Lut Based Uniform And Gaussian Random Generators

Rachana.M.K
Scholor of M.E VLSI Design
Maharaja Engineering College,Avinasi,Coimbatoor

Mrs.P.Hemalatha
Assistant Proffessor
MEC,Avinasi,Coimbatoor

Abstract

Randomness take place everywhere in Life. The many applications of randomness have led to the development of several different methods for generating random data. Functions in most software development tools output random numbers with uniform distribution. Simulations often need random numbers in normal distribution. Random number generators are very useful in developing Monte Carlo-Method simulations or a quantitative risk analysis technique, as debugging is facilitated by the ability to run the same sequence of random numbers again by starting from the same random seed. A novel method for generating Gaussian random numbers from uniform random number as a seed, simple LUTs and registers are presented here and make best use of all available LUT inputs in a given FPGA architecture using VHDL programming language and the simulation was done and tested on the Model Sim 6.3f. This method provides an easy, efficient and fastest method for simulating Monte Carlo-applications.

1. Introduction

Uniform*-All values have an equal chance of occurring, and the user simply defines the minimum and maximum.*

Normal *– Or "bell curve." The user simply defines the mean or expected value and a standard deviation to describe the variation about the mean. Values in the middle near the mean are most likely to occur. It is symmetric and describes many natural phenomena such as people's heights*

Many scientific and industrial problems have no tractable closed-form solution, and can only be solved through Monte- Carlo simulations. However, such simulations require huge amounts of computational power, and the power and size limits of conventional compute-farms have led to significant interest in the use of FPGAs in such applications. A **random number generator (RNG)** is a computational or physical device designed to generate a sequence of numbers or symbols that lack any pattern, i.e. appear random. Many of the random number generators have existed since ancient times, including dice,coin flipping, the shuffling of playing cards, the use of yarrow stalks (by divination) in the I Ching, and many other techniques. Because of the mechanical nature of these techniques, generating large numbers of sufficiently random numbers (important in statistics) required a lot of work and/or time. Here a Uniform random number is generated, using only the **most basic primitives** of FPGAs: Flip-Flops (FF), Lookup Tables (LUT), Shift Registers (SR) . The construction method is designed to ensure maximum clock rates, while using the minimum of resources, and providing statistical quality at the level of software applications.

The overall system view is as shown in fig:1.In this Uniform random number generated is given as a seed to Gaussian random number generator ,as uniformly distributed between 0 and 1 can be used to generate random numbers of any desired(Gaussian) distribution by passing them through the inverse cumulative distribution function (CDF) of the desired (Gaussian) distribution. Inverse CDFs are also called functions. And the output of MVGRNs is given for simulating Monte carlo applications. MGRNS are also generated only using LUTs,and registers.

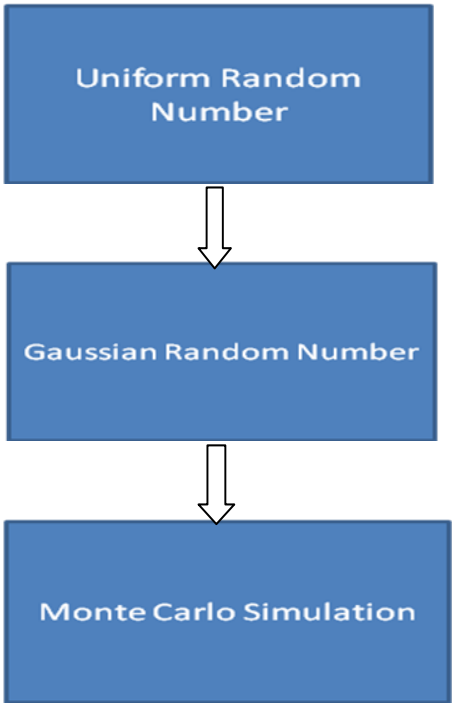


Fig:1 overall system view

Available Uniform PRNGs

Linear Congruential Generator

The classic generator is the *linear congruential generator* (LCG) (Knuth 1969), which uses a transition function of the form $x_{n+1} = (ax_n+c) \bmod m$. The maximum period of the generator is m (assuming the triple (a, c, m) has certain properties), but this means that in a 32-bit integer, the period can be at most 2^{32}, which is far too low. LCGs also have known statistical flaws, making them unsuitable for modern simulations.

Multiple Recursive Generator

A derivative of the LCG is the *multiple recursive generator* (MRG), which additively combines two or more generators. If n generators with periods $m_1, m_2, \ldots, m_n$ are combined, then the resulting period is LCM($m_1, m_2, \ldots, m_n$), thus the period can be at most $m_1x, m_2x, \ldots, m_nx$. These generators provide both good statistical quality and long periods, but the relatively prime moduli require complex algorithms using 32-bit multiplications and divisions, so they are not suitable for current GPUs (NVIDIA 2007, Section 6.1.1.1).

Lagged Fibonacci Generator

A generator that is commonly used in distributed Monte Carlo simulations is the *lagged Fibonacci generator* (Knuth 1969). This generator is similar to an LCG but introduces a delayed feedback, using the transition function $x_{n+1} = (x_n \otimes x_{n-k}) \bmod m$, where $\otimes$ is typically addition or multiplication. However, to achieve good quality, the constant k must be large. Consequentially k words of memory must be used to hold the state. Typically k must be

greater than 1000, and each thread will require its own state, so this must be stored in global memory. We thus reject the lagged Fibonacci method but note that it may be useful in some GPU-based applications, because of the simplicity and small number of registers required.

Mersenne Twister

One of the most widely respected methods for random number generation in software is the *Mersenne twister* (Matsumoto and Nishimura 1998), which has a period of $2^{19,937}$ and extremely good statistical quality. However, it presents problems similar to those of the lagged Fibonacci, because it has a large state that must be updated serially. Thus each thread must have an individual state in global RAM and make multiple accesses per generator. In combination with the relatively large amount of computation per generated number, this requirement makes the generator too slow, except in cases where the ultimate in quality is needed.

Combined Tausworthe Generator

Internally, the Mersenne twister utilizes a binary matrix to transform one vector of bits into a new vector of bits, using an extremely large sparse matrix and large vectors. However, there are a number of related generators that use much smaller vectors, of the order of two to four words, and a correspondingly denser matrix. An example of this kind of generator is the *combined Tausworthe generator*, which uses exclusive-or to combine the results of two or more independent binary matrix derived streams, providing a stream of longer period and much better quality. Each independent stream is generated using TausStep, shown in Listing 37-2, in six bitwise instructions. For example, the four-component LFSR113 generator from L'Ecuyer 1999 requires 6 x 4 + 3 = 27 instructions, producing a stream with a period of approximately 2^{113}.

PROPOSED ALGORITHM

Uniform Random Number Generator

This paper describes a new method to create Random Number Generators (RNG) using only the most basic primitives of FPGAs: Flip-Flops (FF), Lookup Tables (LUT), Shift Registers (SR) . The proposed architecture has the significance of Less area utilisation, less power cosumption,Low clock frequency : about 550 Mhz,High speed generation : 48Gb/s,Extremely long periods: ~2^{11213} -1,and is implemented with VHDL description is platform independent.LUT-SR generator family uses a short and precise algorithm for expanding the full RNG structure. Circuit diagram is shownin fig:2

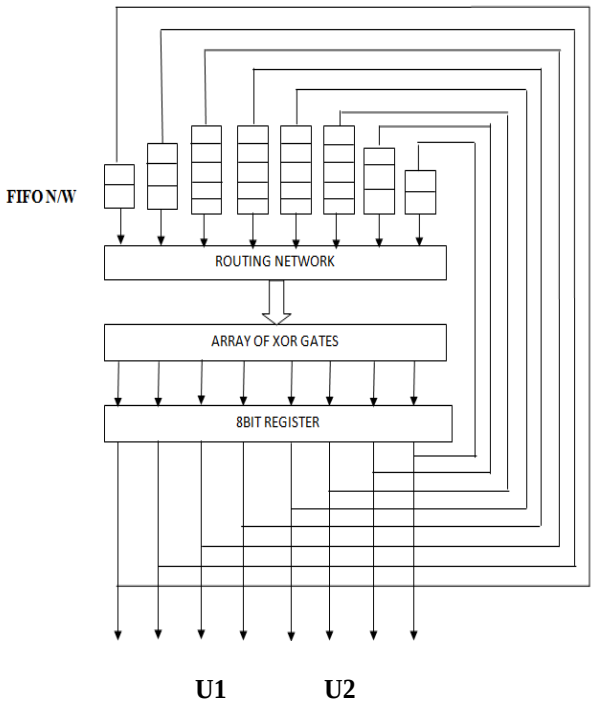


Figure No:2 Circuit Diagram of Uniform Random Number Generator

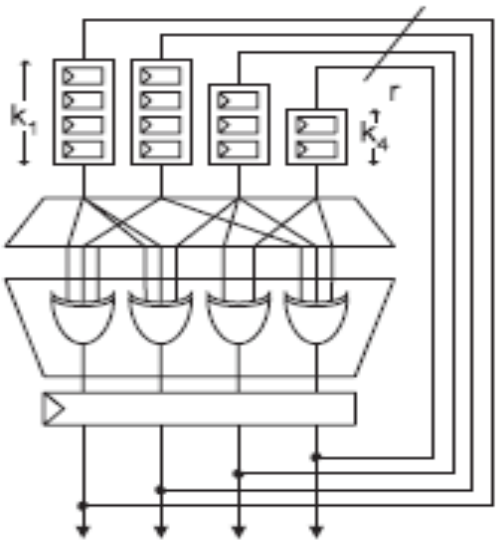


Fig NO:3 Uniform random number generator

Most Modern FPGAs allow LUTs to be configured in a number of different ways, such as basic ROMs, RAMs, and shift registers. . while the FIFO in a LUT-FIFO RNG is usually an expensive block RAM, LUT-based shift registers are very cheap—almost as cheap as the LUTs used to build the XOR gates. So it now becomes economical to use *r* shift registers,one per output bit, increasing the potential state to $n = r(1+k)$. If we assume $k = 32$ (as found in modern FPGAs), and a modest RNG output width of $r = 32$, the state size increases to $n = 1056$. This provides a potential period of $21056 - 1$ for a cost of 64 LUTs, as compared to a period of $264 - 1$ for a LUT-optimized generator with the same resource usage. Configuring LUTs as shift registers provides an attractive means of adding more storage bits to a binary linear generator: for example, adding one Xilinx SRL32 to a LUT-optimized *r* bit generator allows the state size to be increased to This represents a degenerate form of an LUT-FIFO generator $n = r+32$. with $k = 32$ and $w = 1$.Initially the loading step is done by giving a seed. For r bit generator the seed size is r.The seed could not be an all zero number ,as it cancels the Random number generation and makes the generator idle. As soon as the seed is given the bits are routed to inputs of array of XOR gates. Universal shift register performs shifting operation in addition to the parallel-in-parallel-out function.as FIFO Extension 1-bit shift registers are used. Bitwise shift registers improve the rate of mixing. For 8-bit RNG the length of shift register is given by the number of flip-flops.

The Basics of Normal Distribution

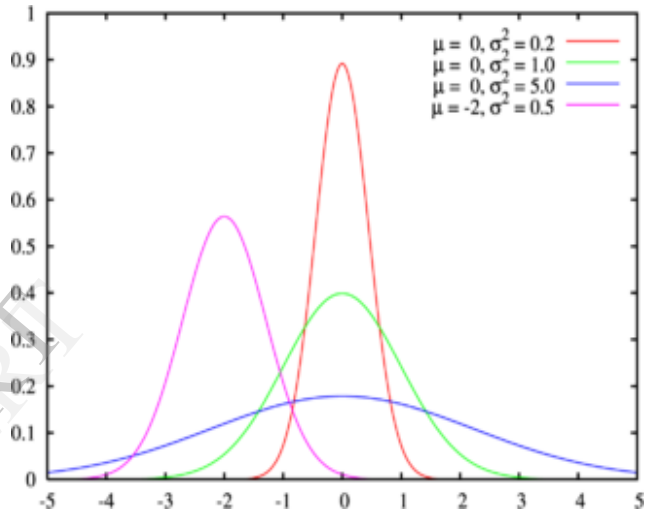


Fig No: 3 Normal Distribution curve

This graphic illustrates the two main characteristics of normal distribution. The first key figure is called the mean. You may also know it as the "average". It represents the most common value and is abbreviated with the Greek letter μ (mu). All of the curves in the example have a mean of 0 except the magenta one that has a mean of 2.

Why is the normal distribution useful?

• Many things actually are normally distributed, or very close to it. For example, height and intelligence are approximately normally distributed; measurement errors also often have a normal distribution

• The normal distribution is easy to work with mathematically. In many practical cases, the methods developed using normal theory work quite well even when the distribution is not normal.
• There is a very strong connection between the size of a sample N and the extent to which a sampling distribution approaches the normal form. Many sampling distributions based on large N can be approximated by the normal distribution even though the population distribution itself is definitely not normal.

Generating Standard Normal Distribution Programmatically

Most of the information out there about normal distribution involves calculating standard deviation and mean from a collection of data points. What we want to do is exactly the opposite.The simplest way of doing this is to invert the standard normal cumulative distribution function.

Random sampling from input distributions

Consider the distribution of an uncertain input variable x. The cumulative distribution function F(x) gives the probability P that the variable X will be less than or equal to x, i.e.

F(x) = P(X<=x)

F(x) obviously ranges from zero to one. Now, we can look at this equation in the reverse direction: what is the value of x for a given value of F(x)? This inverse function G(F(x)) is written as:

G(F(x)) = x

It is this concept of the inverse function G(F(x)) that is used in the generation of random samples from each distribution in a risk analysis model.

The figure below provides a graphical representation of the relationship between F(x) and G(F(x)):

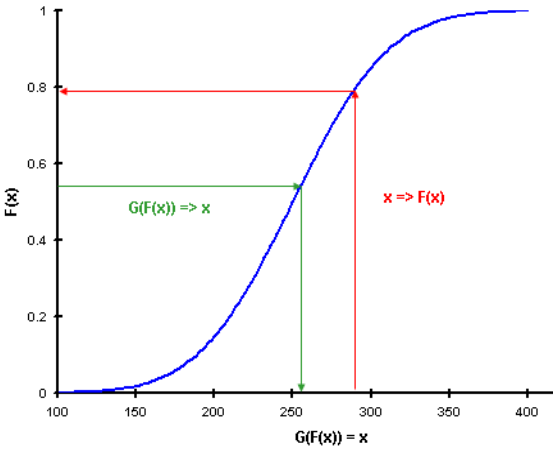


Fig No:4 Cumulative Distribution function for Normal Distribution

BACKGROUND: GENERATING MULTIVARIATE GAUSSIAN RANDOM NUMBERS

A multivariate Gaussian distribution is characterized by its mean vector m and correlation matrix **A**. Several methods to sample from multivariate Gaussian distribution exist in the literature. As far as the hardware implementation of the MVGRNG on an FPGA is concerned, some of the most widely used techniques are Cholesky factorization and Eigen value decomposition. Using Cholesky factorization method, **A** is decomposed using Cholesky decomposition into a product of a lower triangular matrix and its transpose, AA^T. The required samples are generated through a linear combination of univariate Gaussian samples that follow a standard Gaussian distribution $N(0; 1)$. This method reduces the number of computations by half due to the lower triangular property of the matrix in comparison to full matrix-vector multiplication. A disadvantage is that the

Cholesky decomposition only works with positive definite covariance matrices∑ _many matrices constructed from estimates may be singular or very close to singular. An alternative method is to decompose **A** by Eigen value decomposition. As a result of the decomposition, **A** can be expressed as a linear combination of three separable matrices U**S**U^T where U is an orthogonal matrix (UU^T = I) . The disadvantage of the SVD-based construction is that in general all the elements of the matrix are nonzero, resulting in an *n*2 cost in both the number of stored elements, and in the number of multiply-accumulates per transformed vector. However, the SVD algorithm is able to handle a wider range of covariance matrices, such as ill-conditioned matrices that are very close to singular and reduced rank matrices, where the output vector depends on fewer than *n* random factors. Such difficult covariance matrices frequently occur in practice. Generation of the univariate Gaussian distribution with a specific mean μ and variance, σ^2 generating a standard Gaussian variate *r with mean zero and* variance one, then applying a linear transformation,

$$x = \sigma r + \mu$$

For a multivariate Gaussian distribution the mean is a length *n vector* ***m*** **,**variance becomes an *n* × *n covariance matrix . The covariance matrix is a* symmetric matrix.

x = Ar + m.

The matrix A is conceptually similar to the SD .With the existing method that is to perform Cholesky decomposition of the correlation matrix, producing a lower-triangular matrix or an alternative method is to use the Singular Value Decomposition (SVD) algorithm. This decomposes the matrix into an orthogonal matrix U and a diagonal matrix S *Such that*

$\sum = \boldsymbol{USU^T}$. The disadvantage of the SVD-based construction is that in general all the elements of the matrix are nonzero.

Example

For bivariate Gaussian random output =x_1,x_2

Let mean m= $\begin{pmatrix} g \\ h \end{pmatrix}$

Then X= $\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e \\ f \end{pmatrix} + \begin{pmatrix} g \\ h \end{pmatrix}$

where $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is decomposed values by any above method.

and $\begin{pmatrix} e \\ f \end{pmatrix}$ is a vector of n independent identically distributed(IID)standard Gaussian numbers.

Then x_1 = ae+bf+g in 1st cycle

x_2 = ce+df+h in 2nd cycle

Thus for the generation of n Gaussian samples, it requires n cycles. Hence it is very long procedure to be followed.

GENERATION USING LUTS AND ADDERS

Does not required to generate vector r .Multiplication not required. Here uniform samples will be converted to correlated Gaussian samples using *table-lookups.* Each table contains a pre-calculated discretized

Gaussian distribution with the correct SD, so the only operations required are table-lookups and additions. The following text frequently refers to tables, which in this context means an array of read-only elements (a ROM) which will be implemented in the FPGA using LUTs. Unless otherwise specified, each table contains k elements, and is indexed using the syntax $L[i]$ to access table elements $L[1], \ldots, L[k]$. Where arrays of tables are used, sub-scripts identify a table within the array, which can then be indexed as for a standalone table, e.g., $L2,3$. Tables can also be interchangeably treated as discrete random number generators, where the discrete PDF of each table is given by assigning each element of the table an equal probability of $1/k$. The central idea of this method is to construct an $n \times n$ array of tables **G**, such that the discrete distribution of each table $\mathbf{G}i, j$ approximates a Gaussian distribution with SD $\mathbf{A}i, j$

$$\mathbf{G}i, j \sim N(0, \mathbf{A}i, j).$$

Now instead of starting from a Gaussian vector r, the input is an IID uniform vector u. Generation of each output element uses each element of u as a random index into the table, then sums the elements selected from each table . In practice k will be selected to be a power of 2, so each element of u is actually a uniform integer constructed from the concatenation of log2(k) random bits. The simplest method of generating a table-based approximation to the Gaussian distribution is direct cumulative distribution function (CDF) inversion. To generate a table L with SD σ , table elements are chosen according to

$$L[i] = \sigma \Phi^{-1}(i/(k+1)), i \in 1, \ldots, k$$

Where Φ^{-1} is the Gaussian inverse CDF. The central idea in this paper, of replacing Gaussian samples and multipliers with uniform samples and tables, allows for many types of possible implementations.

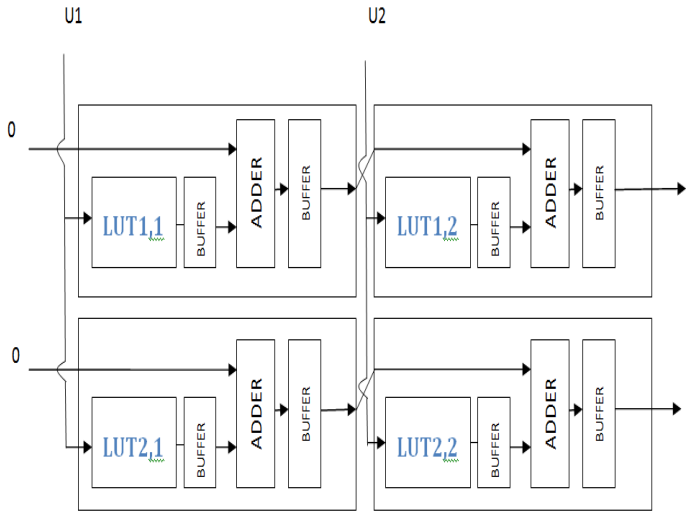


Figure No:5 Circuit Diagram of Gaussian Random Number Generator

At the top is a random bit source,which generates a new vector **u** for every cycle**.** The elements of u are broadcast vertically down through the cells, and used to select one element from each table. The selected elements are then accumulated horizontally from left to right by implementing the function $si, j = si, j-1 + Li, j [u j],$

Table elements are chosen according to

$L[i] = \sigma \Phi^{-1} (i/(k+1))$ i=1,……,k

Pdf of gaussian distribution is

- $\Phi(x)=1/\sqrt{2}\prod e^{-(x-\mu)2/2\sigma 2}$
- CDF is given by
- $\Phi(X)=1/2[1+erf(x/\sqrt{2})]$

Example

If k=8

Then

$L_1=\Phi^{-1}(1/9)$

ie

$0.111= 1/2[1+erf(x/\sqrt{2})]$

$-0.7778=erf(x/\sqrt{2})$

x=-1.72632

Then $L_1=\sigma_{11}*-1.72632$

Here we are going to implement a bi-variate Gaussian random number generator. So we

have to generate two uniform random numbers simultaneously. For this we are using a modified random number generator with variable length FIFO and Shift Registers.

Adder unit

Here we used Carry Select Adders in between two consecutive stages of LUTs. Here we used the carry save adders advantage of fast operation over other kind of adders.

CONCLUSION

This paper presents a low cost architecture for the implementation of Uniform as well as Gaussian Random Number Generator. Both make use of simple LUTs, Shift registers, and adders. This promises a great advantage over other existing system in terms of _high speed, and performance efficiency. It provides Long period independent Random numbers which can be directly used as input for Monte carlo applications.

REFERENCES

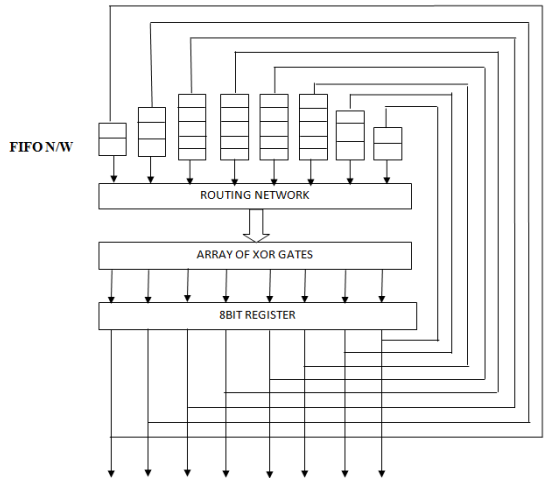
[1] D. B. Thomas and W. Luk, "Multivariate Gaussian random number generation targeting reconfigurable hardware," ACM Transactions on Reconfigurable Technology and Systems, vol. 1, no. 12, 2008.

[2]SHELDON ROSS. *Introduction to Probability Models*. Harcourt India Pvt. Ltd., 2001,
7th edition

[3]D. B. Thomas and W. Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation," in *Proc. ACM Symp. FPGAs Custom Comput. Mach.*, 2008, pp. 229–238.

[4] N. Woods and T. VanCourt, "FPGA acceleration of Quasi-Monte Carlo in finance," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2008, pp. 335–340.

[5]C. Saiprasert, C.-S. Bouganis, and G. Constantinides, "Mapping multiple multivariate gaussian random number generators on an FPGA," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2010, pp. 89–94.



7. Main text

Type your main text in 10-point Times, single-spaced. Do **not** use double-spacing. All paragraphs should be indented 1/4 inch (approximately 0.5 cm). Be sure your text is fully justified—that is, flush left and flush right. Please do not place any additional blank lines between paragraphs.

Figure and table captions should be 10-point boldface Helvetica (or a similar sans-serif font). Callouts should be 9-point non-boldface Helvetica. Initially capitalize only the first word of each figure caption and table title. Figures and tables must be numbered separately. For example: "Figure 1. Database contexts", "Table 1. Input data". Figure captions are to be centered *below* the figures. Table titles are to be centered *above* the tables.

8. First-order headings

For example, "1. Introduction", should be Times 12-point boldface, initially capitalized, flush left, with one blank line before, and one blank line after. Use a period (".") after the heading number, not a colon.

8.1. Second-order headings

As in this heading, they should be Times 11-point boldface, initially capitalized, flush left, with one blank line before, and one after.

8.1.1. Third-order headings. Third-order headings, as in this paragraph, are discouraged. However, if you must use them, use 10-point Times, boldface, initially capitalized, flush left, preceded by one blank line, followed by a period and your text on the same line.

9. Footnotes

Use footnotes sparingly (or not at all) and place them at the bottom of the column on the page on which they are referenced. Use Times 8-point type, single-spaced. To help your readers, avoid using footnotes altogether and include necessary peripheral observations in the text (within parentheses, if you prefer, as in this sentence).

10. References

List and number all bibliographical references in 9-point Times, single-spaced, at the end of your paper. When referenced in the text, enclose the citation number in square brackets, for example [1]. Where appropriate, include the name(s) of editors of referenced books.

[1] A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.
[2] Jones, C.D., A.B. Smith, and E.F. Roberts, *Book Title*, Publisher, Location, Date.

11. Copyright forms and reprint orders

You must include your fully-completed, signed IJERT copyright release form when you submit your paper. We **must** have this form before your paper can be published in the proceedings. The copyright form is available as a Word file in author download section, <IJERT-Copyright-Agreement-Form.doc>.