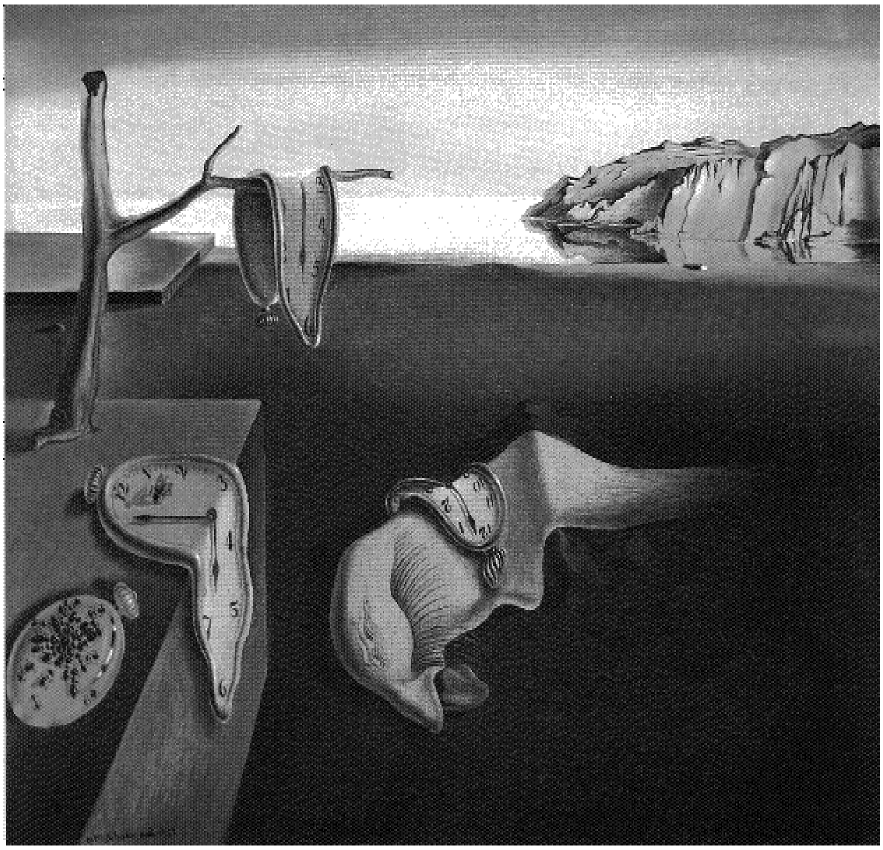
Institut für Automation
Abt. für Automatisierungssysteme

Technische Universität Wien

Projektbericht Nr. 183/1-92
February 1999

Random Generators

Gerald Hummel and Bettina Weiss



Salvador Dali, "Die Beständigkeit der Erinnerung"

Random Generators

Gerald Hummel*
Bettina Weiss†

February 26, 1999

Abstract

Our project SynUTC[1] is devoted to the development of soft- and hardware for very high precision/accuracy clock synchronization in fault-tolerant distributed systems. In the course of this project, the simulation system SimUTC was developed, which allows to execute and evaluate clock synchronization algorithms [Wei99]. However, in order to simulate the indeterministic nature of the hardware we need random generators that allow us to model the transmission delay characteristics of a network or the behaviour of clock drifts. The goal of this work[2] is to implement such generators in the programming language C++.

*Programming and documentation in German (Word).

†Conversion into TeX-document, translation into English, and some modifications and additions to the original document.

[1]The SynUTC-project (*http://www.auto.tuwien.ac.at/Projects/SynUTC/*) has received funding from the Austrian Science Foundation (FWF) grant P10244-ÖMA, the OeNB "Jubiläumsfonds-Projekt" 6454, the BMfWV research contract Zl.601.577/2-IV/B/9/96, and the Austrian START programme Y41-MAT.

[2]Carried out by Gerald Hummel at the Department of Automation in the course of an Informatikpraktikum 1.

1 Introduction

In order to generate a random variable according to a given distribution function, it is first necessary to compute a *uniform* random variable in the interval [0, 1]. Such generators are well documented, see e.g. [Knu81]. We use the uniform random generator provided by the authors of C++SIM (a simulation toolkit, cf. [LM]), see section 2.1.

To get a *non-uniform* distribution within a given interval, first we need to specify the destination interval [`low`, `high`]. Our simulation requires intervals with `low` ≥ 0.

Many distribution functions are defined within an infinite interval, but the computer can only represent values within a finite range. So we have to reduce the original infinite range to a finite "computation interval" which can be represented with the chosen data type, like [1E-308, 1E+308] when using data type `double`. Now when the user requests a random value within a given interval [`low`, `high`], then this interval is a subset of the finite range. The parameters of the distribution, which lie within the user interval, are first mapped to the computation interval. The random value is then computed for this interval and finally mapped back into the user interval.

Before we start explaining the functions, we need to introduce a few commonly used terms: The behaviour of our random generators can be described by a *probability density function (PDF)*. It gives the probability with which a certain value occurs. In order to obtain random variables, however, one has to consider the *cumulative distribution function (CDF)*, which is the integral of the PDF.

Functions are characterized by the *expectation* E, which is the mean of the random values, and in the case of the normal distribution by the *standard deviation* ρ as well.

When computing the distribution, it is best to use the *transformation method* which computes the random values from the inverse function of the CDF. The method utilizes the fact that since a given distribution function F maps an X-distributed random value from the range $[a, b]$ to the uniform range $[0, 1]$, its inverse function maps uniform values from $[0, 1]$ to X-distributed values in $[a, b]$. So we only have to draw a uniform random value, use it as input to the inverse CDF, and the result will be an X-distributed random value. This is a deterministic method which always needs to compute exactly one uniform value to obtain the result.

The transformation method is limited by the fact that it requires the existence and computability of the inverse function of the CDF. If these conditions are not fulfilled, one has to use some kind of *rejection method*, which tries to approximate the inverse function. It uses an appropriate "enclosing function" and a uniform value to compute a random value x. Then, it uses a second uniform value u

and tests if $PDF(x) < u$. If the condition is fulfilled, then x is returned and conforms to the desired distribution. If the test fails, x is discarded and the process is repeated with a new value. Obviously, the algorithm needs to compute an arbitrary number of values until one passes the test. This makes the method less useful for our simulation, because the execution time of calls to the random generators will vary, which is not desired.

In Section 2 we will explain the distribution functions and compare the ideal function to the distributions obtained by our random generators. Section 3 is devoted to a description of the test program we have used to create the plots of the generators' behaviour. The final Section 4 describes the C++ library that contains the implementation of the generators.

2 Distribution Functions

2.1 Uniform

As already mentioned, we have taken over the uniform generator of [LM], which is a combination of two methods:

- The first method uses a *multiplicative generator* to fill an array of fixed length with random values. These values are computed from a given seed $MSeed$, which is changed at every access to the generator according to the formula

  $$MSeed_{new} = (MSeed_{old} * 3125) \bmod 67108864.$$

  The initial value of $MSeed$ is saved in case the user wants to reset the generator to its initial state at some later time.

- The second generator uses a *linear congruential generator* [Knu81] to compute an index into the array. It also needs a seed $LSeed$ which is changed at every access according to the formula

  $$LSeed_{new} = (b * LSeed_{old} + 1) \bmod m,$$

  with $m = 100000000$ (modulus) and $b = 31415821$ (multiplier).

  The value of the seed is mapped to an array index to choose the random value that is returned to the user. This value is then replaced with a new one that is obtained from the multiplicative generator.

  Again, the initial seed is saved.

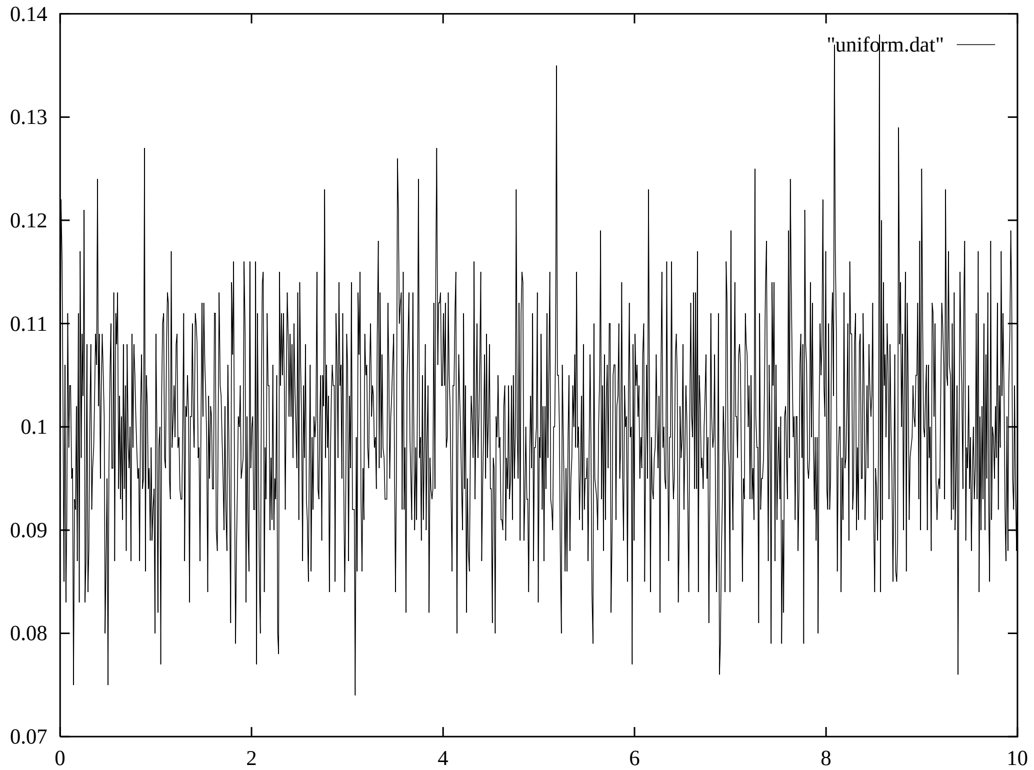


Figure 1: *Measured PDF of the Uniform Distribution*

So uniform deviates are generated by the multiplicative generator, but additionally shuffled by the linear congruential generator, thus (hopefully) causing the resulting sequence to be even more random.

Fig. 1 shows the behaviour of our uniform generator. Ideally, the PDF should be the constant function $f(x) = 0.1$. The real function fluctuates around 0.1, with some values having a slightly higher probability, and with some being less probable. However, we have compared this generator with the uniform generator implemented in [PF88] and it shows a significantly better overall behaviour. Thus, we have chosen it as the basis of our generators.

In order to be able to reset the random generator to a former state, we have modified the original code so that it saves the access parameters $MSeed$ and $LSeed$ that correspond to the current state of the generator. The user can obtain these parameters at any time and can restore this state at some later time.

Internally, we need to keep track of the number of accesses (*distance*) that the user has made, up to the array size, and we always keep the seeds that have been used *distance* accesses ago. These seeds and their *distance* comprise the state information that is returned to the user. When the user restores a state, we check the value of *distance*. If it is less than the array size, we fill the array using the initial $MSeed$ and then draw and discard $distance$ values to get to the current state. If $distance$ is equal to the array size, we just restore the seeds and draw and discard $distance$ values to get to the current state. Note that the values we draw have nothing to do with the restored state, they still belong to the former state of the generator. But after they were drawn, the multiplicative generator has filled the array with exactly the same values that it had when the user has saved the state. So the next user access to the random generator will result in the same value as was returned by the first access after the state had been saved.

2.2 Exponential

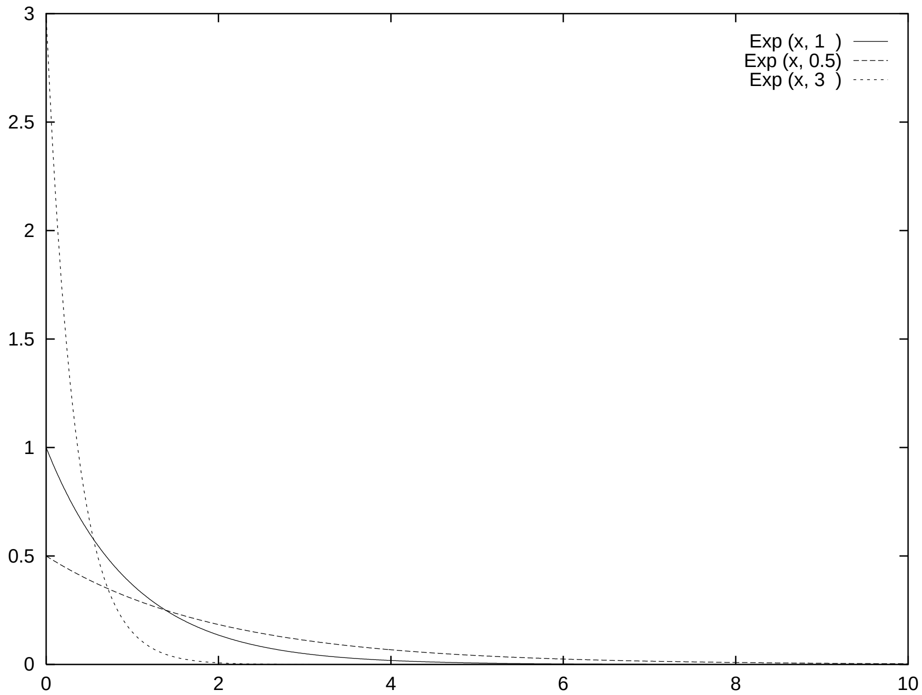


Figure 2: *Ideal PDF of the Exponential Distribution, 2nd Param.* λ

The exponential function is characterized through the following PDF (see Fig. 2)

$$f(x) = \lambda * e^{-\lambda * x}.$$

The CDF and the corresponding inverse function are

$$F(x) = 1 - e^{-\lambda * x},\ F^{-1}(y) = -\frac{ln(1-y)}{\lambda}$$

with the expectation $E = 1/\lambda$.

Since the inverse of the CDF is available, we utilize the transformation method to obtain the random value. We use a uniform random variable y as input to the inverse function $F^{-1}(y)$ to get the corresponding x. The result of $F(y)$ follows an exponential distribution.

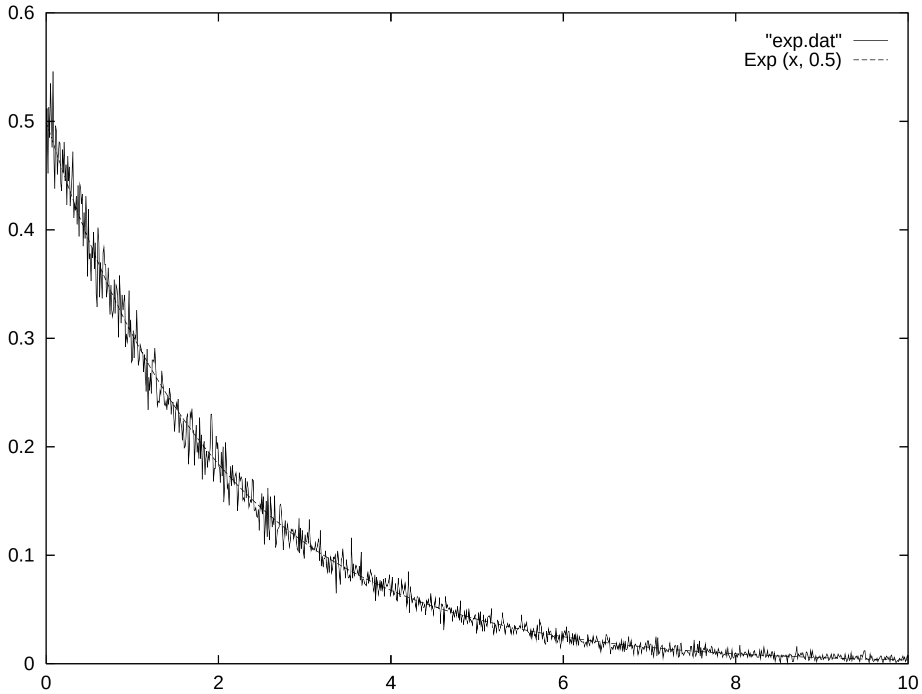


Figure 3: *Measured PDF of the Exponential Distribution,* $E = 2$

The implementation is taken from [PF88] but modified to draw random values within a given interval [`low`, `high`] which conform to a given expectation E.

Fig 3 compares the ideal PDF for $E = 2$ within the interval $[0,\ 10]$ against the PDF of the exponential generator. The ideal function is nearly invisible behind the real function, so the functions are closely together.

2.3 Normal

The normal distribution is marked by its symmetrical, bell-shaped density function (see Fig. 4)

$$f(x) = \frac{1}{\sigma * \sqrt{2 * \pi}} * e^{-\frac{1}{2} * (\frac{x-\mu}{\sigma})^2},\ \ \sigma > 0,$$

with the expectation $E = \mu$ and the standard deviation σ.

The generator uses the *Box-Muller Algorithm*: first, two uniform random values are obtained, which are taken as the cartesian coordinates of a two-dimensional point. In order to be usable, the uniform values must first pass

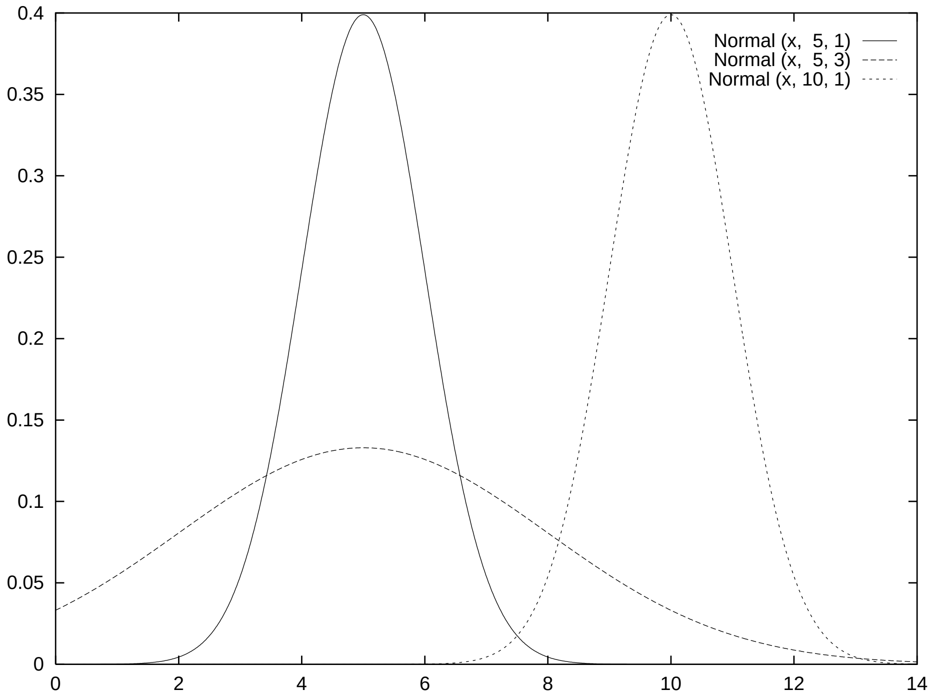


Figure 4: *Ideal PDF of the Normal Distribution, 2nd Param. μ, 3rd Param. σ*

a test: if the distance r of the point from the origin is less than 1, then the random values can be further used to compute the normal deviate. Otherwise, they are discarded and two new values are drawn. Using the Box-Muller algorithm, a surviving pair of uniform deviates is transformed into values conforming to a normal distribution with $\mu_0 = 0$ and $\sigma_0 = 1$, so we have to multiply the values with σ and add μ to obtain random values with the mean μ and standard deviation σ. From these two values, we return the first and store the second in case a future test fails.

Our implementation is based on [PF88], but allows an arbitrary mean and standard deviation. Note that we do not scale the interval: if a (valid) result does not fall into the desired range, then it is discarded and a new result is computed. Since this may theoretically result in an endless loop, we keep a counter that is set to zero upon function entry and which is incremented if the value has to be discarded. If the counter reaches a given upper limit, then we abort the program, since this means that something is seriously wrong. In such a case, one should check the parameters of the algorithm to determine the problem. In our tests, however, we have never encountered such emergency aborts.

Fig. 5 compares the ideal PDF for $\mu = 0.3$ and $\sigma = 0.06$ within the interval $[0.2,\ 0.5]$ against the PDF of the generator. Again, the ideal function is nearly invisible behind the real function. However, it seems to be smaller than the measured PDF, because it is defined within an infinite range, whereas the measured

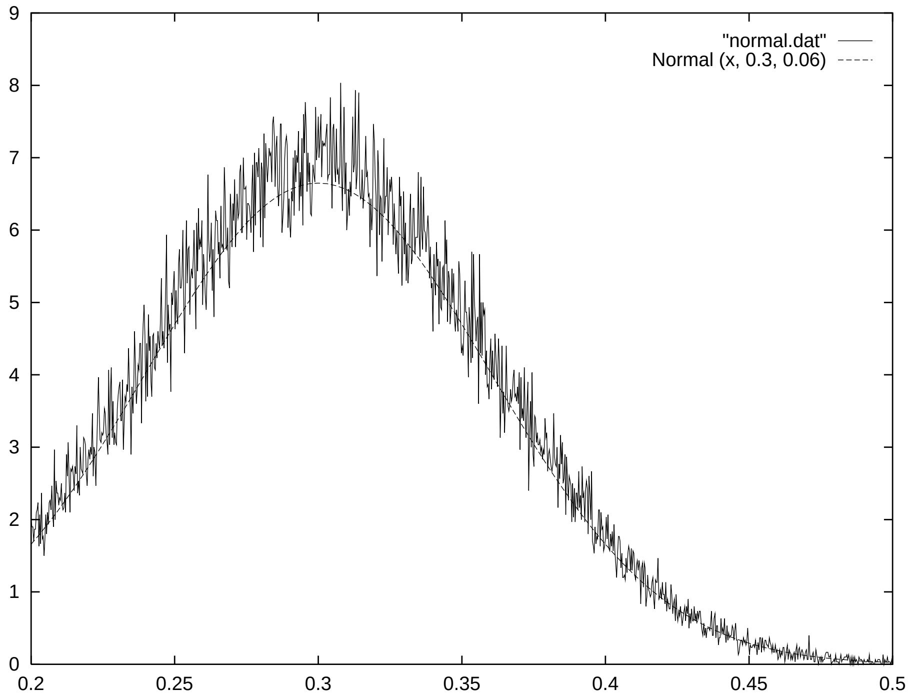


Figure 5: *Measured PDF of the Normal Distribution, $\mu = 0.3$, $\sigma = 0.06$*

PDF is only defined in the target interval.

2.4 Gamma

This function looks exponential for an expectation near the interval bounds and is bell-shaped, albeit asymmetrical, for all other expectation values (see the first three functions in Fig. 6). Its PDF is

$$f(x) = \frac{x^{\alpha-1} * e^{-x}}{\Gamma(\alpha)}, \;\; \alpha, x > 0.$$

The function has to be computed with the rejection method. For this method we first need an "enclosing" function which should be greater or equal to the density function and which should be computable. Our implementation uses the algorithm in [PF88], which is based on the *Lorentzian distribution*. The comparison function is given by

$$f(x) = (1 + y^2) * e^{(\alpha-1)*ln(\frac{x}{\alpha-1})-s*y}$$

where

$$y = tan(\pi * U), \; s = \sqrt{2 * (\alpha - 1) + 1}, \text{ and } x = s * y + \alpha - 1.$$

Parameter U is the uniform deviate, and the expectation is $E = \alpha$.

To compute the desired random value, we split α into an integral part $\lfloor\alpha\rfloor$ and a fractional part $\alpha - \lfloor\alpha\rfloor$, compute gamma-distributed deviates for each of the values and finally add these deviates to get the desired value.

For the integral part, we use the algorithm presented in [PF88]. We first draw a uniform deviate and use the enclosing function to compute the $f(x)$ value. Then, we draw a second uniform deviate. If this second value is less than $f(x)$ then we use the computed x, otherwise we have to repeat the process.

For the fractional part, we use the algorithm presented in the exercises of [Knu81]. It also works with rejection but uses a different enclosing function.

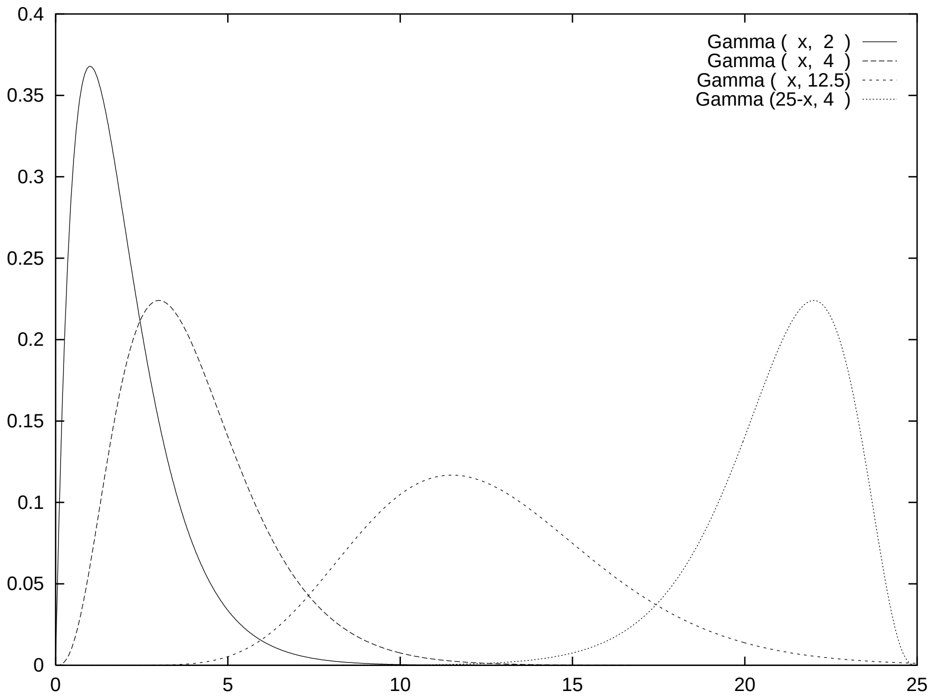


Figure 6: *Ideal PDF of the Gamma Distribution, 2nd Param.* α

We have modified the original generator to draw random values within a given interval [`low`, `high`]. This has posed unexpected problems, since the obvious method of scaling the intervals does not work. This is due to the properties of the gamma distribution, which has a highly asymmetric appearance if the expectation is "near" zero, but gets flatter and more symmetrical if E is large. So if we just scale our target interval to a fixed and large computation interval, then we will also scale the target E to a large value, and the resulting gamma distribution will loose its asymmetrical shape. However, it is this asymmetry that makes the gamma distribution particularly interesting. So we had to reduce the computation interval to a size that still keeps this property. Plots of different gamma distributions have led us to the computation interval [0, 25]. This interval

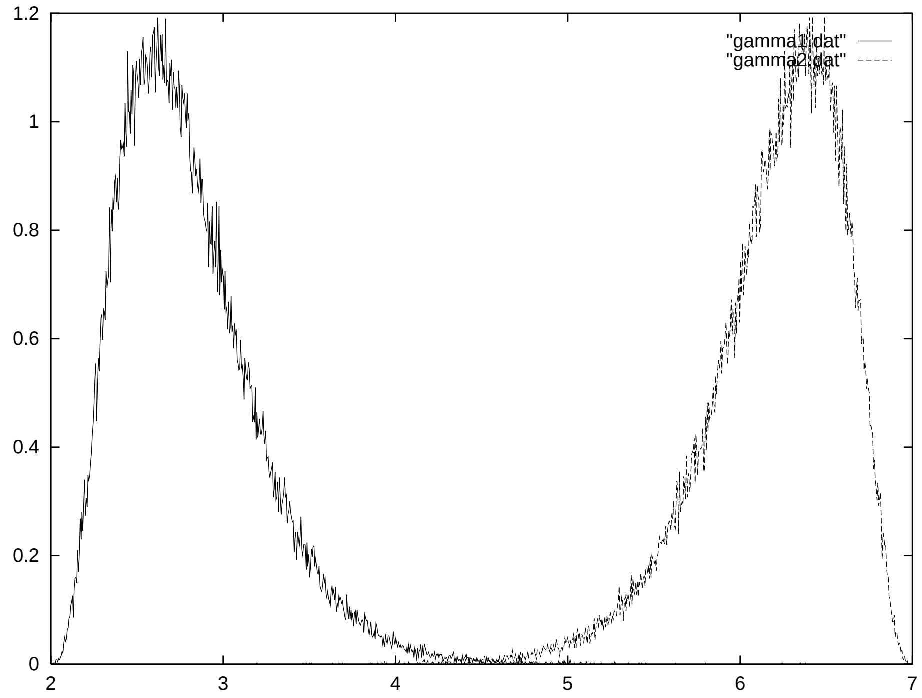


Figure 7: *Measured PDF of the Gamma Distribution,* $E_1 = 2.8, E_2 = 6.2$

is large enough to make values beyond the upper bound quite unlikely (such values are rejected and the computation is repeated), but small enough to make functions with an expectation within this range sufficiently asymmetrical. As a second modification, we differentiate between an expectation that is within the left half of the interval, and an expectation that is within the right half. If E is in the right half of the interval [`low`, `high`], then we use the expectation $high - E$ in our computations, that is, we "mirror" the expectation into the lower part of the interval. For this new E, we compute the gamma distribution, and then we mirror the result back again. Fig. 6 shows the ideal probability functions. Function $Gamma(25 - x, 4)$ plots the desired result of a call with $E = 21$. It is the mirror of function $Gamma(x, 4)$.

Fig. 7 shows the results of two calls to the generator with the target interval [2, 7] and the expectations $E = 2.8$ and $E = 6.2$. The results should be comparable in shape to the ideal plots with expectations $E = 4$ and $E = 21$. The real functions have higher y-values, however, since the functions only exist in the target interval, whereas the ideal functions are defined within an infinite interval.

2.5 Weibull

This distribution function has a characteristic asymmetrical bell-shape that depends on the location of the expectation within the interval (see Fig. 8). The PDF is

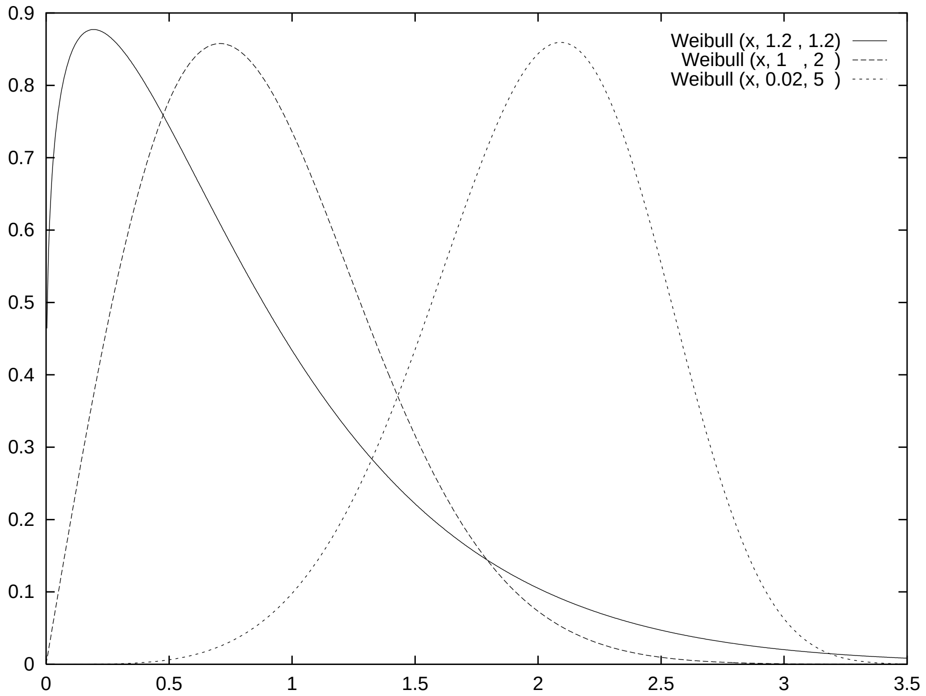


Figure 8: *Ideal PDF of the Weibull Distribution, 2nd Param. λ, 3rd Param. α*

$$f(x) = \lambda * \alpha * x^{\alpha-1} * e^{-\lambda * x^{\alpha}}.$$

The CDF and its inverse functions are

$$F(x) = 1 - e^{-\lambda * x^{\alpha}},\ F^{-1}(y) = \sqrt[\alpha]{\frac{-ln(1-y)}{\lambda}}.$$

The expectation is given by

$$E = \frac{1}{\lambda}^{\frac{1}{\alpha}} * \Gamma(1 + \frac{1}{\alpha})$$

and the standard deviation is defined through

$$S = \frac{1}{\lambda}^{\frac{2}{\alpha}} * [\Gamma(1 + \frac{2}{\alpha}) - (\Gamma(1 + \frac{1}{\alpha}))^2].$$

Since the CDF and its inverse function are known, we use the transformation method. Parameter λ is a scaling parameter and is computed from the given interval bounds. Parameter α determines the form of the function and is taken from a table which contains a list of pairs $\langle E, \alpha \rangle$ that cover the range of E. The user specifies E and the corresponding α is obtained from the table. Fig. 9 shows the result for the interval $[0, 10]$ with the expectation $E = 7$. The ideal function that is plotted against the measured one has been obtained by experimenting with the parameters until the shape and location was approximately the same.

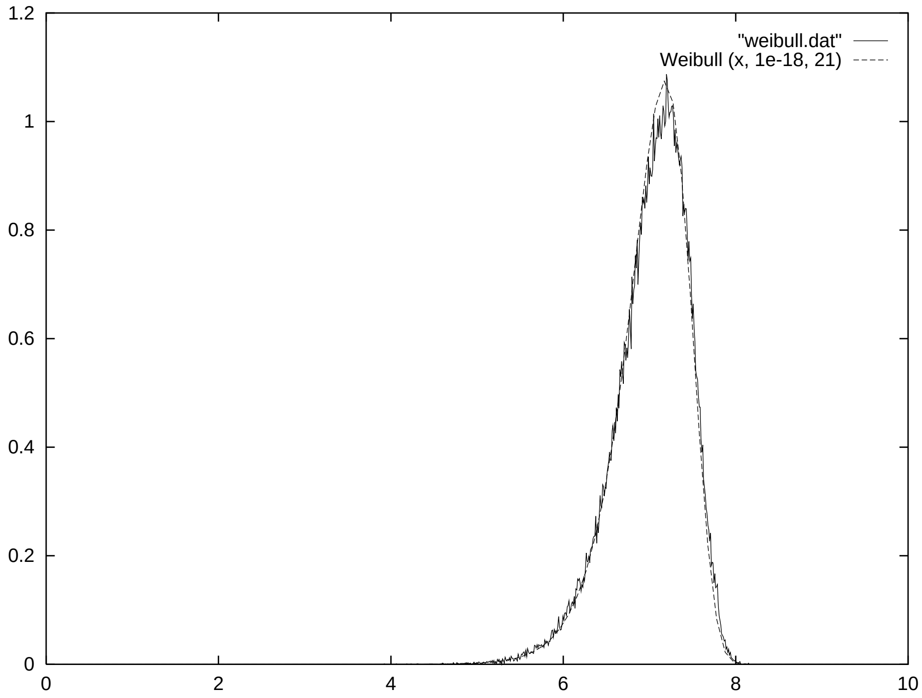


Figure 9: *Measured PDF of the Weibull Distribution,* $E = 7$

Stepping through the code would have yielded the exact values, but setting $\lambda = 10^{-18}$ and $\alpha = 21$ resembles the generated function pretty good.

For the implementation of the Γ function, we have used the source code of the plot program "gnuplot". See [TKS82] for further information on the Weibull distribution.

2.6 Generic

Here, the user can obtain any distribution function, provided that its inverse CDF is passed as a table of $\langle F(x), x\rangle$ values, where x is the random value and $F(x)$ the corresponding CDF. The table must be sorted by $F(x)$. To obtain a random value, the algorithm computes a uniform deviate, searches the table for the $F(x)$ that best matches this value and returns the corresponding x. This is like the transformation method but uses a pre-computed inverse CDF instead of computing it online.

To demonstrate the generic function, we have added two normal distributions with $\mu_1 = 3$ and $\sigma_1 = 0.5$ and $\mu_2 = 6$ and $\sigma_2 = 1$ to get a function with two peaks. For this function, we have computed the table of the inverse CDF for the interval $[0, 10]$ and loaded it into our random generator. Fig. 10 shows the resulting PDF

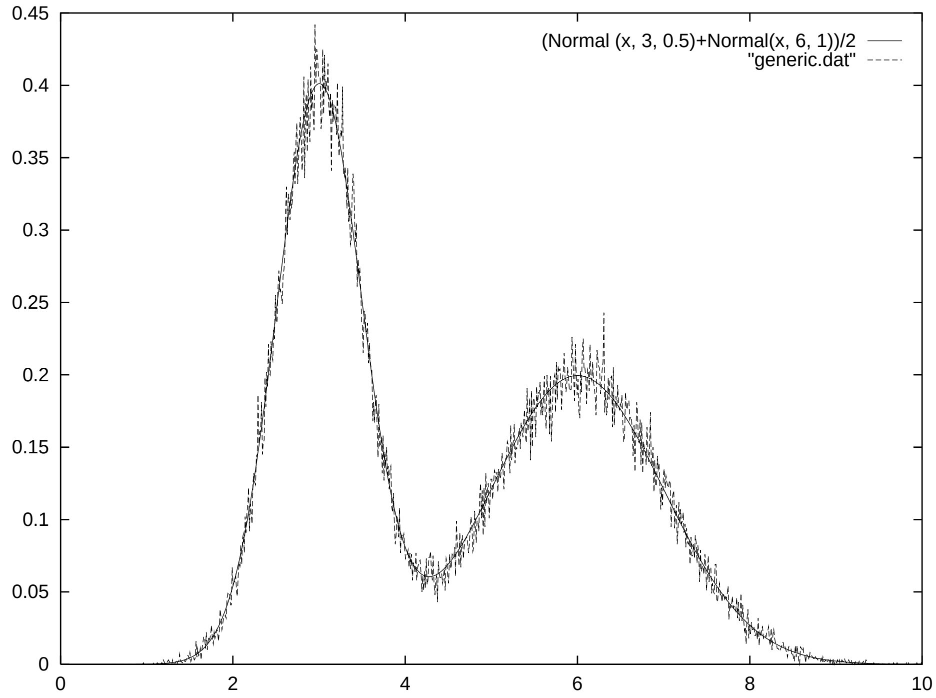


Figure 10: *Measured PDF of the Generic Distribution*

of the generator plotted against the ideal PDF. For the latter, we have divided the sum of the normal distributions by two to normalize the resulting PDF.

The table was generated by plotting the ideal PDF into a file, incrementing x in steps of 0.01. This file was further processed to compute the corresponding CDF by adding the y-values of the PDF. With this data we have then created the table of the inverse CDF. When generating such a table, the incremental value for x must be chosen with care. If the value is too small, then you will get “holes” in the resulting PDF. If it is large, the PDF of the generator will be good, but the table will be large and searching it will take much time.

In any case, you should be aware that the generator searches its table for computing the entries. The number of table accesses until a matching value is found will vary with the CDF, but since we use a linear search method, the value is proportional to the number of pairs in the table. So the generator should certainly not be used in a real-time environment.

3 Test Program

For evaluating the performance of our random generators, we use the test program `randtst`. It lets the user choose from a list of available distribution functions and asks for the mean, for the standard deviation (if the normal distribution was selected), and for the lower and upper bound of the target interval. Then it prompts the user to enter the number of deviates that should be computed and to specify the number of equivalence classes into which the target interval should be parted[3].

The program then draws the random deviates, sorts these values into the equivalence classes and keeps track of the number of values that have fallen into each class. After the desired number of deviates has been drawn, the number of elements in each equivalence class is multiplied by the number of classes and divided by the product of the total number of values drawn times the interval range, and is then printed to the file `ranvars.dat` in gnuplot format. Therefore, we can use the plot program `gnuplot` to generate a graphical representation of the results. The plotted function should approximate the probability density function. Note that the quality of the resulting function depends on the number of samples used for the plot and on the choice of the parameters (since we use only finite intervals, we introduce errors if the expectation is too near to the bounds). The width of the equivalence classes also influences the result. When testing a function, you will have to experiment to find a good choice for the number of samples and the number of equivalence classes.

For testing the generic random generator, we have developed a second program `cdfgen` to generate the inverse CDF from a gnuplot file. You have to follow these steps to generate the table:

1. Use gnuplot to create your desired PDF. For our example in section 2.6, we have generated the PDF with the commands

   ```
   Normal (x, m, s) = 1/(s*sqrt(2*pi))*exp(-0.5*((x-m)/s)**2)
   plot [0:10] Normal (x, 3, 0.5)+Normal(x, 6, 1)
   ```

2. Print the PDF to a file. You should pay attention to the granularity of the x-value, since it influences the quality of the resulting CDF. For our example, we have used the gnuplot commands

   ```
   set samples 1000
   set term table
   set output "generic.dat"
   plot [0:10] Normal (x, 3, 0.5)+Normal(x, 6, 1)
   ```

   to print a gnuplot table of the PDF.

3. Compute the factor $div = \int PDF(x)dx$ by which one has to divide the CDF to normalize it. The factor is $div = l_{interval} * n_{func}$ where $l_{interval}$ is the interval range and n_{func} is the number of PDFs you have added to get your customized PDF.

[3]The equivalence classes are used to part the interval into a finite number of intervals of the same length. Whenever a random deviate is drawn, it is attributed to the equivalence class whose interval contains the value

4. Call `cdfgen -d` *div*. The program will read the file `generic.dat` and generate two files `generic_cdf.dat` and `gentab.hpp`. The former contains the inverse CDF in gnuplot format, the latter a table in C++ format that is included by `randtst` (you have to compile again to include it).

4 C++ Library

The distribution classes are contained in the files `distrib.hpp` and `distrib.cpp` which can be found in the SimUTC directory `SimUTC/Sources/Includes` (header file) and `SimUTC/Sources` (implementation file). The test program `randtst` and the generic function table generator `cdfgen` are located in `SimUTC/Sources/-Tests`.

4.1 distrib.hpp

This is the header file and declares several classes:

distributions: Provides the uniform generator used to obtain the uniform values that are needed by the algorithms. The class is used as a base class for all other random classes. It should not be used directly.

CUniform: Uniform random generator. Users should always declare their uniform generators from this class, never from `distributions`.

CExponential: Exponential random generator.

CNormal: Normal random generator.

CGamma: Gamma random generator.

CWeibull: Weibull random generator.

CGeneric: Generic random generator with function table.

CRandom: Access class that combines all random generators and lets the user dynamically choose the distribution.

4.2 distrib.cpp

This is the implementation file. As already mentioned, access to the generators is co-ordinated by the class `CRandom`. The following member functions of `CRandom` enable the user to access the single generators:

double RandomVar (EDistribution dist): Obtains a random number with the distribution *dist*.

`dist`: Uniform | Exponential | Normal | Gamma | Weibull | Generic

double RandomVar (EDistribution dist, double c, double low, double high, double mean, double free_par): Obtains a random number with the distribution *dist*, but first sets the parameters of the distribution.

Note that this function cannot be used for distributon `Generic`.

`dist`: Uniform | Exponential | Normal | Gamma | Weibull

`c`: a constant that is added to the random value before it is returned

`low`: lower interval bound

`high`: upper interval bound

`mean`: expectation E

`free_par`: a free parameter, its meaning depends on the distribution (e.g., it is ignored for `Exponential` and is taken as the standard deviation for `Normal`.

void Set (EDistribution dist, double c, double low, double high, double mean, double free_par): Sets the parameters of the distribution without drawing a random value. Use this function to preload parameters that should stay fixed for several accesses to the generator and use `RandomVar(EDistribution)` to obtain the random values.

The parameters have the same meaning as in the function above.

void Set_generic (double c, double low, double high, double *inv_funct, unsigned long tabsize): This loads the function table `inv_funct` into the generic random generator and at the same time sets the interval bounds and the additive constant.

`inv_funct`: pointer to the first element in the function table; the table consists of pairs $(F(x), x$ of `double` values that are sorted by $F(x)$.

`tabsize`: number of pairs in the function table; note that this is *not* the size of the table (the table has size $2 * tabsize$).

void GetSeeds (EDistribution dist, long& mgSeed, long& lcgSeed, long& distance) const: Obtains the current state of a particular random generator.

`dist`: Uniform | Exponential | Normal | Gamma | Weibull | Generic

`mgSeed`: seed of the multiplicative generator

`lcgSeed`: seed of the linear congruential generator

`distance`: distance of the seeds

void SetSeeds (EDistribution dist, long mgSeed, long lcgSeed, long distance): Sets the given generator to a former state.

`dist:` Uniform | Exponential | Normal | Gamma | Weibull | Generic

`mgSeed:` seed of the multiplicative generator

`lcgSeed:` seed of the linear congruential generator

`distance:` distance of the seeds

Take care only to use a triple of seeds and distance values that has been obtained by an earlier call to `GetSeeds()`.

References

[Knu81] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[LM] M.C. Little and D.L. McCue. Construction and use of a simulation package in C++. http://cxxsim.ncl.ac.uk/homepage.html.

[PF88] William H. Press and Brian B. Flannery. *Numerical Recipes*. Cambridge Univ. Press, 1988.

[TKS82] Trivedi, Kishor, and Shridharbhai. *Probability and Statistics*. Prentice-Hall, 1982.

[Wei99] Bettina Weiss. Simulation environment for clock synchronization. *Technical Report 183/1-88, Vienna University of Technology, Dept. of Automation*, February 1999.