x,a,b)/a;
      }
      else{
            return 1.0-bt*BetaCF(1.0-x,b,a)/b;
      }
}

double Gamma(double x,double alfa,double beta)       /*PDF of gamma
distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return pow(x,alfa-1)*pow(beta,alfa)*exp(-
1*beta*x)/FGamma(alfa);
      }
}
```

```
double Der1Gamma(double x,double alfa,double beta)                /*first
derivative of gamma distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return pow(beta,alfa)*exp(-1*beta*x)*pow(x,alfa-2)*(alfa-1-
beta*x)/FGamma(alfa);
      }
}

double CdfGamma(double x,double alfa,double beta)                 /*CDF of
gamma distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return IncGamma(alfa,x*beta)/FGamma(alfa);
      }
}

double Beta(double x,double alfa,double beta)         /*PDF of beta
distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0 || x>1){
            return 0;
      }
      else{
            return pow(x,alfa-1)*pow(1-x,beta-1)/FBeta(alfa,beta);
      }
}

double CdfBeta(double x,double alfa,double beta)            /*CDF of beta
distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else if (x>1){
            return 1;
      }
      else {
```

```
            return IncBeta(x,alfa,beta);
      }
}

double Der1Beta(double x,double alfa,double beta)                /*first
derivative of beta distribution*/
{
      if (alfa<=0 || beta<=0){
            printf("error\n");
            return 0;
      }
      if (x<=0 || x>=1){
            return 0;
      }
      else {
            return pow(x,alfa-2)*pow(1-x,beta-2)*(alfa-1-x*(alfa+beta-
2))/FBeta(alfa,beta);
      }
}

double Expo(double x,double lambda,double null)            /*PDF of
exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      else{
            return lambda*exp(-1*lambda*x);
      }
}

double CdfExpo(double x,double lambda,double null)               /*CDF of
exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return 1-exp(-1*lambda*x);
      }
}

double Der1Expo(double x,double lambda,double null)              /*first
derivative of exponential distribution*/
{
      if (lambda<=0){
            printf("error\n");
            return 0;
      }
      if (x<0){
            return 0;
      }
      else{
            return -1*pow(lambda,2)*exp(-1*lambda*x);
      }
```

```
}

double Cauchy(double x,double x0,double gamma)             /*PDF of
cauchy distribution*/
{
      if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 1/(3.141592653589793*gamma*(1+pow((x-x0)/gamma,2)));
}

double CdfCauchy(double x,double x0,double gamma)          /*CDF of
cauchy distribution*/
{
      if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 0.5+(atan((x-x0)/gamma)/3.141592653589793);
}

double Der1Cauchy(double x,double x0,double gamma)               /*first
derivative of cauchy distribution*/
{
            if (gamma<=0){
            printf("error\n");
            return 0;
      }
      return 2*(x0-x)*gamma/(3.141592653589793*pow((pow(x0,2)-
2*x0*x+pow(x,2)+pow(gamma,2)),2));
}

double U01()           /*standard uniform random variate generator*/
{
      static unsigned long int seed=SEED;
      seed=(69069*(seed)+1);
      return(seed/4.294967296e9);
}

double Dfchia(double x,long n)           /*approximation*/
{
      double chix,y;
      y=2./(9.*n);
      chix=(exp(log(x/n)/3)-1+y)/sqrt(y);
      return(CdfNormal(chix,0,1));
}

double Chi2Test(long b[],          /*obsereved frequencies*/
                  long l,          /*number of classes*/
                  long wid,        /*sample size*/
                  int printflag)   /*0..no output,1..little,2..more*/
{
      long int i;
      double chi2=0.,erw,pval;

      erw=(double)wid/l;
      if (erw<5.){
            printf("Error Chi2Test: expected frequency smaller than
5\n");
```

```
      }
      for (i=0;i<l;++i){
            if (printflag==2){
                  printf("%4ld:%4ld;",i,b[i]);
            }
            chi2+= (b[i]-erw)*(b[i]-erw);
      }
      chi2/=erw;
      pval=1.-Dfchia(chi2,l-1);
      if (printflag>=1)
      {
            printf("\nChi2-test: samplesize=%ld  number of classes= %ld
\n",wid,l);
            printf("Chi2-value %f   Approximate P-Value:
%.15f\n",chi2,pval);
      }
      return(chi2);
}

int IndexSearch(double u,           /*std. uniform RV*/
                  int guide[],      /*guide table*/
                  int gwidth,       /*length of guide table*/
                  double cdfvec[])  /*cumulative probability values*/
{
      int sind=u*gwidth;
      int x;
      x=guide[sind];
      while (u>cdfvec[x]){
            x++;
      }
      return x-1;
}

void Integ(double coeff[], double *resvec) /*integrates a polynomial
coefficients*/
{
      resvec[0]=0;
      int i;
      for (i=1;i<PTS+1;i++){
            resvec[i]=coeff[i-1]/i;
      }
}

double Pol(double x,double coeffs[],int n){    /*returns a polynomial
function response*/
      int i;
      double res=0;
      for (i=0;i<n;i++){
            res+=pow(x,i)*coeffs[i];
      }
      return res;
}

double PinvBrent(double rhs,double a,double b,double coeffs[],int
n,double error)   /*finds the unique root of a polynomial in an
interval*/
{
      double d=0,h,fh,c,fc,s,fs,maxint,minint;
      int mflag;
      double fa=Pol(a,coeffs,n)-rhs;
```

```
	double fb=Pol(b,coeffs,n)-rhs;
	if (fa*fb>0){
		printf("error");
	}
	else{
		if (fabs(fa)<=fabs(fb)){
			h=a;
			fh=fa;
			a=b;
			fa=fb;
			b=h;
			fb=fh;
		}
		c=a;
		fc=fa;
		mflag=1;
		int z=0;
		while (fabs(fb)>=error){
			if((fa!=fc) && (fb!=fc)){
				s=(a*fb*fc/((fa-fb)*(fa-fc)))+(b*fa*fc/((fb-
fa)*(fb-fc)))+(c*fa*fb/((fc-fa)*(fc-fb)));
			}
			else{
				s=b-fb*(b-a)/(fb-fa);
			}
			if (((3*a+b)/4)>b){
				maxint=((3*a+b)/4);
				minint=b;
			}
			else {
				minint=((3*a+b)/4);
				maxint=b;
			}
			if((s<minint || s>maxint) || (mflag==1 && fabs(s-
b)>(fabs(b-c)/2)) || (mflag==0 && fabs(s-b)>(fabs(c-d)/2))){
				s=(a+b)/2;
				mflag=1;
			}
			else {
				mflag=0;
			}
			fs=Pol(s,coeffs,n)-rhs;
			d=c;
			c=b;
			if (fa*fs<0){
				b=s;
				fb=fs;
			}
			else {
				a=s;
				fa=fs;
			}
			if (fabs(fa)<=fabs(fb)){
				h=a;
				fh=fa;
				a=b;
				fa=fb;
				b=h;
				fb=fh;
			}
```

```
            }
      }
      return b;
}

void ChebyPoints(double a,double b,double *vec)      /*evaluates
interpolation points in an interval*/
{
      double phi=3.141592653589793/(2*PTS);
      double diff[PTS];
      int i=0;

      diff[i]=sin(2*i*phi)*tan(phi);

      for (i=1;i<PTS;i++){
            diff[i]=sin(2*i*phi)*tan(phi)+diff[i-1];
      }
      for(i=0;i<PTS;i++){
            vec[i]=a+(b-a)*diff[i];
      }
}

void ChebyExtrema(double a,double b,double *vec)     /*evaluates
approximation error control points in an interval*/
{
      int z;
      double pi=3.141592653589793;
      for (z=0;z<PTS-1;z++){
            vec[z]=0.5*(a+b)+0.5*(b-a)*cos((PTS-z-
1)*pi/(PTS))/cos(pi/(2*(PTS)));
      }
}

double CombSum(double vec[],int n,int m)  /*sum of the multiplication of
n-element subsets of an m element vector*/
{
      double res=0;
      double revec[PTS];
      int i;

      if (n>m || n<=0){
            res=0;
      }
      else if (n==1){
            for (i=0;i<m;i++){
                  res+=vec[i];
            }
      }
      else {
            for (i=1;i<m;i++){
                  revec[i-1]=vec[i];
            }
            res=vec[0]*CombSum(revec,n-1,m-1)+CombSum(revec,n,m-1);
      }
      return res;
}

void Interpol(double xvec[],double yvec[],double *coeffs)  /*interpolates
points and returns coefficients of the interpolation polynomial*/
{
```

```
	double dtable[PTS][PTS]={0};
	double dmat[PTS][PTS]={0};
	double xmat[PTS][PTS]={0};
	double dvec[PTS]={0};
	int i,j;

	/*CREATION OF DMATRIX*/

	for (j=0;j<PTS;j++){
		dtable[0][j]=yvec[j];
	}
	for (i=1;i<PTS;i++){
		for (j=i;j<PTS;j++){
			dtable[i][j]=(dtable[i-1][j]-dtable[i-1][j-
			1])/(xvec[j]-xvec[j-i]);
		}
	}

	for (j=0;j<PTS;j++){
		dvec[j]=dtable[j][j];
	}
	for (i=0;i<PTS;i++){
		for (j=0;j<PTS-i;j++){
			dmat[i][j]=dvec[i+j];
			if (j%2){
				dmat[i][j]*=-1;
			}
		}
	}

	/*CREATION OF XMATRIX*/

	for (i=0;i<PTS;i++){
		xmat[i][0]=1;
	}
	for (i=0;i<PTS-1;i++){
		for (j=1;j<PTS-i;j++){
			xmat[i][j]=CombSum(xvec,j,i+j);
		}
	}

	/*CREATION OF RESULT*/

	for (i=0;i<PTS;i++){
		coeffs[i]=0;
		for (j=0;j<PTS;j++){
			coeffs[i]+=dmat[i][j]*xmat[i][j];
		}
	}
}

double UppSearch(double u,double coeff[],double lb,double ub)    /*root-
finding algorithm directly used by generation algorithm*/
{
	double cmlf[PTS+1];
	Integ(coeff,cmlf);
	double intlb=Pol(lb,cmlf,PTS+1);
	double intub=Pol(ub,cmlf,PTS+1);
	double rhs=(intub-intlb)*u+intlb;
	return PinvBrent(rhs,lb,ub,cmlf,PTS+1,ERR);
```

```
}

double Min2(double x1,double x2)         /*returns the minimum of two
real numbers*/
{
      if (x1<=x2){
            return x1;
      }
      else{
            return x2;
      }
}

double Max2(double x1,double x2)         /*returns the minimum of two
real numbers*/
{
      if (x1<=x2){
            return x2;
      }
      else{
            return x1;
      }
}

int FlexSubint(double lcutoff,     /*left cutoff point of the unbounded
density*/
               double linflect,    /*left inflection point*/
               double mode,        /*global maximum of the density*/
               double rinflect,    /*right inflection point*/
               double rcutoff,     /*right cutoff point of the unbounded
density*/
               double error,       /*max. vertical distance between
secant and density in the center point*/
               double rerror,      /*max. value for the proportion
betweeen error and the density*/
               double perror,      /*max. polynomial approximation error
in control points*/
               double *subints,    /*pointer to the vector that will
store subintervals*/
               double *coeffs,     /*pointer to the vector that will
store coefficients*/
               double (*dens)(double a,double b,double c),  /*pointer to
corresponding distribution density function*/
               double (*der1)(double d,double e,double f),  /*pointer to
the first derivative of corresponding distribution density function*/
               double p1,          /*parameter 1*/
               double p2)          /*parameter 2*/
/*creates flexible subintervals with an heuristic method*/
{
      double initsivec[5]={lcutoff,linflect,mode,rinflect,rcutoff};
      int ninit=4;

      double last=initsivec[ninit]; /*upper boundary of questioned
subinterval for dividing*/
      double start=initsivec[0];    /*lower boundary of questioned
subinterval for dividing*/
      int k=ninit-1;
      double aim[MAXTABLESIZE];     /*vector that holds sequenced upper
boundaries of subintervals to be questioned*/
```

```
while (k>=0){
      aim[k]=initsivec[ninit-k];
      k--;
}

k=ninit-1;
int i=0,j,z;
double erquest,rerquest,grad,cons;
double xpts[PTS],ypts[PTS],coeff[PTS],expts[PTS-1],diff,maxdiff;

subints[i]=initsivec[0];

while (subints[i]!=last){
      erquest=fabs((*dens)((aim[k]+start)/2,p1,p2)-
      ((*dens)(start,p1,p2)+(*dens)(aim[k],p1,p2))/2);
      rerquest=erquest/(*dens)((aim[k]+start)/2,p1,p2);

      if (erquest<=error && rerquest<=rerror){
            ChebyPoints(start,aim[k],xpts);
            ChebyExtrema(start,aim[k],expts);
            if (aim[k]<linflect || aim[k]>rinflect ||
            start<linflect || start>rinflect){
                  if ((*dens)(start,p1,p2)<=(*dens)(aim[k],p1,p2)){
                        grad=(*der1)(start,p1,p2);
                        cons=(*dens)(start,p1,p2)-grad*start;
                  }
                  else{
                        grad=(*der1)(aim[k],p1,p2);
                        cons=(*dens)(aim[k],p1,p2)-grad*aim[k];
                  }
            }
            else{
                  grad=((*dens)(aim[k],p1,p2)-
                  (*dens)(start,p1,p2))/(aim[k]-start);
                  cons=(*dens)(aim[k],p1,p2)-grad*aim[k];
            }

            for (j=0;j<PTS;j++){
                  ypts[j]=(*dens)(xpts[j],p1,p2)-
                  (grad*xpts[j]+cons);
            }

            Interpol(xpts,ypts,coeff);
            maxdiff=0;
            for (j=0;j<PTS-1;j++){
                  diff=fabs((*dens)(expts[j],p1,p2)-
                  (grad*expts[j]+cons+Pol(expts[j],coeff,PTS)));
                  if (maxdiff<diff){
                        maxdiff=diff;
                  }
            }

            if (maxdiff<=perror){         /*accept subinterval*/
                  i++;
                  subints[i]=aim[k];
                  for (z=0;z<PTS;z++){
                        coeffs[PTS*(i-1)+z]=coeff[z];
                  }
                  start=subints[i];
                  k--;
```

```
                }
                else{       /*reject and divide subinterval*/
                      k++;
                      aim[k]=(start+aim[k-1])/2;
                }
          }
          else {
                k++;
                aim[k]=(start+aim[k-1])/2;
          }
    }
    return i;
}

void GuideTable(int length,double *cumvec,int *outputvec)  /*produces a
guide table, double size of subinterval vector*/
{
    outputvec[0]=0;
    int i=0;
    int j;

    for (j=1;j<(2*length);j++){
          while (((double)j*cumvec[length-1]/(2*length))>cumvec[i]){
                i++;
          }
          outputvec[j]=i;
    }
}

PDIGEN *Setup(double lcutoff,      /*left cutoff point of the unbounded
density*/
              double linflect,     /*left inflection point*/
              double mode,         /*global maximum of the density*/
              double rinflect,     /*right inflection point*/
              double rcutoff,      /*right cutoff point of the unbounded
density*/
              double error,        /*max. vertical distance between
secant and density in the center point*/
              double rerror,       /*max. value for the proportion
betweeen error and the density*/
              double perror,       /*max. polynomial approximation error
in control points*/
              double (*dens)(double a,double b,double c),  /*pointer to
corresponding distribution density function*/
              double (*der1)(double d,double e,double f),  /*pointer to
the first derivative of corresponding distribution density function*/
              double p1,           /*parameter 1*/
              double p2)           /*parameter 2*/
/*creates the table required by the generation algorithm*/
{
    double subints[MAXTABLESIZE]; /*vector that records approved
subintervals*/
    double coeffs[MAXTABLESIZE*PTS];    /*vector that records approved
coefficients*/
    int i;
    i=FlexSubint(lcutoff,linflect,mode,rinflect,rcutoff,error,rerror,
perror,subints,coeffs,(*dens),(*der1),p1,p2);

    PDIGEN *gen;     /*a structure that holds data table*/
    gen=(PDIGEN *)malloc(sizeof(PDIGEN));
```

```
      gen->sivec=(double *)malloc(sizeof(double)*(i+1));   /*allocation
for subintervals of the data table*/
      gen->cavec=(double *)malloc(sizeof(double)*(i+1));   /*allocation
for cumulative probabilities of the data table*/
      gen->pcvec=(double *)malloc(sizeof(double)*i*PTS);   /*allocation
for polynomial coefficients of the data table*/
      gen->rcvec=(double *)malloc(sizeof(double)*i);       /*allocation
for the cumulative probability of rectangular region in a subinterval*/
      gen->trvec=(double *)malloc(sizeof(double)*i);       /*allocation
for the cumulative probability of triangular region in a subinterval*/
      gen->guidevec=(int *)malloc(sizeof(int)*2*i);        /*allocation
for the guide table of indexed search*/
      gen->n=i;
      gen->mode=mode;
      gen->sivec[0]=subints[0];
      gen->cavec[0]=0;

      int j,k;
      double grad,cons,length,siarea,polyarea[MAXTABLESIZE];
      double sum=0;
      double scale[PTS],xpts[PTS],ypts[PTS],coeff[PTS],intcoeff[PTS+1];
      ChebyPoints(0,1,scale);

      for (j=1;j<=i;j++){/*depending on the property of the subinterval
(convexity, concavity, increasing or decreasing) calculates table
values*/
            gen->sivec[j]=subints[j];
            length=subints[j]-subints[j-1];
            if (subints[j-1]<linflect || subints[j-1]>rinflect ||
            subints[j]<linflect || subints[j]>rinflect){
                  if ((*dens)(subints[j-1],p1,p2)<=
                  (*dens)(subints[j],p1,p2)){
                        gen->rcvec[j-1]=(*dens)(subints[j-1],
                        p1,p2)*length;
                        grad=(*der1)(subints[j-1],p1,p2);
                        cons=(*dens)(subints[j-1],p1,p2)
                        -subints[j-1]*grad;
                        gen->trvec[j-1]=gen->rcvec[j-1]
                        +grad*length*length/2;
                  }
                  else {
                        gen->rcvec[j-1]=(*dens)(subints[j],p1,p2)*length;
                        grad=(*der1)(subints[j],p1,p2);
                        cons=(*dens)(subints[j],p1,p2)-subints[j]*grad;
                        gen->trvec[j-1]=gen->rcvec[j-1]
                        +grad*(-1)*length*length/2;
                  }
            }
            else {
                  grad=((*dens)(subints[j],p1,p2)
                  -(*dens)(subints[j-1],p1,p2))/length;
                  cons=(*dens)(subints[j],p1,p2)
                  -subints[j]*grad;
                  if ((*dens)(subints[j-1],p1,p2)<=
                  (*dens)(subints[j],p1,p2)){
                        gen->rcvec[j-1]=(*dens)(subints[j-1],p1,p2)
                        *length;
                        gen->trvec[j-1]=gen->rcvec[j-1]
                        +((*dens)(subints[j],p1,p2)
                        -(*dens)(subints[j-1],p1,p2))*length/2;
```

```
                  }
                  else {
                        gen->rcvec[j-1]=(*dens)(subints[j],p1,p2)*length;
                        gen->trvec[j-1]=gen->rcvec[j-1]
                        +((*dens)(subints[j-1],p1,p2)
                        -(*dens)(subints[j],p1,p2))*length/2;
                  }
            }

            for (k=0;k<PTS;k++){
                  gen->pcvec[(j-1)*PTS+k]=coeff[k]=coeffs[(j-1)*PTS+k];
            }

            Integ(coeff,intcoeff);
            polyarea[j-1]=Pol(subints[j],intcoeff,PTS+1)
            -Pol(subints[j-1],intcoeff,PTS+1);
            gen->cavec[j]=gen->trvec[j-1]+polyarea[j-1]+sum;
            sum=gen->cavec[j];
      }
      for (j=1;j<((gen->n)+1);j++){
            gen->cavec[j]=gen->cavec[j]/gen->cavec[gen->n];
            siarea=gen->trvec[j-1]+polyarea[j-1];
            gen->rcvec[j-1]=gen->rcvec[j-1]/siarea;
            gen->trvec[j-1]=gen->trvec[j-1]/siarea;
      }

      GuideTable(gen->n,gen->cavec,gen->guidevec);

      return gen;
}
int FreeGen(PDIGEN *gen)      /*frees the memory*/
{
      free(gen->cavec);
      free(gen->pcvec);
      free(gen->guidevec);
      free(gen->rcvec);
      free(gen->sivec);
      free(gen->trvec);
      free(gen);
      return 0;
}
double Generator(PDIGEN *pdigen,
                        int *counter,    /*a vector that holds accepted
variate types*/
                        int countflag)   /*other than zero, it counts*/
/*random variate generation algorithm that works with a created table*/
{
      double u1,u2;
      double urc1,urc2;
      int index;
      double x,x2;
      double coeff[PTS];
      int i;

      u1=U01();   /*a standard uniform variate to generate index*/
      index=IndexSearch(u1,pdigen->guidevec,2*pdigen->n,pdigen->cavec);
/*index generated*/
      urc1=(u1-pdigen->cavec[index])/(pdigen->cavec[index+1]-pdigen
      ->cavec[index]);  /*a new standard uniform variate is recycled to
choose region*/
```

```
      if (urc1<=pdigen->rcvec[index]){    /*recycled variate says
rectangular region*/
            urc2=urc1/pdigen->rcvec[index];     /*a new standard uniform
variate is recycled*/
            if (countflag!=0){counter[0]++;}

            x=pdigen->sivec[index]+(pdigen->sivec[index+1]-pdigen
            ->sivec[index])*urc2;   /*random variate is generated
      uniformly*/
            return x;
      }
      else if (urc1<=pdigen->trvec[index]){    /*recycled variate says
triangular region*/
            urc2=(urc1-pdigen->rcvec[index])/(pdigen->trvec[index]-
            pdigen->rcvec[index]);  /*a new standard uniform variate is
            recycled*/
            if (countflag!=0){counter[1]++;}
            x=pdigen->sivec[index]+(pdigen->sivec[index+1]-pdigen
            ->sivec[index])*urc2;   /*it generated the first uniform
      variate over the subinterval*/
            u2=U01();   /*a standard uniform variate to generate index*/
            x2=pdigen->sivec[index]+(pdigen->sivec[index+1]-pdigen-
>sivec[index])*u2;      /*it generated the second uniform variate over
the subinterval*/
            if (x<=pdigen->mode && x2<=pdigen->mode){      /*increasing
triangular case*/
                  return Max2(x,x2);
            }
            else {      /*decreasing triangular case*/
                  return Min2(x,x2);
            }
      }
      else {      /*recycled variate says polynomial region*/
            urc2=(urc1-pdigen->trvec[index])/(1-pdigen->trvec[index]);
/*a new standard uniform variate is recycled*/
            if (countflag!=0){counter[2]++;}
            for (i=0;i<PTS;i++){
                  coeff[i]=pdigen->pcvec[PTS*index+i];     /*coefficients
are called from the table*/
            }
            return UppSearch(urc2,coeff,pdigen->sivec[index],pdigen
            ->sivec[index+1]);      /*random variate is generated through
      cdf inverse and root finding algorithm*/
      }
}
```

APPENDIX C: APPROXIMATION ERROR ANALYSIS R CODES

Following codes are written in “R” in order to analyze the error behavior with piecewise constants, piecewise linears and piecewise polynomials.

Useful Functions:

```
#Combination Sum: Takes sums of the products of n-element subsets
combsum<-function(vec,n){
if (n>length(vec) | n<=0){
      res<-0;
}
else if (n==1){
      res<-sum(vec);
}
else {
      res<-vec[1]*combsum(vec[-1],n-1)+combsum(vec[-1],n);
}
res;
}

#Sign Matrix: In order to assign signs of D-Matrix, creates a
#coefficient matrix of 1,0 and -1s.
signmat<-function(n){
sign<-c(-1,1);
smat<-matrix(0,n,n);
for (i in 1:n){
      for(j in 1:(n-i+1)){
            smat[i,j]<-sign[j%%2+1]
      }
}
smat;
}

#D-Matrix: Calculates D-Matrix (D Coefficients for polynomial)
dmatrix<-function(xvec,yvec){
lng<-length(yvec);
dtable<-matrix(0,lng,lng);
dtable[1,]<-yvec;
for (i in 2:lng){
      for (j in i:lng){
            dtable[i,j]<-(dtable[i-1,j]-dtable[i-1,j-1])/(xvec[j]-xvec[j-
i+1]);
      }
}
dvec<-0;
for (i in 1:lng){
      dvec[i]<-dtable[i,i];
}
dmat<-matrix(0,lng,lng);
for (i in 1:lng){
      for (j in 1:(lng-i+1)){
```

```
            dmat[i,j]<-dvec[i+j-1];
      }
}
dmat*signmat(lng);
}

#X-Matrix: Takes X-Vector to build a X-Matrix (uses combsum)
xmatrix<-function(xvec){
lng<-length(xvec);
xmat<-matrix(0,lng,lng);
xmat[,1]<-1;
for (i in 1:(lng-1)){
      for (j in 2:(lng-i+1)){
            xmat[i,j]<-combsum(xvec[1:(i+j-2)],j-1);
      }
}
xmat;
}

#Polynomial Calculator: Takes the coefficients and x vector and finds
#p(x)s.
pol<-function(x,coeff){
n<-length(coeff)-1;
res<-0;
for (i in 1:length(x)){
      res[i]<-sum(coeff*(x[i]^(0:n)));
}
res;
}

#Unscaled Chebyshev Points
uschebypoints<-function(n,a=0,b=1){
      i<-n:1;
      x<-cos((2*i-1)*pi/(2*n));
      a+(b-a)*(x+1)/2;
}

#Creation of Chebyshev Points
chebypoints<-function(n,a=0,b=1){
phi<-pi/(2*(n));
i<-0:(n-1);
diff<-sin(2*i*phi)*tan(phi);
a+(b-a)*cumsum(diff);
}

#Creation of Chebyshev Control Points
chebyextrema<-function(g,a=0,b=1){
      0.5*(a+b)+0.5*(b-a)*cos((g:1)*pi/(g+1))/cos(pi/(2*(g+1)));
}

#Newton Polynomial Interpolation
interpol<-function(xvec,      #x-axis values of points
            yvec        #y-axis values of points
){
dmat<-dmatrix(xvec,yvec);
xmat<-xmatrix(xvec);
pmat<-dmat*xmat;
res<-0;
for (i in 1:length(xvec)){
      res[i]<-sum(pmat[i,]);
```

```
}
res;
}

pol<-function(x,coeff){
n<-length(coeff)-1;
res<-0;
for (i in 1:length(x)){
      res[i]<-sum(coeff*(x[i]^(0:n)));
}
res;
}

#First Derivation of Particular PDFs
der1norm<-function(x,mu,sigma){
      (mu-x)*exp(-(x-mu)*(x-
mu)*0.5/(sigma^2))/(2.506628274630963*(sigma^3));
}

der1cauchy<-function(x,x0,gamma){
      2*(x0-x)*gamma/(pi*(x0^2-2*x0*x+x^2+gamma^2)^2);
}

der1exp<-function(x,lambda,null){
      -1*lambda^2*exp(-1*lambda*x);
}

FGamma<-function(x){
      p0<-1.000000000190015;
      p1<-76.18009172947146;
      p2<--86.50532032941677;
      p3<-24.01409824083091;
      p4<--1.231739572450155;
      p5<-1.208650973866179e-3;
      p6<--5.395239384953e-6;

      sum<-
p0+(p1/(x+1))+(p2/(x+2))+(p3/(x+3))+(p4/(x+4))+(p5/(x+5))+(p6/(x+6));

      sum*sqrt(2*pi)*((x+5.5)^(x+0.5))*exp(-1*(x+5.5))/x;
}

der1gamma<-function(x,alfa,beta){
      (beta^alfa)*exp(-1*beta*x)*(x^(alfa-2))*(alfa-1-
beta*x)/FGamma(alfa);
}

FBeta<-function(x,y){
      FGamma(x)*FGamma(y)/FGamma(x+y);
}

der1beta<-function(x,alfa,beta){
      (x^(alfa-2))*((1-x)^(beta-2))*(alfa-1-x*(alfa+beta-
2))/FBeta(alfa,beta);
}
```

Piecewise Constants:

```
PDI0Error<-function(n,
                  error,
                  subintvec=c(-6,-1,0,1,6),
                  dens=dnorm,
                  rvdens=rnorm,
                  p1=0,
                  p2=1){
      subints<-subintvec;
      i<-1;
      j<-1;
      k=length(subints);
      consvec<-0;
      while(subints[i]!=subintvec[5]){
            erquest<-abs(dens(subints[i],p1,p2)-
dens(subints[i+1],p1,p2))/2;
            if (erquest<=error){
                  i=i+1;
                  consvec[i-1]<-(dens(subints[i-
1],p1,p2)+dens(subints[i],p1,p2))/2;
            }
            else{
                  subints[(i+2):(k+1)]<-subints[(i+1):k];
                  subints[i+1]<-(subints[i+2]+subints[i])/2;
                  k=k+1;
            }
      }

      rvs<-rvdens(n,p1,p2);
      exact<-dens(rvs,p1,p2);
      appro<-0;
      for(j in 1:n){
            index<-length(subints[subints<rvs[j]]);
            appro[j]<-consvec[index];
      }
      hist(abs(exact-appro),100);

      rvs2<-runif(n,subintvec[1],subintvec[5]);
      exact2<-dens(rvs2,p1,p2);
      appro2<-0;
      for(j in 1:n){
            index<-length(subints[subints<rvs2[j]]);
            appro2[j]<-consvec[index];
      }
      windows()
      hist(abs(exact2-appro2),100);

      y<-0;
      x<-((subintvec[1]*1000):(subintvec[5]*1000))/1000;
      dx<-dens(x,p1,p2);
      for(j in 1:(length(x))){
            index<-length(subints[subints<x[j]])+(j==1);
            y[j]<-consvec[index];
      }
      windows()
      plot(x,abs(dx-y),pch="*");

      res<-mean(abs(exact-appro)/exact);
      res[2]<-sd(abs(exact-appro)/exact);
      res[3]<-mean(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
      res[4]<-sd(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
```

```
      res[5]<-length(subints)-1;
      res;
}
```

Piecewise Linears:

```
PDI1Error<-function(n,
                  error,
                  subintvec=c(-6,-1,0,1,6),
                  dens=dnorm,
                  rvdens=rnorm,
                  p1=0,
                  p2=1){

      subints<-subintvec;
      i<-1;
      j<-1;
      k=length(subints);
      gradvec<-0;
      consvec<-0;
      while(subints[i]!=subintvec[5]){
            erquest<-abs(dens((subints[i]+subints[i+1])/2,p1,p2)-
((dens(subints[i],p1,p2)+dens(subints[i+1],p1,p2))/2));
            if (erquest<=error){
                  i=i+1;
                  gradvec[i-1]<-(dens(subints[i],p1,p2)-dens(subints[i-
1],p1,p2))/(subints[i]-subints[i-1]);
                  consvec[i-1]<-dens(subints[i-1],p1,p2)-gradvec[i-
1]*subints[i-1];
            }
            else{
                  subints[(i+2):(k+1)]<-subints[(i+1):k];
                  subints[i+1]<-(subints[i+2]+subints[i])/2;
                  k=k+1;
            }
      }

      rvs<-rvdens(n,p1,p2);
      exact<-dens(rvs,p1,p2);
      appro<-0;
      for(j in 1:n){
            index<-length(subints[subints<rvs[j]]);
            appro[j]<-gradvec[index]*rvs[j]+consvec[index];
      }
      hist(abs(exact-appro),100);

      rvs2<-runif(n,subintvec[1],subintvec[5]);
      exact2<-dens(rvs2,p1,p2);
      appro2<-0;
      for(j in 1:n){
            index<-length(subints[subints<rvs2[j]]);
            appro2[j]<-gradvec[index]*rvs2[j]+consvec[index];
      }
      windows()
      hist(abs(exact2-appro2),100);

      y<-0;
      x<-((subintvec[1]*1000):(subintvec[5]*1000))/1000;
      dx<-dens(x,p1,p2);
      for(j in 1:(length(x))){
```

```
            index<-length(subints[subints<x[j]])+(x[j]==subints[1]);
            y[j]<-gradvec[index]*x[j]+consvec[index];
      }
      windows()
      plot(x,abs(dx-y),pch="*");

      res<-mean(abs(exact-appro)/exact);
      res[2]<-sd(abs(exact-appro)/exact);
      res[3]<-mean(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
      res[4]<-sd(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
      res[5]<-length(subints)-1;
      res;
}
```

Piecewise 4th Order Polynomials (Polynomial Density Inversion):

```
PDIError<-function(n,
                  error,
                  rerror,
                  perror,
                  spoints=5,
                  subintvec=c(-6,-1,0,1,6),
                  dens=dnorm,
                  rvdens=rnorm,
                  der=der1norm,
                  p1=0,
                  p2=1
){
      ########
      #Flexible Subinterval Creation
      ########

      subints=subintvec;
      i=1;
      j=1;
      k=length(subints);
      coeffmat<-matrix(0,5,700);
      gradvec=0;
      consvec=0;
      while (subints[i]!=subintvec[5]){

      erquest=abs(((dens(subints[i],p1,p2)+dens(subints[i+1],p1,p2))/2)-
dens((subints[i]+subints[i+1])/2,p1,p2));
            rerquest=erquest/dens((subints[i]+subints[i+1])/2,p1,p2);
            if (erquest<=error & rerquest<=rerror){
                  xpts<-chebypoints(spoints,subints[i],subints[i+1]);
                  expts<-chebyextrema(spoints-1,subints[i],subints[i+1]);
                  grad<-
der(subints[i],p1,p2)*(subints[i+1]<=subintvec[2])+der(subints[i+1],p1,p2
)*(subints[i]>=subintvec[4])+((dens(subints[i+1],p1,p2)-
dens(subints[i],p1,p2))/(subints[i+1]-
subints[i]))*(subints[i]>=subintvec[2])*(subints[i+1]<=subintvec[4]);
                  cons<-(dens(subints[i],p1,p2)-
subints[i]*grad)*(subints[i]<subintvec[3])+(dens(subints[i+1],p1,p2)-
subints[i+1]*grad)*(subints[i]>=subintvec[3]);
                  ypts<-dens(xpts,p1,p2)-(grad*xpts+cons);
                  coeff<-interpol(xpts,ypts);

                  diff=abs(dens(expts,p1,p2)-
(grad*expts+cons+pol(expts,coeff)));
```

```
                  if(max(diff)<=perror){
                        i=i+1;
                        coeffmat[,(i-1)]<-coeff;
                        gradvec[i-1]<-grad;
                        consvec[i-1]<-cons;
                  }
                  else{
                        subints[(i+2):(k+1)]<-subints[(i+1):k];
                        subints[i+1]<-(subints[i+2]+subints[i])/2;
                        k=k+1
                  }
            }
            else {
                  subints[(i+2):(k+1)]<-subints[(i+1):k];
                  subints[i+1]<-(subints[i+2]+subints[i])/2;
                  k=k+1
            }
      }

      rvs<-rvdens(n,p1,p2);
      exact<-dens(rvs,p1,p2);
      appro<-0;
      appro2<-0;
      y<-0;
      for(j in 1:n){
            index<-length(subints[subints<rvs[j]]);
            appro[j]<-
gradvec[index]*rvs[j]+consvec[index]+pol(rvs[j],coeffmat[,index]);
      }
      windows();
      hist(abs(exact-appro),100);
      rvs2<-runif(n,subintvec[1],subintvec[5]);
      exact2<-dens(rvs2,p1,p2);
      for(j in 1:n){
            index<-length(subints[subints<rvs2[j]]);
            appro2[j]<-
gradvec[index]*rvs2[j]+consvec[index]+pol(rvs2[j],coeffmat[,index]);
      }
      windows()
      hist(abs(exact2-appro2),100);
      x<-((subintvec[1]*1000):(subintvec[5]*1000))/1000;
      dx<-dens(x,p1,p2);
      for(j in 1:(length(x))){
            index<-length(subints[subints<x[j]])+(j==1);
            y[j]<-
gradvec[index]*x[j]+consvec[index]+pol(x[j],coeffmat[,index]);
      }
      windows()
      plot(x,abs(dx-y),pch="*");
      res<-mean(abs(exact-appro)/exact);
      res[2]<-sd(abs(exact-appro)/exact);
      res[3]<-mean(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
      res[4]<-sd(abs(exact2-appro2)*(subintvec[5]-subintvec[1]));
      res[5]<-length(subints)-1;
      res;
}
```

REFERENCES

Ahrens, J. H., 1995, *A One-table Method for Sampling from Continuous and Discrete Distributions*, Computing 54(2), 127-146.

Brent, R. P., 1973, *Algorithms for Minimization without Derivatives,* Prentice-Hall Inc, Englewood Cliffs, New Jersey.

Burden, R. L. and J. D. Faires, 1997, *Numerical Analysis*, Brooks/Cole Publishing Company, 6th Edition.

Chen, H. C. and Y. Asau (1974), *On Generating Random Variates from An Empirical Distribution,* AIIE Trans. 6, 163-166.

Hurley, W. J. and W. S. Andrews, 2007, *Normal Approximation to A Sum of Geometric Random Variables with Application to Ammunition Stockpile Planning,* Defence Science Journal, Vol. 57, No. 5, 733-737.

Hörmann, W., 1995, *A Rejection Technique for Sampling from T-concave Distributions,* ACM Trans. Math. Software 21(2), 182-193.

Hörmann, W., 2007, *Monte Carlo Simulation in Finance*, Lecture Notes, Bogazici University.

Hörmann, W. and J. Leydold, 2007, *Sampling from Linear Multivariate Densities,* Tech. Rep., Department of Statistics and Mathematics, WU Wien

Hörmann, W., J. Leydold, and G. Derflinger, 2004, *Automatic Nonuniform Random Variate Generation,* Berlin Heidelberg: Springer-Verlag.

Karawatzki, R., 2006, *The Multivariate Ahrens Sampling Method,* Tech. Rep. 30, Department of Statistics and Mathematics, WU Wien.

Runge, C., 1901, *Über Empirische Funktionen und die Interpolation Zwischen Äquidistanten Ordinaten,* Zeitschrift für Mathematik und Physik 46, 224-243.

REFERENCES NOT CITED

Leydold, J. and W. Hörmann, 2006, *Black-box Algorithms for Sampling from Continuous Distributions*, Winter Simulation Conference.

Maindonald, J. H., 2004, *Using R for Data Analysis and Graphics, Introduction, Code and Commentary,* Centre for Bioinformation Science, Australian National University.

Verzani, J., 2002, *Using R for Introductory Statistics,* The CSI Math Department.

http://en.wikipedia.org/wiki/Bisection_method.html

http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm.html

http://en.wikipedia.org/wiki/False_position_method.html

http://en.wikipedia.org/wiki/Secant_method.html

http://en.wikipedia.org/wiki/Triangular_distribution.html