MODELLING AND SIMULATION OF A FADING CHANNEL

by

Ali Keyvani

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF APPLIED SCIENCE

in the

School of Engineering Science

SIMON FRASER UNIVERSITY

December, 2003

Table of Contents

Table of Tables

Table of Figures

Abstract

The term *fading*, in the context of mobile communications, refers to the interference caused by the reception of numerous scattered copies of a given signal at an antenna. Fading can produce significant random variations of signal power –in the scale of 10s of dBs over fractions of a wavelength- and hence, can be extremely destructive to the signal. Therefore, in order to achieve reliable mobile communications, provisions must be considered to counter the effects of fading. Indeed, a first step towards this end is to understand these effects through producing models of and simulating this phenomenon.

This thesis discusses the development of an Object-Oriented MATLAB component that simulates the effects of fading, based on a somewhat simplified mathematical model. Clearly the simplified model might not present a very realistic view of the highly complex and random nature of fading; nevertheless, this simulation serves as an extremely useful research tool for comparing and measuring the effectiveness of different communication techniques.

Font Conventions

A number of font conventions have been used to assist in the understanding of the text and to avoid confusion. These are listed in the following table.

Sample Font Used	Comments
Normal text	Normal Text!
Special Word	A term that has a specific meaning defined elsewhere in the text. Exception when italics are used to *emphasize* certain words.
Class Name	Name of a class (e.g. ComplexGainGen)
memberFunction or **memberVariable**	A member function or a member variable of a given class (e.g. **process**)
A = generate(c, 0);	MATLAB code

1 Introduction

Designers and engineers of mobile communications systems are faced with three main challenges, which are introduced by the communication channel: Path Loss, Shadowing and Fading.

Path Loss refers to the decrease in signal power, which is mainly brought about by the physical distance between the communications devices. Shadowing takes on a more local view and refers to the loss of power attributed to large obstacles such as hills and tall buildings. Finally, Fading, the main topic of this thesis, takes on a yet more microscopic view and is concerned with the interference caused by the reception of numerous scattered copies of the signal at the antenna.

The interference caused by fading produces significant random variations of signal power –in the scale of 10s of dB over fractions of the wavelength- about a mean power predicted by the Path Loss and Shadowing models. As a result, fading can be extremely destructive to the signal and hence, in order to achieve reliable communication, provisions must be considered to counter the effects of fading. Indeed, a first step towards this end is to understand these effects through producing models of and simulating fading.

This thesis project is concerned with producing a reusable simulation component in MATLAB, based on a somewhat simplified model of fading. Clearly the simplified model might not present a very realistic view of the highly complex and random nature of fading; nevertheless, this simulation serves as an extremely useful research tool for comparing and measuring the effectiveness of different communication techniques. The simulation model will be created in the form of an easily customizable component that can be effortlessly “plugged-in” to other MATLAB simulations.

2 A Brief Mathematical Model of Fading

The transmission of a bandpass and narrowband signal to a mobile is modeled as a multipath channel structure shown in Figure 2-1.

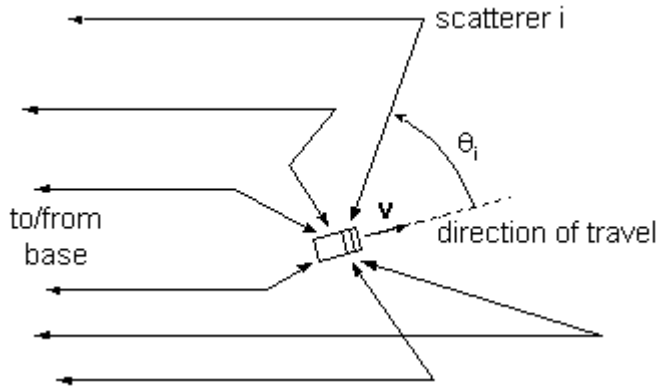


Figure 2-1: Multipath channel structure. [1]

The above figure shows the reception of multiple scattered copies of the signal, each with gain a_i and angle θ_i, by a mobile moving with speed v. A simple mathematical derivation yields the following result for *y(t)*, the complex envelope of the received signal.

$$y(t) = \sum_i a_i e^{-j2\pi f_D \cos\theta_i t} s(t-\tau_i) \qquad \textbf{(2.1)}$$

Equation 2.1 shows that scatterer i corresponds to a signal copy that is shifted in time by τ_i and in frequency by $f_D \cos(\theta_i)$, where $f_D = \frac{v}{\lambda}$ is the *Doppler frequency.* We also define the d*elay spread* (τ_d) as the largest delay (τ_i).

We can also show the above equation as an integral over a density,

$$y(t) = \int_{-\infty}^{\infty} \int_{-f_D}^{f_D} \gamma(\nu,\tau) e^{-j2\pi\nu t} s(t-\tau) d\nu d\tau \qquad \textbf{(2.2)}$$

where $\gamma(\nu,\tau)d\nu d\tau$ is the overall gain caused by all scatterers with a 'delay τ in $d\tau$ and Doppler ν in $d\nu$' [1]. Consequently, the output corresponding to an input signal consisting only of the one frequency f is

$$y(t) = e^{j2\pi ft} \int_{-\infty}^{\infty} \int_{-f_D}^{f_D} \gamma(\nu,\tau) e^{j2\pi\nu t} e^{j2\pi f\tau} d\nu d\tau = e^{j2\pi ft} G(t,f) \qquad \textbf{(2.3)}$$

Here, *G(t,f)* is the gain experienced by frequency component *f* at time *t* –i.e., a time-dependent frequency response. Therefore, we can also write our input/output relationship in frequency domain as

$$y(t) = \int_{-\infty}^{\infty} G(t,f)S(f)e^{j2\pi ft}df \quad \textbf{(2.4)}$$

while the time-domain equivalent of the above equation is given by

$$y(t) = \int_{0}^{\tau_d} g(t,\tau)s(t-\tau)d\tau \quad \textbf{(2.5)}$$

Using definition of *G(t,f)* given in Equation 2.3 and assuming a WSSUS channel, we calculate the autocorrelation (for components separated in time by Δt and in frequency by Δf),

$$R_G(\Delta t, \Delta f) = \int_{-\infty}^{\infty}\int_{-f_D}^{f_D} S_\gamma(\nu,\tau)e^{j2\pi\nu\Delta t}e^{j2\pi\Delta f\tau}d\nu d\tau \quad \textbf{(2.6)}$$

Here, $S_\gamma(\nu,\tau)$ is the 'delay-Doppler power density' [1], referring to the gain experienced at Doppler *v* and delay τ .

In order to simplify our model we consider a special case with the following two properties:

1. delay-Doppler power density is *separable,* i.e. we have 'two one-dimensional functions to describe the scattering, rather than one two-dimensional function.'[1] We then have

$$S_\gamma(\nu,\tau) = \frac{S_g(\nu)}{\sigma_g}.\frac{P_g(\tau)}{\sigma_g} \quad \textbf{(2.7)}$$

   where $S_g(\nu)$ is referred to as the *Doppler spectrum,* i.e. the power at Doppler ν *,* and $P_g(\tau)$ denotes the *power delay profile*, i.e. the power at delay τ .

2. Scattering is *isotropic,* i.e. the power received by the mobile from all angles, $P(\theta)$, is uniform. We then have,

$$S_g(\nu) = \frac{\sigma_g^2}{\pi f_D} \cdot \frac{1}{\sqrt{1-(\frac{\nu}{f_D})^2}} \quad \textbf{(2.8)}$$

and, consequently

$$R_g(\tau) = \sigma_g^2 J_0(2\pi f_D \tau) \quad \textbf{(2.9)}$$

3 Component High-Level Description

3.1 Usage

Viewed as a 'black-box', the general usage scheme of the Fading Simulation Component is shown in the following diagram, where the input/output relationship is governed by Equation 2.1. The Fading Simulation receives the input signal s(t), adds simulated fading effects based on the Fading Parameters and outputs the signal y(t). The input signal s(t) is originated in a Transmitter, while the output signal y(t) eventually reaches a Receiver.

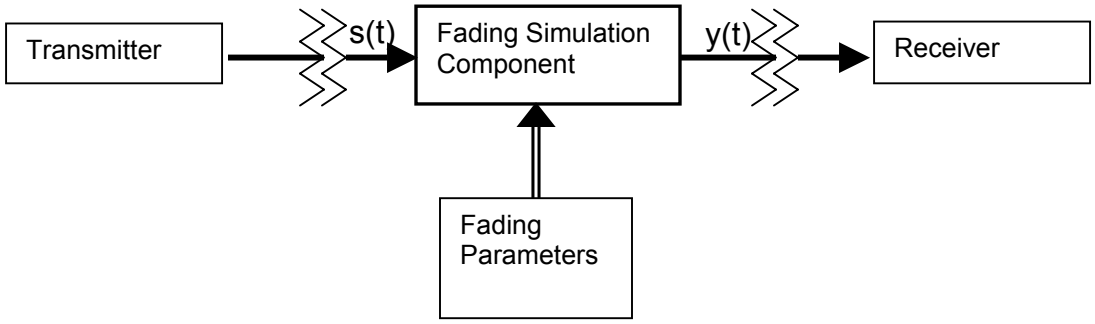


Figure 3.1: Black-box view of the Simulation Component

The input and output of the component are in the form of arrays of complex numbers of an arbitrary length, containing the input and output samples respectively. In addition, the component could be used in real-time, in conjunction with another MATLAB program.

3.2 Structure

Internally, the Fading Simulation Component consists of a Tapped Delay Line (TDL) and a number of Complex Gain Generators.

3.2.1 The Tapped Delay Line

The TDL structure is shown in the following diagram. Corresponding to Equation 2.1, the TDL generates multiple time-shifted copies of the input signal, each of which is weighted by a complex gain (which is, in turn, generated by the complex gain generator.)

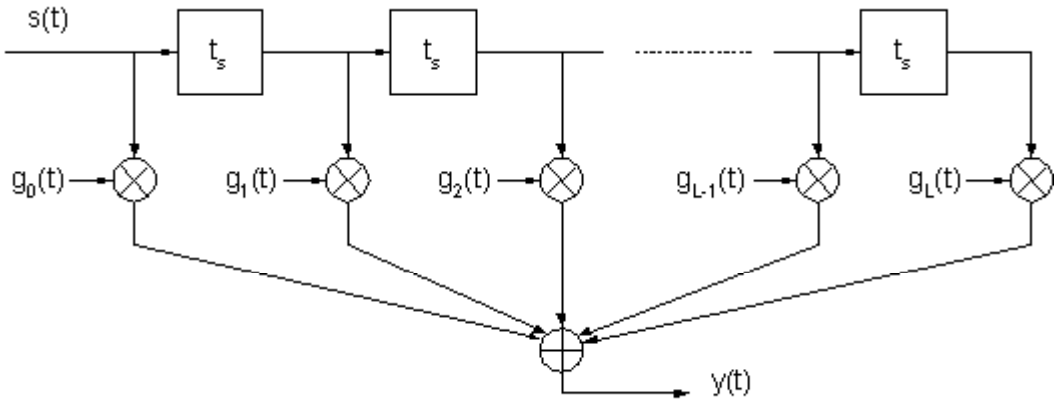


Figure 3.2: Tapped Delay Line [1]

To mimic Equation 2.1, the TDL is, in effect, using discrete 'delay bins' of width *ts=1/fs*, where the complex gain corresponding to each bin reflects the weighted sum of the complex gain of all delayed signal copies that are included in that bin. Also, note that the Nyquist criterion requires $f_s>2W$, where W is the bandwidth of the signal plus the Doppler frequency (f_D).

3.2.2 The Complex Gain Generator

The following three properties must hold for the complex gain generator [1]:

1. Real and imaginary parts of the generated complex gain must be independent and Gaussian, having the same autocorrelation.
2. Different complex gain generators (corresponding to different taps in the TDL model) must be independent.
3. The autocorrelation function must be the one given in Equation 2.8.

Two of the most widely used methods for complex gain generation were implemented. These are Jakes Method and the method of Filtered White Noise.

a) Jakes Method

Here, a slightly modified version of Jake's Method [1] is used. Basically, Jake's Method works by simulating the physical model of isotropic scattering. It assumes that there are N equispaced scatterers around the mobile, all with the same gain. However, the phase angle associated with each scatterer is chosen at random.

The following equation is used by Jake's Method to generate complex gain samples:

$$y(t) = \frac{1}{\sqrt{N_s}} \sum_{i=0}^{N_s-1} e^{j[\phi_i + 2\pi f_D t \cos(\theta_i)]} \quad \textbf{(3.1)}$$

where, ϕ_i is the random phase and θ_i is the arrival angle.

Through careful mathematical derivation, we can prove that Jake's Method does in fact hold the three properties given above (In particular, Property 2 is only satisfied if no two complex gain generators share the same arrival angles, θ_i) Also, using Jake's Method is advantageous in that it can be run backward and forward in time with desired spacing, it has low computational load and is fairly simple [1].

b) Filtered White Noise

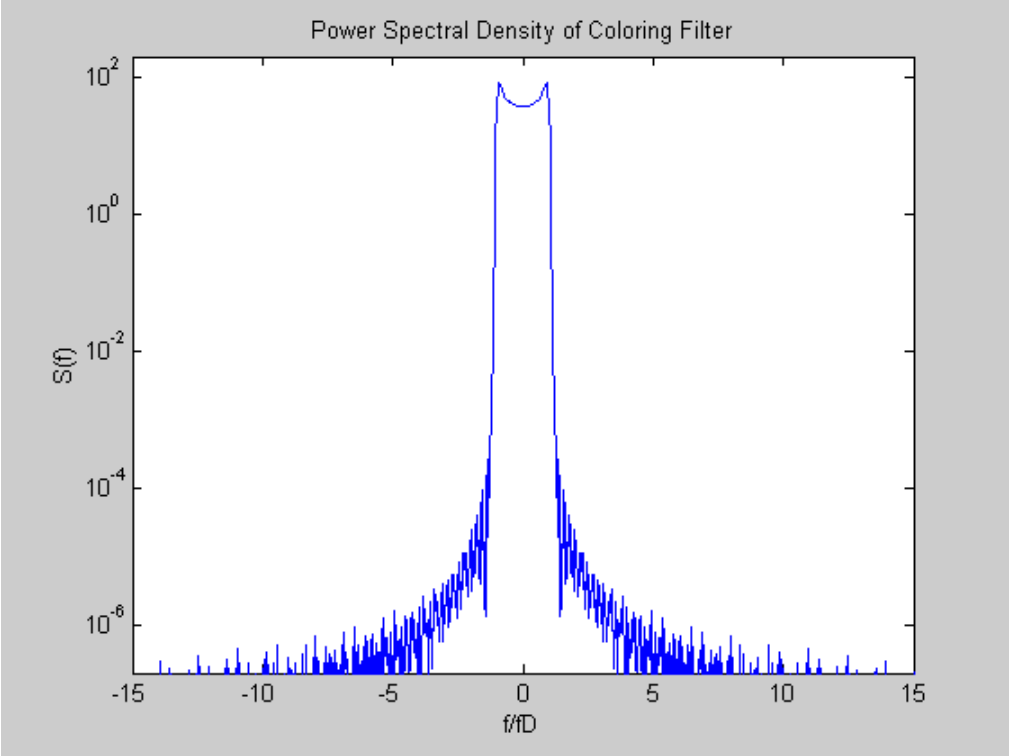


Figure 3.2-1: Power Spectral Density of the Coloring Filter

The method of Filtered White Noise achieves the desired power spectrum by generating white Gaussian noise samples and passing them through a spectrum shaping filter (the frequency response is plotted in Figure 3.2-1). Two independent white noise generators are used for the real and imaginary parts respectively. The filters used have the following frequency response corresponding to Equation 2.7.

$$H(\nu) = \sqrt{S_g(\nu)} \qquad \textbf{(3.2)}$$

By virtue of its construction, the Filtered White Noise technique also holds the three properties stated above. Also, using this method is advantages in that it generates truly Gaussian and stationary samples. On the other hand, it requires a large FIR or IIR filter (and therefore a high computational load) and cannot be easily run backward in time.

3.3 Input Parameters

To be able to use the component, the user must specify a number of parameters to effectively customize the component to his/her needs. The basic parameters are the following:

- Normalized sample spacing, $f_D.t_s$
- Number of Taps in the TDL
- The power delay profile, specified as a vector containing the variance of each Tap of the TDL (i.e. σ_{gi}^2)
- The LOS components for each tap (if any)

In addition, when using Jake's Method, N_s, the number of scatterers must be specified. On the other hand, when using the method of Filtered White Noise, the truncation length of the filter used and the Kaiser Window parameter, β, must be supplied as parameters.

4 Component Implementation

As discussed before, the programming language chosen for this project was MATLAB. This is indeed a sensible choice because MATLAB is often the preferred language for creating communications simulations. Therefore, being in MATLAB, the component is available to a wide range of applications.

Another crucial decision made was to use the Object Oriented extensions of MATLAB and create a modular and object-based program. As a result, this simulation consists of a number of subcomponents (or classes) that, on the one hand, form the Fading Simulation Component, and on the other hand, are independent enough to be used individually.

This section describes the Object Oriented structure of the simulation, including relevant details regarding each subcomponent (i.e. class) of the internal structure. While this section focuses on the *general function* of each subcomponent, as well as how this function is *performed*, Appendix B contains brief instructions on how each subcomponent should be *used.*

Please note that a basic understanding of general Object Oriented concepts is required for this section. It is highly recommend that you read Appendix A for an introduction to Object Orientation, if you are not familiar with such concepts.

4.1 Object-Oriented Static Architecture

The UML static diagram contained in the next page depicts the internal class structure of the component.

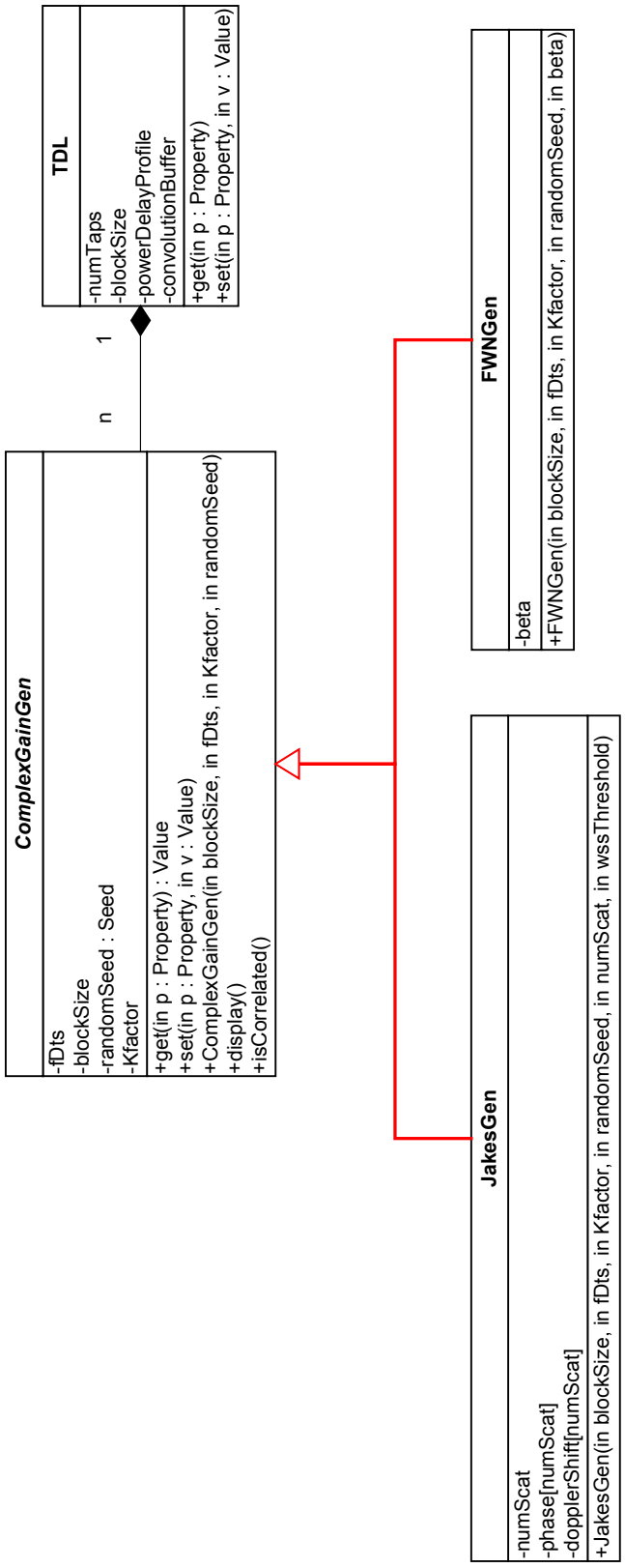


Figure 4.1-1: Object-Oriented Static Diagram

As shown in Figure 4.1-1, the structure consists of four classes: ComplexGainGen, JakesGen, FWNGen and TDL. ComplexGainGen is an *abstract base class* from which JakesGen and FWNGen are *derived.* On the other hand, the class TDL forms an *association* with ComplexGainGen.

4.2 Class description

This section describes the functionality of each of the four classes listed above. Where appropriate, references are made to Section 3, to better explain the purpose of each class. Also, references are made to Section 2 to indicate the mathematical basis on which each class is built.

4.2.1 ComplexGainGen: Complex Gain Generator Base Class

ComplexGainGen is an abstract class defining the common *interface* shared by all types of Complex Gain Generator classes. This common interface is simply based on the following facts about any Complex Gain Generator, regardless of class:

1. It needs to know the sampling frequency, f_s. This is normalized by the Doppler frequency f_D forming *fD/fs* or *fDts*. Thus, **fDts** is a *member variable*.

2. It (obviously!) generates complex gain samples -hence the **generate** *member function.* This function returns an array of complex gain samples.

   It is important to note that **generate** is in fact an *abstract function.* In other words, the interface of ComplexGainGen merely suggests that a Complex Gain Generator should be able to generate an array of complex gain samples. This only specifies *what* needs to be done; *how* this is done is not specified and is left for the derived classes to implement. Obviously, different types of complex gain generators (i.e. derived classes) have different ways of implementing this function.

3. It needs to know the size of the array of complex gain samples returned by **generate** –hence the **blockSize** member variable.

4. It needs to know the seed to be used when generating random numbers. The complex gain generation is a random process and therefore, requires the use of a random number generator (which generates “random” numbers based on a given Seed.) In order to reproduce the same samples generated by a Complex Gain Generator ‘A’, all we need to do is to set the Seed value of Complex Gain

Generator 'B' to be the same as that of 'A'. Therefore, **randomSeed** is a member variable.

5. It needs to know the Ricean K-factor[1] used to add an LOS component to the generated gains –hence the **Kfactor** member variable.

6. It should provide means of displaying a brief textual representation of its internal state -hence the **display** member function.

7. It should provide means of changing and acquiring the values of its member variables -hence the **get** and **set** member functions.

8. It should be able to determine if it is correlated to another Complex Gain Generator of the same type –hence the **isCorrelated** member function[2].

4.2.2 JakesGen: Complex Gain Generation using the Jakes Method

JakesGen is derived from ComplexGainGen and implements the Jakes Method of Complex Gain Generation (this is described in Section 3.2.2.) As stated before, JakesGen is a self-contained module, which can be used independently to generate complex gain samples.

The main function of the JakeGen class is to provide an implementation for the **generate** member function derived form ComplexGainGen, by employing Equation 3.1. To this end, JakesGen requires the number of scatterers (N_s) along with the initial phase (ϕ_i) and the arrival angle (θ_i) of each scaterrer. These values are initialized in the *constructor*:

The constructor, stores the initial phase angles as an array of uniformly distributed random angles in the **phase[]**[3] member variable. Since the scatterers are assumed to be equispaced around the mobile (see Section 3.2.2), all we need is the arrival angle of only *one* of the scatterers (which is selected at random); the rest of the arrival angles can be generated accordingly. The **dopplerShift[]** member variable is used to store the Doppler Shifts corresponding to each of the arrival angles.

---

[1] The K-factor is defined as the ratio of power in the LOS component to the power in the scattered component of the complex gain [1].

[2] A general-purpose function called 'createUncorGenerators' has been provided that uses the **isCorrelated** member function to create an array of uncorrelated Complex Gain Generators of desired size.

[3] The square brackets in phase[] are used to signify the fact that this member is an array of values.

It is important to note that the constructor uses the following two criteria to ensure that the generated process is stationary [1]:

1) N_s (**numScat**, the number of scatterers) should be odd
2) No arrival angle should be equal to $\pm\frac{\pi}{2}$

In order to make sure that the second criteria holds the constructor goes through the following steps:

1. θ_0 = random angle
2. $\theta_i = \theta_0 + \frac{2\pi}{numScat}.i$ for $i = 1..numScat - 1$
3. Is any of the θ_i "close" to $\pm\frac{\pi}{2}$, if yes go to 1, otherwise go to 4
4. Done

Please note that, in step 3, "closeness" is determined using a user specified threshold value (**wssThreshold**.)

JakesGen also provides an implementation for the **isCorrelated** member function. This implementation is based on the fact that, as mention in Section 3.2.2, when using the Jakes Method, two Complex Gain Generators are uncorrelated if they do not share the same arrival angles (θ_i). Since the arrival angles are equispaced and are generated using a random initial angle, for two Complex Gain Generators to be uncorrelated it suffices to check that their initial arrival angles are not too close (again "closeness" is determined using a threshold parameter passed to **isCorrelated**).

4.2.3 FWNGen: Complex Gain Generation using Filtered White Noise

FWNGen is derived from ComplexGainGen and implements the Method of Filtered White Noise (this is described in section 3.2.2.) As stated before, FWNGen is a self-contained module, which can be used independently to generate complex gain samples.

The main function of the FWNGen class is to provide an implementation for the **generate** member function derived from ComplexGainGen. As discussed in Section 3.2.2, this implementation directly filters White Gaussian Noise samples to achieve the desired power spectral density (Equation 2.7.) This is done through the Overlap-Add Filter Method [2] as described in the following flowchart:

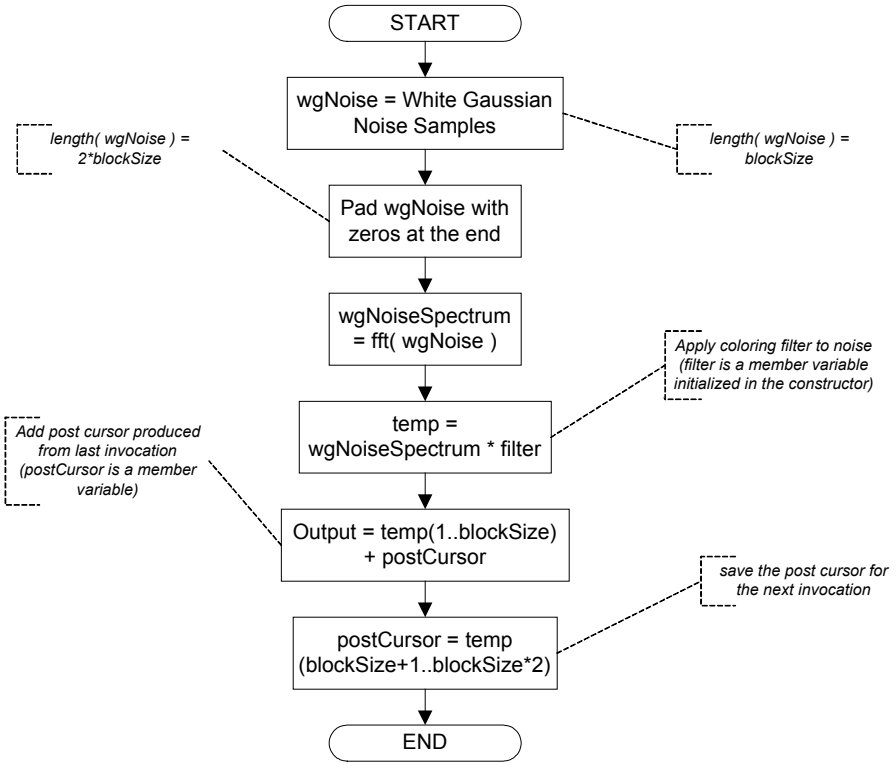


Figure 4.2-1: Overlap-Add Filter Method Flowchart

The coloring filter used in **generate** is created by the constructor and stored in the filter member variable. It is important to note that the length of this filter (i.e., the truncation length) is equal to the **blockSize** member variable (i.e., the length of the complex gain blocks returned by **generate**.) Figure 4.2-3 shows a flowchart that describes how the filter is created by the constructor.

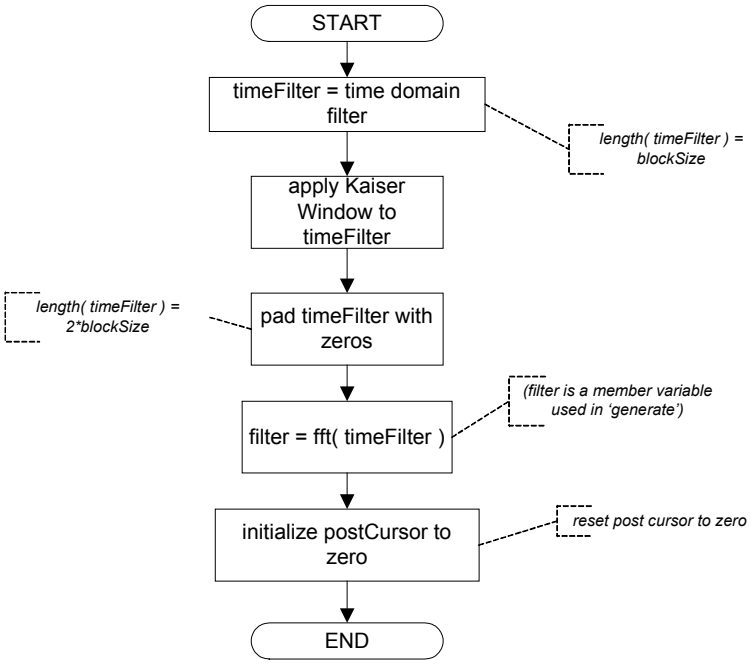


Figure 4.2-2: Creating the coloring filter for FWNGen

4.2.4 TDL: The Tapped Delay Line

As discussed in Section 3.2.1, the main function of the TDL class is to apply the effects of fading to an input signal, according to the time varying linear transfer function of Equation 2.3; this is done in the **process** member function. The function receives as an argument, an array of complex samples representing the signal, simulates the effects of fading on this signal and returns the results.

In order to perform this function, the TDL class maintains the following information as member variables:

- **Power Delay Profile:** This information is supplied by the user as an argument to the **constructor**. The Power Delay Profile is an array of variances, one for each tap of the TDL, defining the power of the complex gain in each "delay bin".

- **Convolution Buffer:** This array is used to hold the results of the convolution resulting from a call to process. Obviously, the results of the convolution are retained in this buffer between successive calls to the process member function, ensuring continuity of the process.

- **Array of Complex Gain Generators:** This array holds multiple instances of Complex Gain Generators (either JakesGen or FWNGen), one for each tap of the TDL. The association relationship in Figure 4.1-1 represents this array.

The following flowchart describes how the **process** function works.

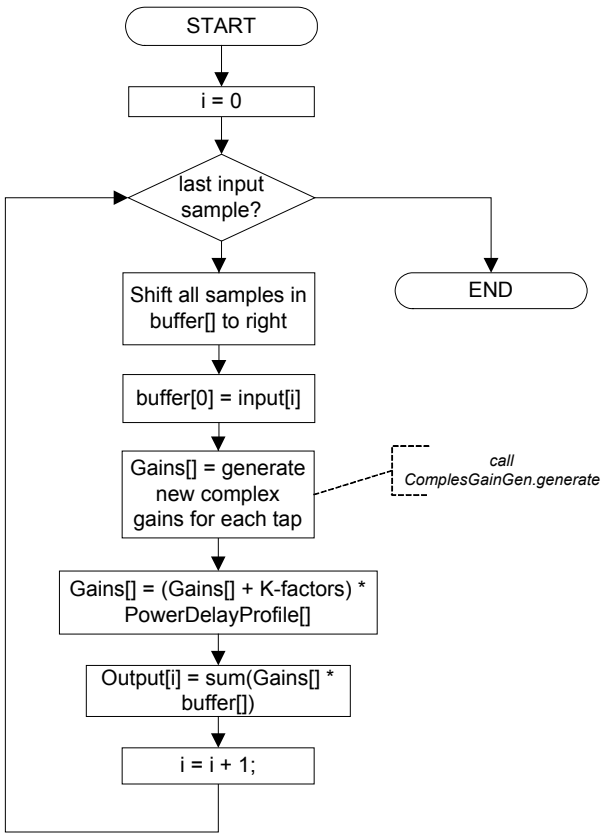


Figure 4.2-3: Flowchart of the process member function of the TDL class

4.3 Design Advantages

The Object-Oriented design outlined in Section 4.1 brings about numerous advantages. This section briefly discusses some of the important benefits that are gained from this design.

4.3.1 Modularity

One of the most important aspects of Object Oriented systems is Modularity. An effort has been made to construct loosely coupled and independent modules, which can be used individually as well as together as a system. For instance, although the JakesGen class is used in conjunction with the TDL for the purpose of Fading Simulation, it can very well be used on its own for the purpose of Complex Gain Generation.

4.3.2 Extendibility

A very important aspect of this system is the fact that it can be effortlessly extended. This is brought about by the use of *polymorphism* in the construction of the Complex Gain Generator classes.

As discussed in section 4.2.1, ComplexGainGen has an interface for which no implementation is provided. By relying on this common interface, the TDL class decouples itself from the specific type of generator used. As long as a Complex Gain Generator conforms to this interface, the TDL will be able to use it. This notion is depicted in figure 4.3-1.

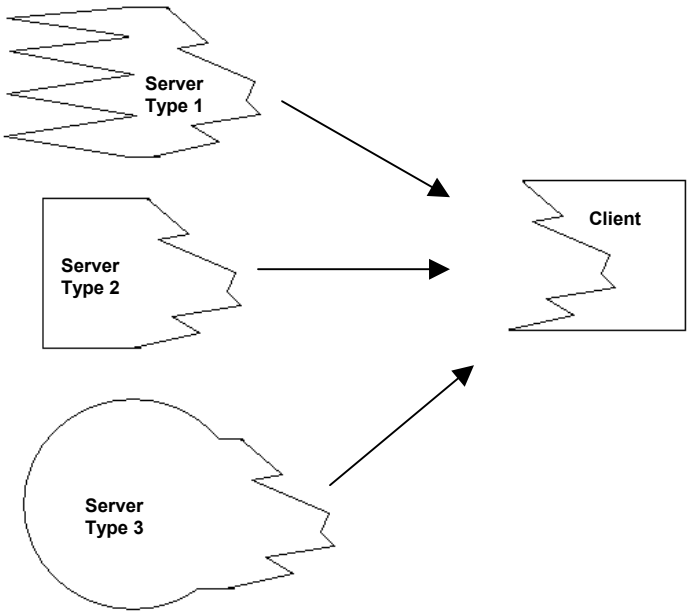


Figure 4.3-1: Polymorphism

The important result of this property is that in the future, other techniques of generating complex gains can be implemented as new ComplexGainGen derived

classes and be used in conjunction with TDL –all that, with absolutely no change being required in the implementation of TDL.

5 Testing

After (and during) implementation, extensive testing was done on each subcomponent of the simulation to ensure that the operation of each was as expected. To this end, various properties were measured and compared with their theoretical counterparts. The MATLAB test code used is attached in Appendix C.

Each of the following subsections deals with the testing done on one of the classes.

5.1 JakesGen

The following table lists the parameters used for the various test performed on the complex gain samples generated by the JakesGen class.

Table 1: Parameters used for testing JakesGen.

Parameter	Value
Number of Scatterers	25
Block Size	10,000
Number of Blocks Generated	100
Normalized Sampling Frequency (fDts)	0.001

The testing was carried out in four steps (please note that the code used for these tests can be found in Appendix C)

5.1.1 Step One: Mean and Variance

The following table shows the measured as well as theoretical values for the Mean and Variance of the generated gain samples.

Table 2: Measured/Theoritical Mean and Variance[4].

Parameter	Measured	Theoretical
Mean	8.7113e-004 -9.4914e-004i	0
Variance	0.995	1

5.1.2 Step Two: Probability Density Function

Clearly, the real and imaginary parts of the generated gain samples should follow a Gaussian PDF. Figure 5.1-1 shows the PDF of the real part of the generated

[4] These statistics were calculated over 100 blocks of 10,000 samples each or 1,000,000 samples in total.

gains whereas Figure Figure 5.1-2 shows the theoretical Gaussian PDF (variance = ½ and mean = 0).

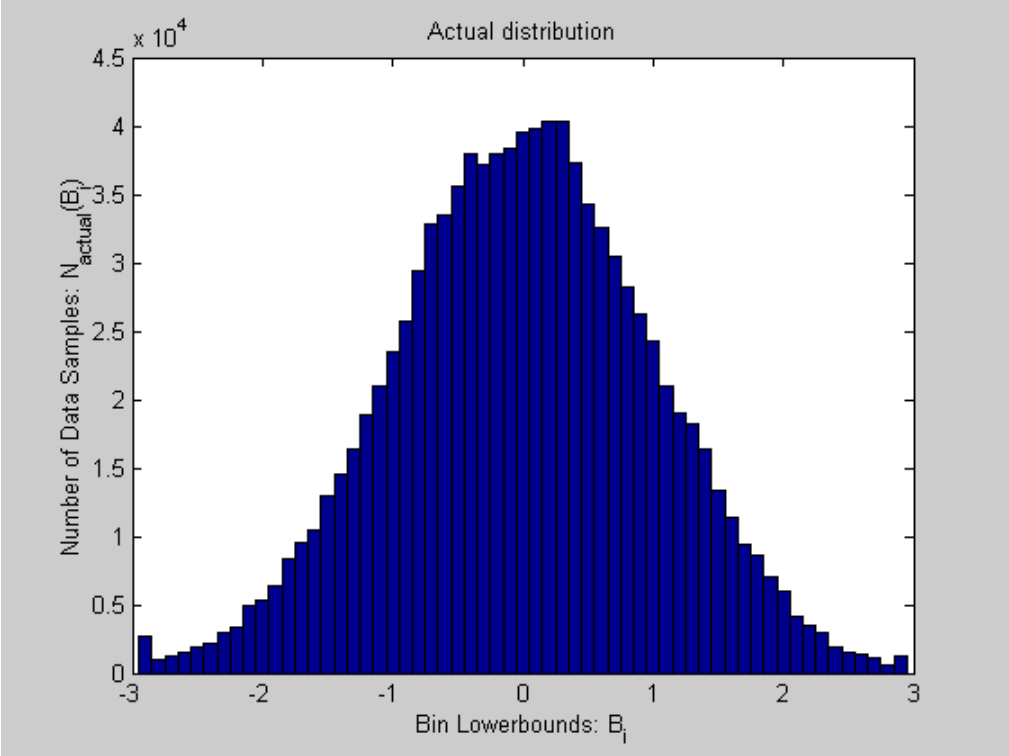


Figure 5.1-1: PDF of gain samples generated by JakesGen

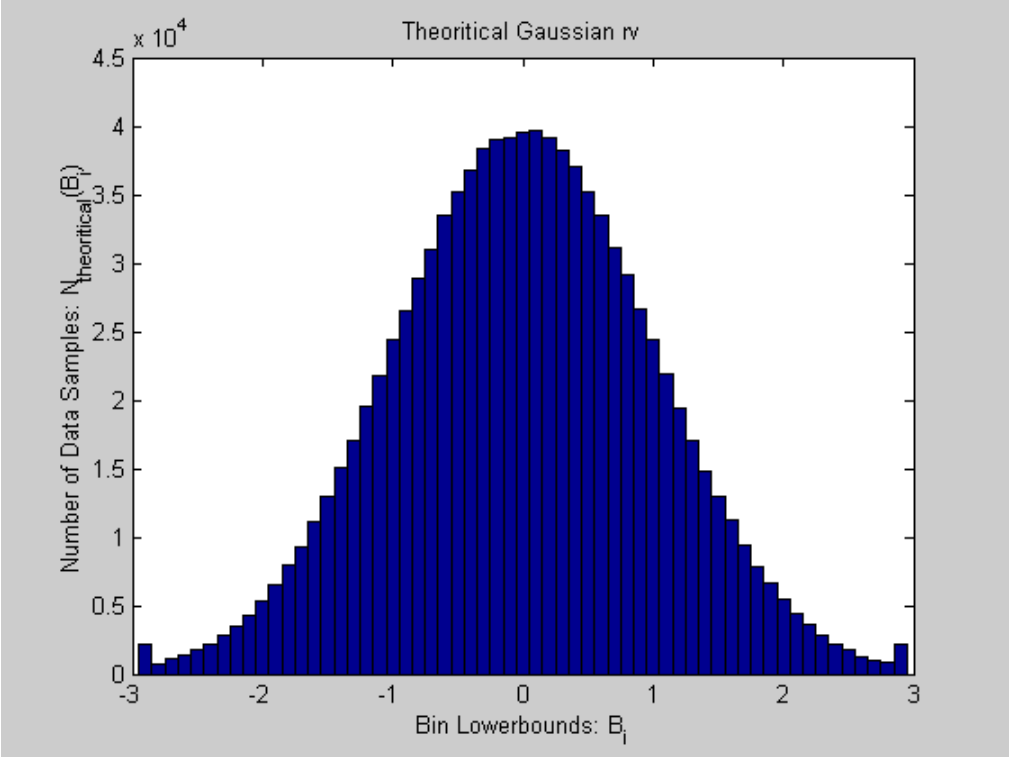


Figure 5.1-2: Theoretical Gaussian PDF

Each of the above graphs is plotted using a corresponding vector of sample frequencies. As another measure of the similarity between the theoretical and measured results, the difference between these two vectors is plotted in Figure 5.1-3.

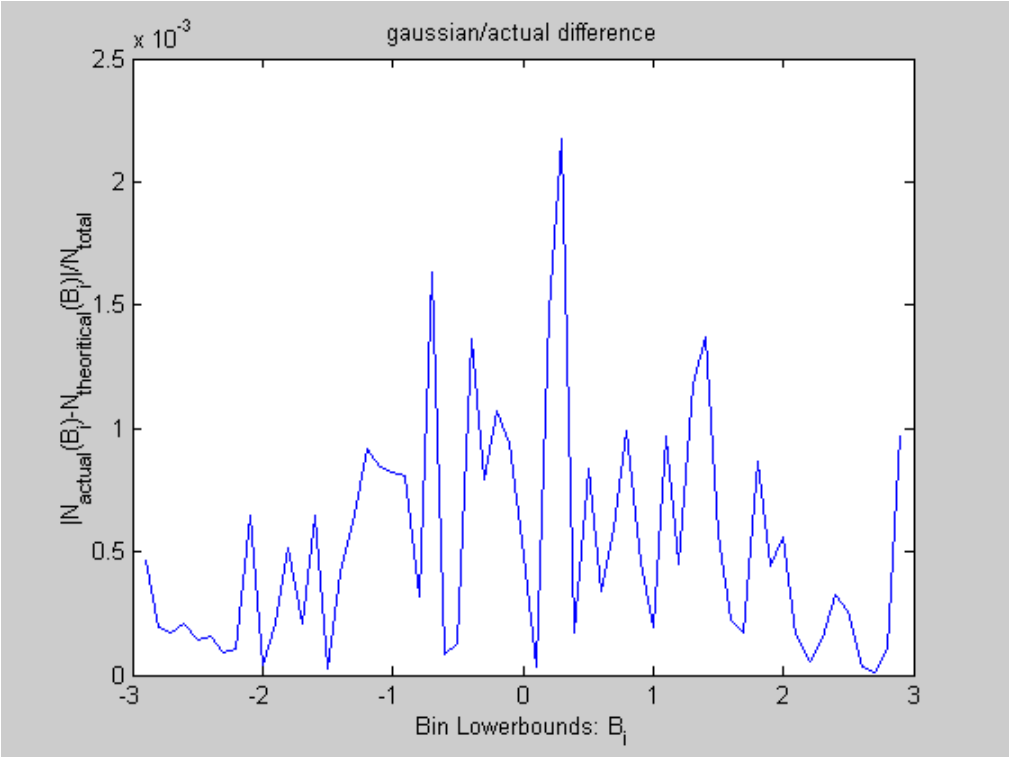


Figure 5.1-3: Difference between actual and theoretical PDFs

Alternatively, we perform the Chi-squared goodness-of-fit test to compare the actual PDF and the theoretical Normal PDF. To this end, we use the following formula to calculate the χ^2 statistic:

$$\chi^2 = \sum_{i=1}^{k} (E_i - O_i) / E_i \qquad \textbf{(5.1)}$$

where *k* is the number of bins, E_i is the expected frequency in bin *i* and O_i is the observed frequency in bin *i*. We then use the formula

$$P = 1 - C_{k-1}(\chi^2) \qquad \textbf{(5.2)}$$

where C_{k-1} denotes the CDF of the χ^2 distribution with *k-1* degrees of freedom, to obtain the P-value.

In this case we achieve a P-value of 0.1076 which is large enough for us to conclude that the PDF of our gain samples is in fact Gaussian.

5.1.3 Step Three: Autocorrelation

In this step we compare the measured autocorrelation to the theoretical one given in Equation 2.8. Figure 5.1-4 shows a plot of the difference between the real part of the measured and theoretical autocorrelation vectors (please note that, in order to be able to zoom in sufficiently, only the middle 20,000 samples are shown.)

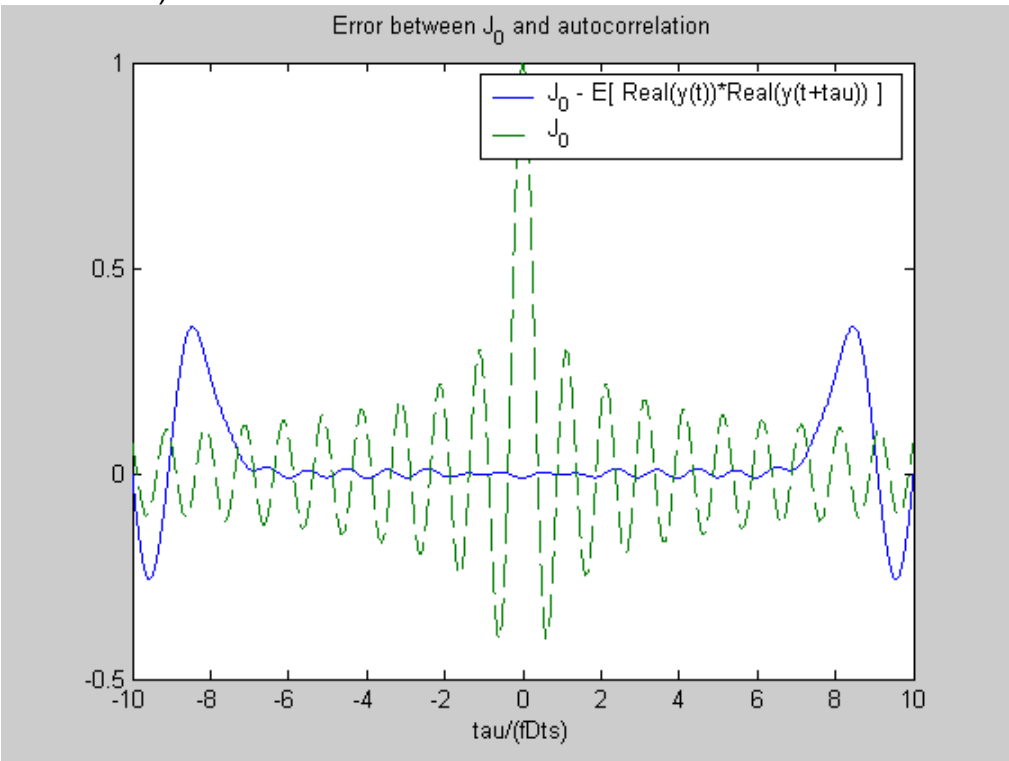


Figure 5.1-4: Difference between actual and theoretical autocorrelation.

The mean and variance the difference vector plotted above were measured to be 0.000157 and 0.005998 respectively. These statistics indicate that the measured and theoretical autocorrelation functions are indeed close.

5.1.4 Step Four: Cross-Correlation Between Real and Imaginary Parts

As discussed in Section 3.2.2, the real and imaginary components of the generated gain samples should be completely uncorrelated. The plot of the cross correlation function between the real and imaginary parts of the generated gain, shown in Figure 5.1-5, indicates the two *are* in fact uncorrelated.

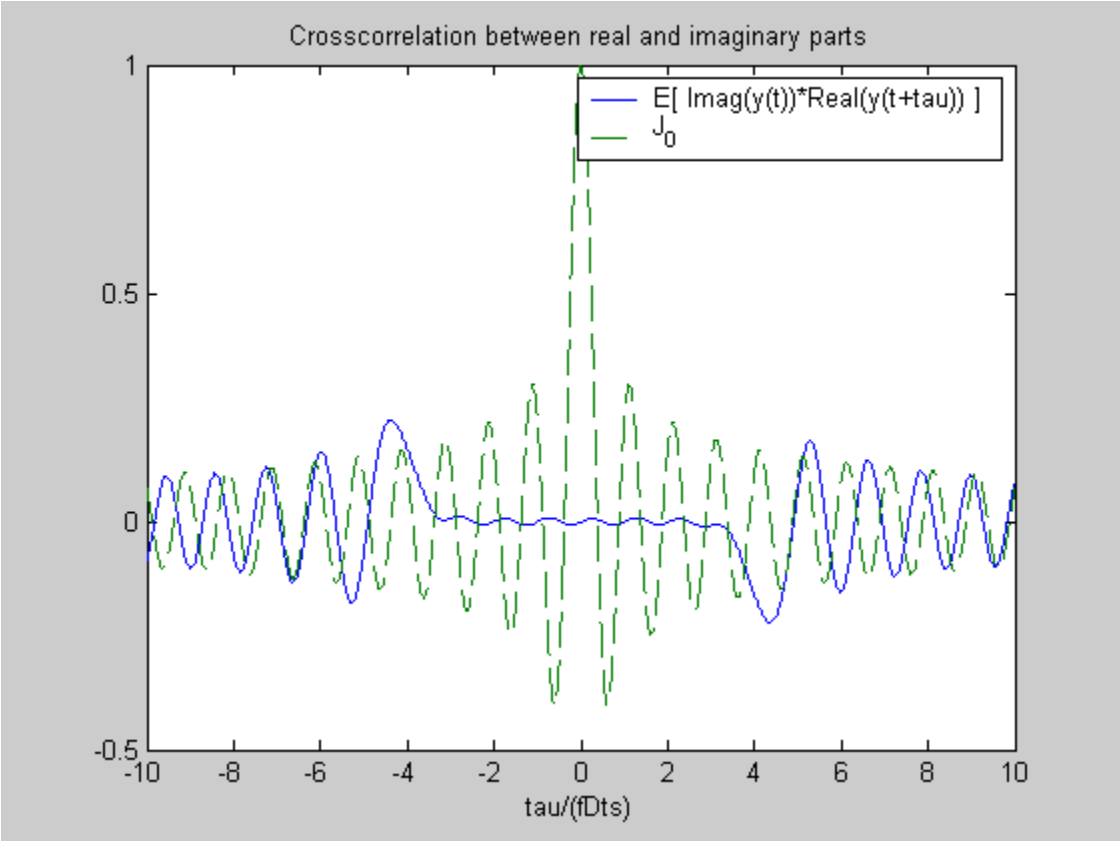


Figure 5.1-5: Cross-correlation between real and imaginary components.

The mean and variance the cross-correlation vector plotted above were measured to be -4.1342e-007 and 0.001627 respectively, which also indicate that the real and imaginary components are uncorrelated.

5.2 FWNGen

The following table lists the parameters used for the various test performed on the complex gain samples generated by the FWNGen class.

Table 3: Parameters used for testing FWNGen.

Parameter	Value
Block Size	10,000
Number of Blocks Generated	100
Normalized Sampling Frequency (fDts)	0.01
Beta (used for Kaiser Windowing)	5

The testing procedure used was almost identical to the one used for the JakesGen class, and was also carried out in four steps (please refer to Appendix C for the test code.)

5.2.1 Step One: Mean and Variance

The following table shows the measured as well as theoretical values for the Mean and Variance of the generated gain samples.

Table 4: Measured/Theoretical Mean and Variance.

Parameter	Measured	Theoretical
Mean	-0.0037 - 0.0068i	0
Variance	0.9637	1

5.2.2 Step Two: Probability Density Function (PDF)

Clearly, the real and imaginary parts of the generated gain should follow a Gaussian PDF. Figure 5.2-1 shows the PDF of the real part of the generated gain samples whereas Figure 5.2-2 shows the theoretical Gaussian PDF (variance = ½ and mean = 0). Please note that the PDF of the gain samples generated by FWNGen is somewhat more Gaussian than its counterpart, generated by JakesGen (see Figure Figure 5.1-1.)

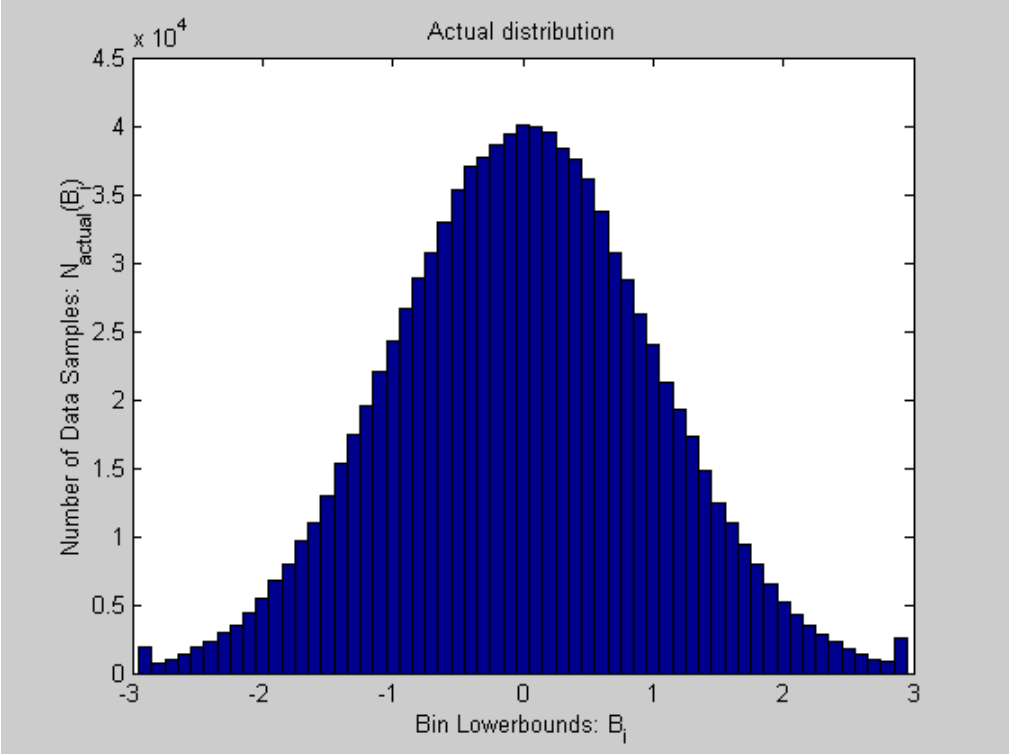


Figure 5.2-1: PDF of gain samples generated by FWNGen.

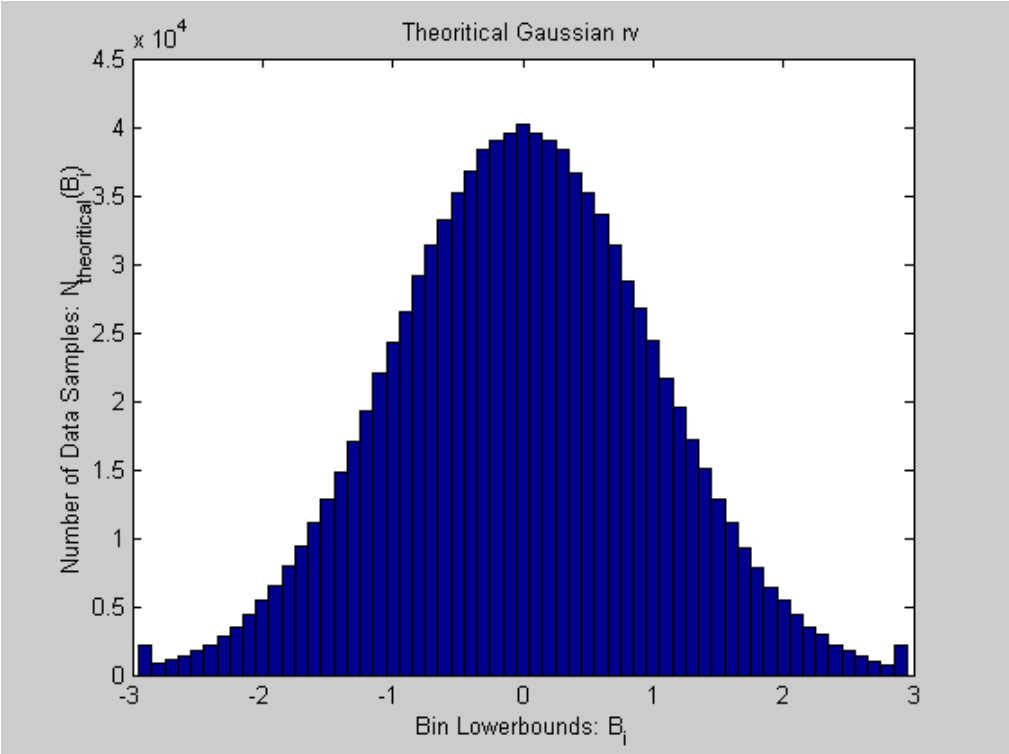


Figure 5.2-2: Theoretical Gaussian PDF.

Each of the above graphs is plotted using a corresponding vector of sample frequencies. As another measure of the similarity between the theoretical and measured results, the difference between these two vectors (of relative frequencies) is plotted in Figure 5.2-3.

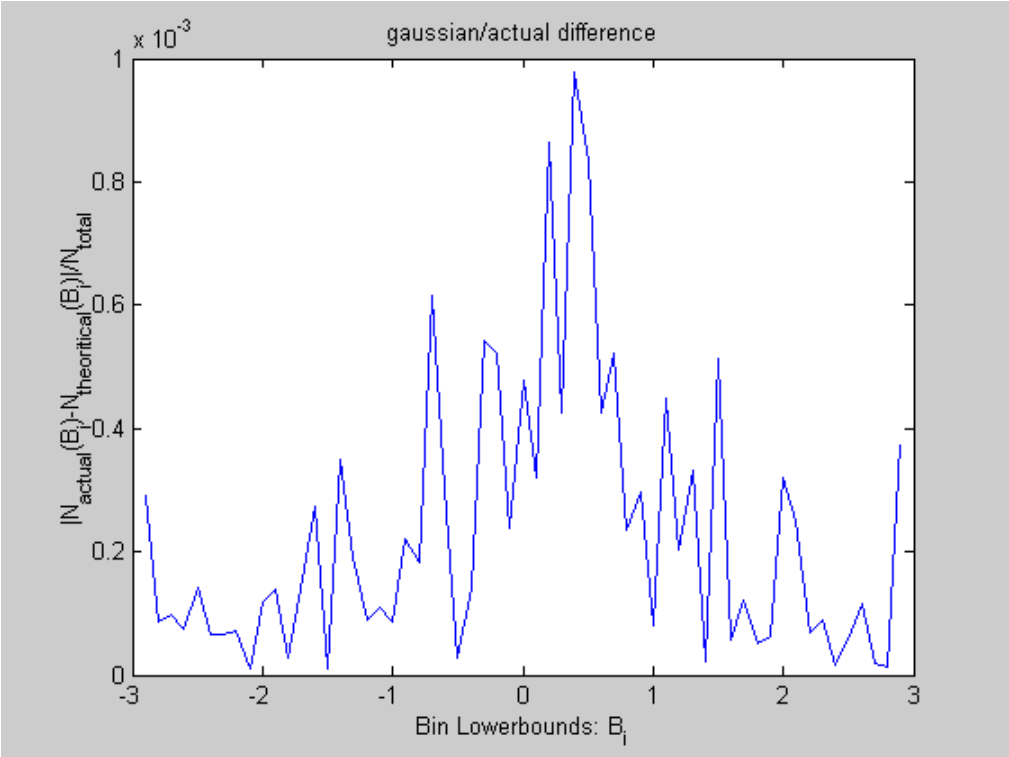


Figure 5.2-3: Difference between actual and theoretical PDFs.

Alternatively, we perform the Chi-squared goodness-of-fit test to compare the actual PDF and the theoretical Normal PDF. Using the same process described in Section 5.1.2, we calculate a P-value of 0.1455 which is large enough for us to conclude that the PDF of our gain samples is in fact Gaussian.

5.2.3 Step Three: Autocorrelation

In this step we compare the measured autocorrelation to the theoretical one given in Equation 2.8. Figure 5.2-3 shows a plot of the difference between the real part of the measured and theoretical autocorrelation vectors (please note that, in order to be able to zoom in sufficiently, only the middle 2,000 samples are shown.)

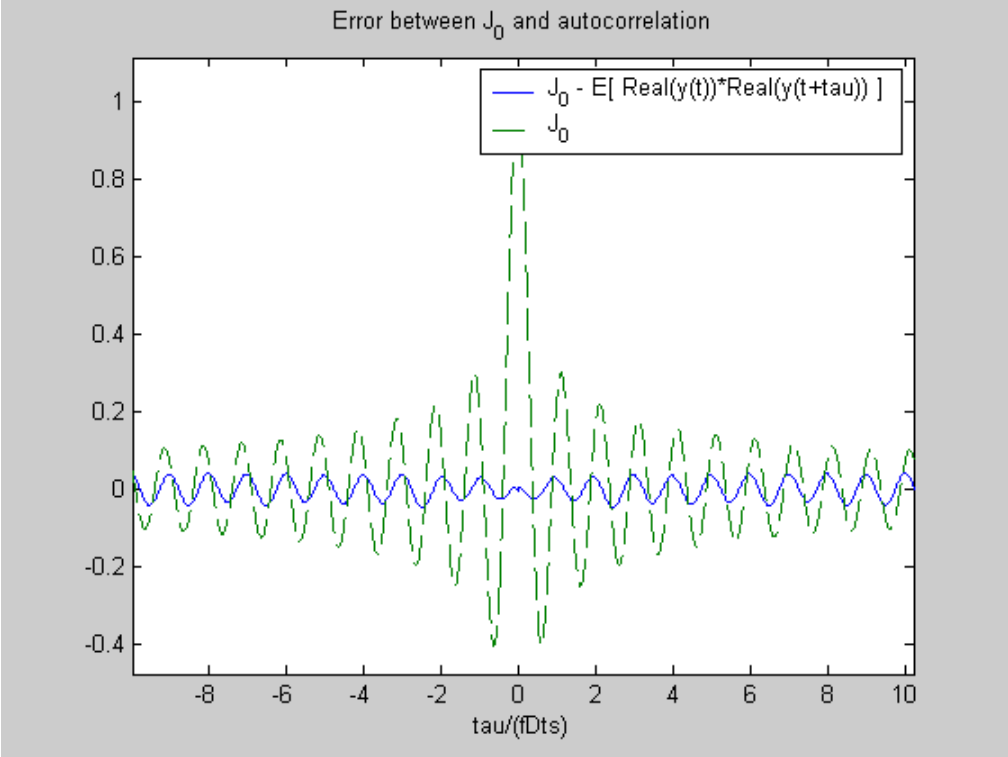


Figure 5.2-4: Difference between actual and theoretical autocorrelation.

The mean and variance the difference vector plotted above were measured to be -0.000014 and 0.000056 respectively. These statistics indicate that the measured and theoretical autocorrelation functions are indeed close.

5.2.4 Step Four: Cross-Correlation Between Real and Imaginary Parts

As discussed in section 3.2.2, the real and imaginary components of the generated gain samples should be completely uncorrelated. The plot of the cross correlation function between the real and imaginary parts of the generated gain, shown in Figure 5.2-5, indicates the two *are* in fact uncorrelated.

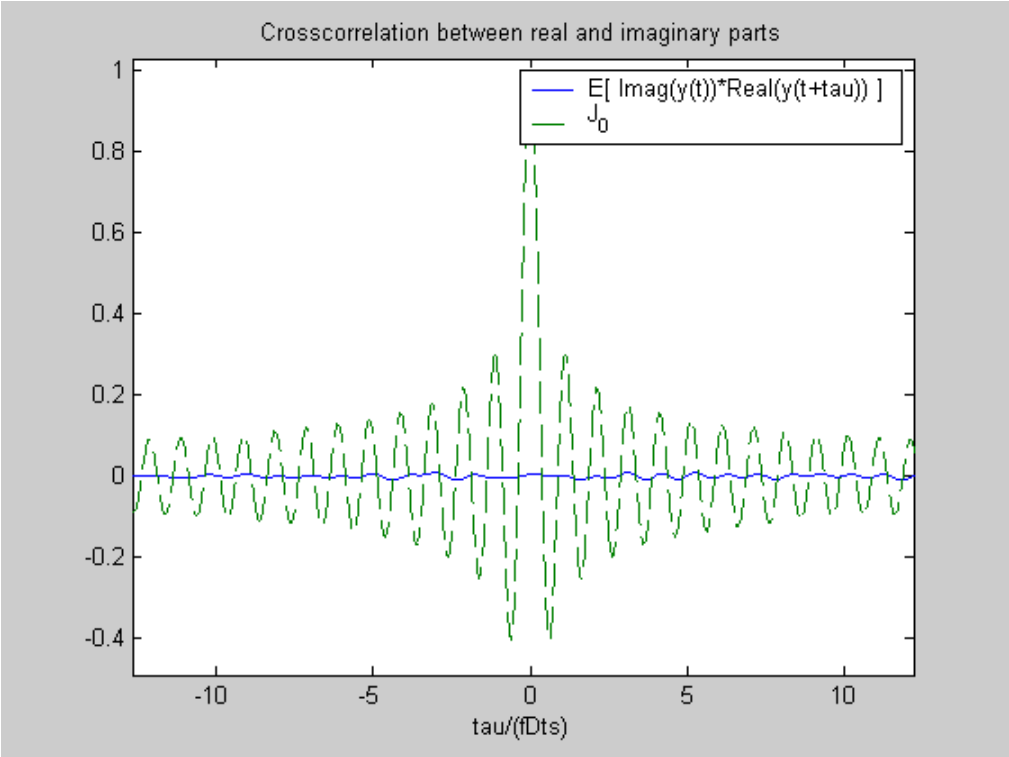


Figure 5.2-5: Cross-correlation between real and imaginary components.

The mean and variance the cross-correlation vector plotted above were measured to be 0.000012 and 0.000006 respectively, which also indicate that the real and imaginary components are uncorrelated.

5.3 TDL: Fading Simulation Example

Subsequent to performing unit testing on each of the Complex Gain Generators, we can put the whole system together for testing, which constitutes an actual Fading Simulation example. This is done by putting a known signal through the TDL and observing the results[5].

The following table lists the parameters used to setup the system for testing.

Table 5: Parameters used for testing TDL.

Parameter	Value
Sampling Frequency (fs)	320 KHz
f_D	1Hz
Number of Taps	8
Delay Spread	250us
Generator Block Size	1000
Number of Scatterers (JakesGen only)	25
Beta (FWNGen only)	10

In addition to the parameters above, we need to specify a Power Delay Profile. That is, we need to state what the complex gain power is at each tap of the TDL. Therefore, in this case, we use a linearly decaying Power Delay Profile that matches the specifications above (see Figure 5.3-1.), and also whose total power equals 1.

[5] The test scenario used here is taken from [1].

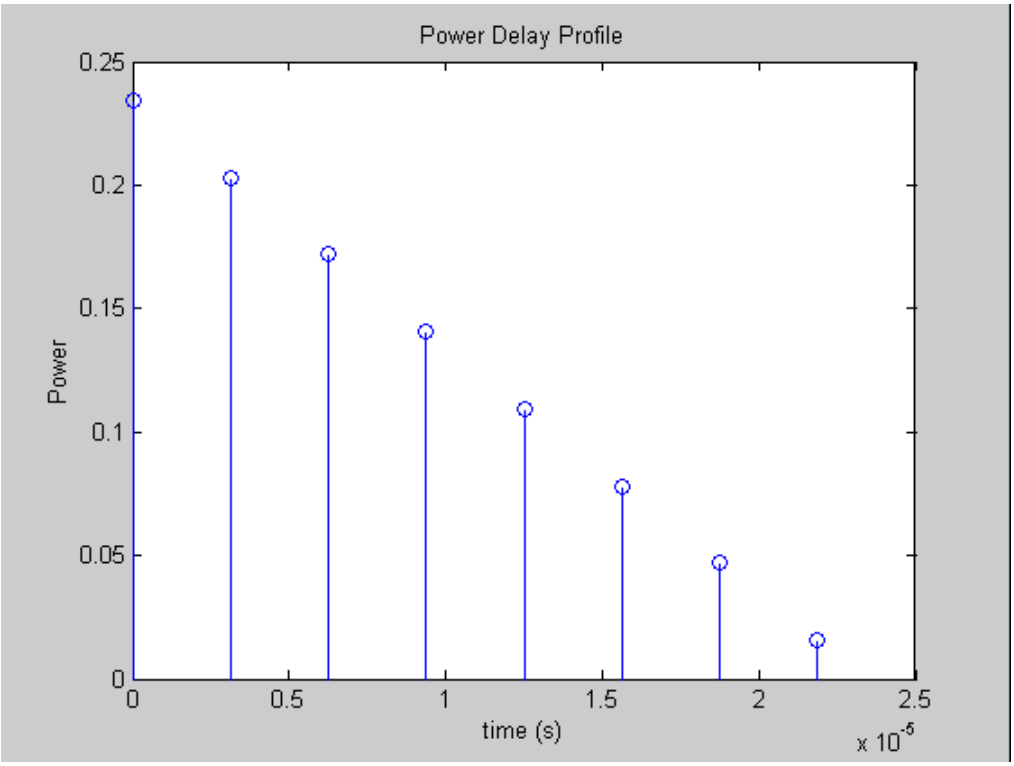


Figure 5.3-1: Power Delay Profile.

Now that we have defined the parameters of the system we should define our input signal. To generate a sample input signal, we phase modulate a complex carrier using a cosine:

$$s(t) = e^{j.2\pi.\mathrm{mod}\,Index.\cos(2\pi.f_m.t)} \quad \textbf{(5.1)}$$

The parameters used for the input signal are given in the Table 6 and the signal is plotted in Figure 5.3-1.

Table 6: Input signal parameters

Parameter	Value
Modulation Index	0.4
Modulation Frequency	10Khz
Delay Spread	250us
Number of input samples	100

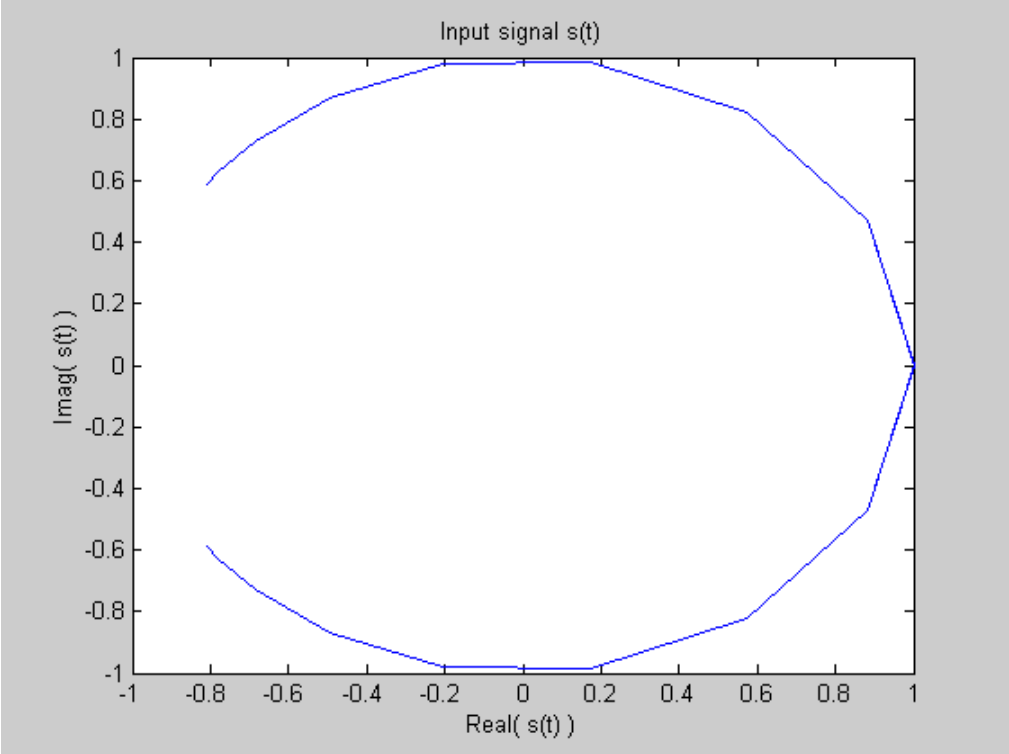


Figure 5.3-2: Input signal.

Finally, we simply pass the signal through the TDL and plot the results (see Figure 5.3-3[6]).

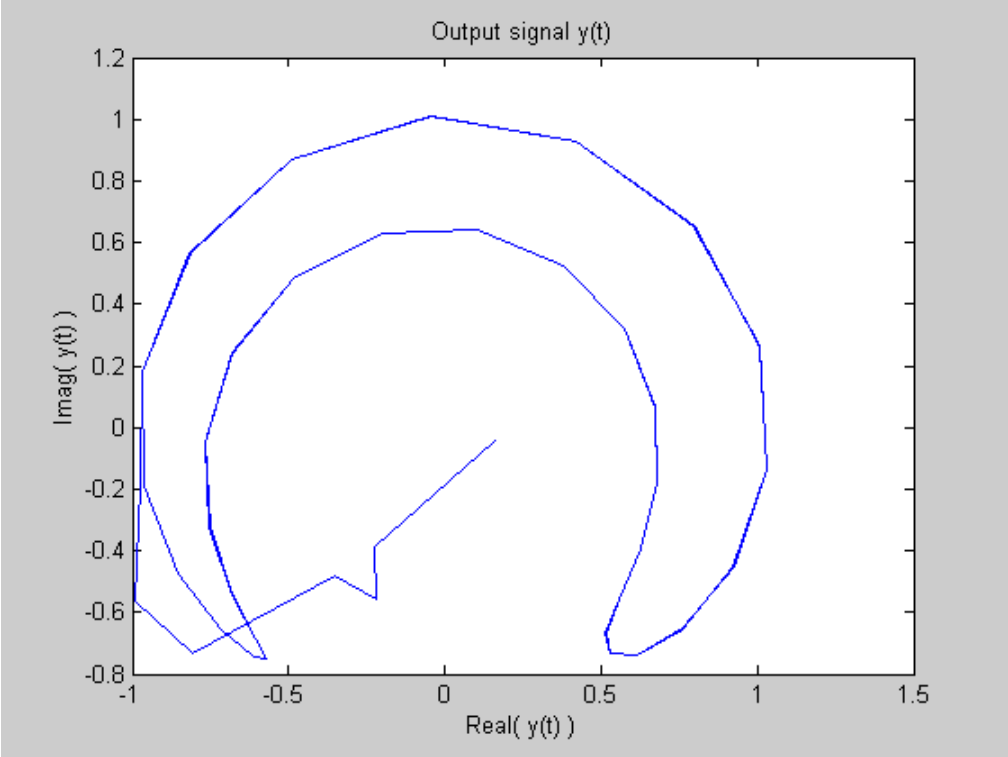


Figure 5.3-3: Output of the Fading Channel Simulator.

In addition, for the sake of comparison we generate another plot using a smaller delay spread (125us), and consequently less taps in the TDL (4 taps to be exact.) As expected, when using a smaller delay spread, the effects of fading are somewhat less destructive to the signal.

[6] Note that the plot contains a "spike" going from the center of the graph towards the bottom right. This is caused by the transient component of the output of the convolution and shows the initial contents of the convolution buffer. In fact, this result is even expected in real physical systems.

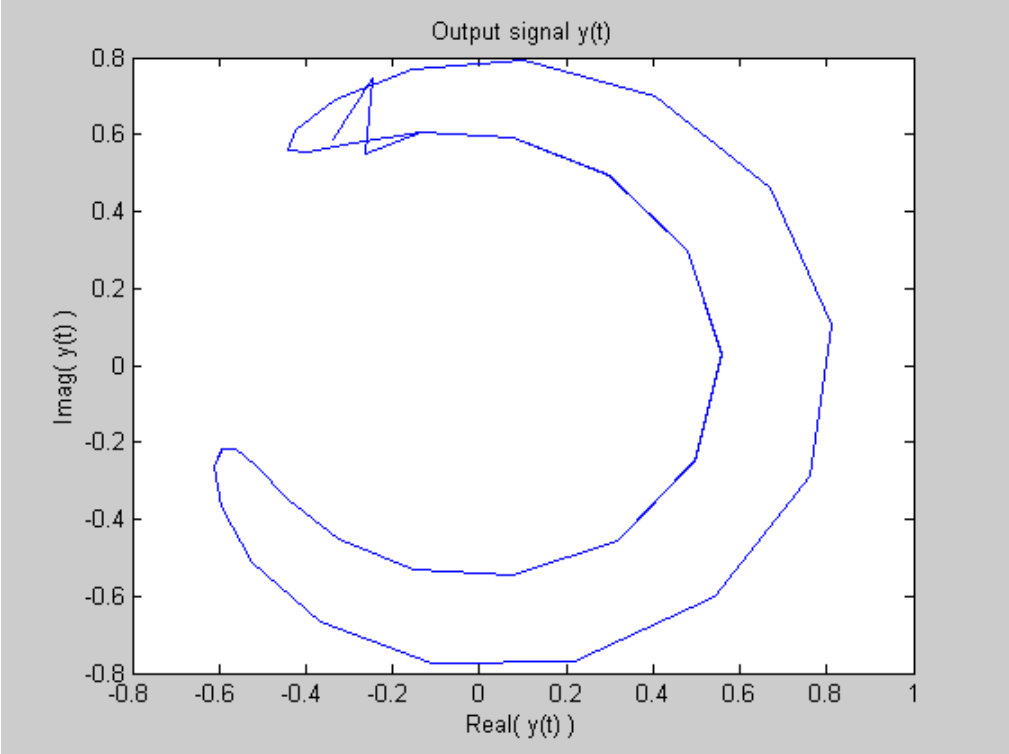


Figure 5.3-4: Output of the Fading Channel Simulator (4 taps.)

A simple plot of the spectrum of the input signal reveals that the bandwidth is approximately 50Khz while the delay spread (τ_d) is $250\mu s$ ($\tau_d = 125\mu s$ in the case of four taps). Consequently, the product of these two quantities (bandwidth times delay spread) is large, which shows that the channel is a very *frequency selective* one. The effects of this frequency selectivity are evident from both Figure 5.3-3 and Figure 5.3-4: in both cases the signal is significantly damaged.

On the other hand, our observation interval of 312.5us (100 samples at 320Khz) is quite short compared to our Doppler frequency of 1Hz (the product of the two quantities is very small). Therefore, we are faced with *slow fading,* the consequence of which is the fact that damages to the time structure of the signal (e.g. its periodicity) are small (refer to Figure 5.3-3 and Figure 5.3-4).

6 Conclusions

The research and development carried out for the purpose of this thesis was focused on creating a Fading Channel Simulator in MATLAB. This simulation models a simplified view of the Fading phenomenon encountered in communications systems. By being embedded in other MATLAB programs, the simulation can prove to be very useful for measuring the performance of various communication techniques.

The work mainly consisted of creating a Tapped Delay Line structure to implement a time varying linear filter and also implementing two different techniques of Complex Gain Generation: The Jake's Method and The Method of Filtered White Noise. To this end, an Object Oriented architecture was designed, consisting of a number of autonomous classes, which can be used together in a system, or otherwise independently. Various advantages of this design such as Modularity and Extendibility were explored.

Furthermore, extensive testing was carried out at the unit and also at the system level. These tests ensured that each component as well as the whole system performed as expected.

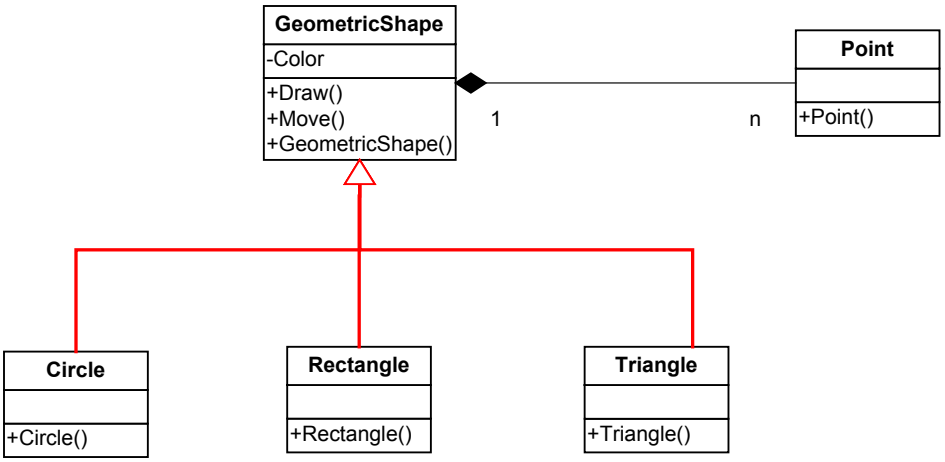
The end result of this project is a MATLAB class library. In order to improve the performance of the various functions included, in the future, some or all of the functions could be ported to a more low level programming language such as C. Furthermore, additional Complex Gain Generator classes can be implemented to support other types of Complex Gain Generation.

Appendix A: A Quick Intro to Object Orientation

An Object-Oriented program consists of *Objects* and *Classes*. Classes are specifications about a given type of data (hence, they are also referred to as *Abstract Data Types.*) A class is a "package" or "bundle" consisting of a number of functions and data variables; these are referred to as *member functions* and *member variables* respectively.

An *Object* is an instance of a class. Any number of objects can be created from a given class, each having its own member variables while sharing the same member functions. As an analogy, if we consider the 2.0 MHz Intel Pentium 4 Processor a class, then the one I have in my computer is an instance and therefore an object of this class.

The following figure shows an example of simple class diagram using UML (Unified Modeling Language.) The diagram illustrates a number of important Object-Oriented concepts, which are discussed below.



A 1: A simple class diagram.

In the diagram, each class is represented by a box, which is divided into three sections: the top section contains the class name, the middle section lists the member variables while the bottom section lists the member functions. Notice that every class has a member function with the same name as the class; these are called the *constructors.* A constructor is used to create and initialize an instance (object) of a class.

The Circle is in fact, a special type of Geometric Shape –i.e. it has all the properties of a Geometric Shape as well as its own properties. Therefore, there is an inheritance relationship between GeometricShape and Circle, and as a result, the class Circle inherits all the member variables and member functions of GeometricShape -this indeed makes sense because a Circle too can have a **color**

and Circle too can have a **move** and a **draw** member function. The same idea holds for the Rectangle and Triangle classes.

Another very important point to consider is that it does not make sense for the class GeometricShape to provide an implementation for the Draw member function, it is only there to define the interface. *Derived classes* like Circle or Rectangle provide their own specific implementation for the **Draw** member function inherited from GeometricShape. Now, if we had a list containing a number of Circle, Rectangle and Triangle objects, we could draw them without knowing of which exact type they are, by simply calling the **Draw** member function of each. This concept is referred to as *polymorphism*: the same function takes different meanings in different classes.

The last relationship to consider in Figure A1 is the *1-n aggregation* between the GeometricShape and the Point classes. This *association* indicates that a given instance of GeometricShape can own and use more than one instance of the class Point. This is modeling the fact that any given Geometric Shape (like a rectangle for example) consists of a number of points that are connected to each other.

Appendix B: Component User's Guide

This section briefly discusses how the Fading Simulation Component should be used through a number of examples[7].

B.1 Using The JakesGen Class

The following MATLAB code excerpt, illustrates how a JakesGen object is created:

```
% generator parameters
Ns=25;                  % Number of scatterers, odd
blockSize=10000;        % the block size
fDts=0.001;             % normalized sampling period
wssThreshold=.0001;     % threshold value used to determine if WSS criteria is satisfied
randomSeed=1;           % seed value used to init random number generators (optional)
Kfactor=0;              % Ricean K-factor

% create complex gain generator
gainGen = JakesGen( blockSize, fDts, Kfactor, randomSeed, Ns, wssThreshold );
```

The following line of MATLAB code shows how JakesGen can be used to generate a block of 10,000 (the block size) complex gain samples into an output vector:

```
[gainGen, y] = generate( gainGen, 0 );
```

Please note that the first argument passed to generate indicates which Complex Gain Generator object is used, while the second argument is a sequence number. Each generated block is identified with a sequence number (i.e. the blocks follow each other sequentially.) Using the sequence number the complex gain generation can be run forwards or backwards in time.

Furthermore, note that gainGen appears as a return value as well as an argument. This allows generate to update the state of the gain generator.

[7] The classes discussed here contain implementations of a number of algorithms provided in the book 'Mobile Channel Characteristics' by James K. Cavers, Kluwer Academic Publishers Inc. In particular, Appendix B of the book contains information about the Jakes Method, while Appendix K provides a description of the Filtered White Noise Method.

B.2 Using the FWNGen Class

The following MATLAB code excerpt, illustrates how the FWNGen object is created:

```
% generator parameters
blockSize=10,000;       % block size
fDts=0.01;              % normalized sampling period
beta = 5;               % beta value used for the Kaiser Window
randomSeed=1;           % seed value used to init random number generators (optional)
Kfactor=0;              % Ricean K-factor

% create complex gain generator
gainGen = FWNGen( blockSize, fDts, Kfactor, randomSeed, beta );
```

The following lines of MATLAB code shows how FWNGen can be used to generate a block of 10,000 (the block size) complex gain samples into an output vector:

```
[gainGen, y] = generate( gainGen, 0 );
```

Please note that the first argument passed to generate indicates which Complex Gain Generator object is used, while the second argument is ignored. Also, note that gainGen appears as a return value as well as a parameter to allow generate to update the state of the gain generator.

B.3 Using the TDL Class

The following MATLAB code excerpt, illustrates how a TDL object is created:

```
% create complex gain generator
gainGen = JakesGen( blockSize, fDts, Kfactor, randomSeed, Ns, wssThreshold );

% create the power delay profile, 4 taps
Ps = [ 0.5 0.25 1.5 0.1 ];

% the K-facors
Kfactors = [ 0 1 1 0 ];

% threshold value used to check whether two Jakes Generators are correlated
corThreshold = 0.01;

% create the TDL
myTDL = TDL( Ps, Kfactors, gainGen, corThreshold  );
```

As can be seen from the above code excerpt, to initialize the TDL it is necessary to create a “sample” gain generator object and pass it as the second parameter the constructor of TDL. The constructor will then “clone” a copy of this parameter for each tap. This is how the type and specifications of the Complex Gain Generator used in the TDL is determined.

In order to create a TDL consisting of FWNGen Complex Gain Generators instead, the first line of the above code excerpt should be replaced by the following line:

```
gainGen = FWNGen( blockSize, fDts, Kfactor, randomSeed, beta );
```

The following lines of MATLAB code illustrate how a given input signal is processed through the TDL:

```
% input signal
s = exp( j * 2*pi*modIndex*cos(2*pi*fm*t) );

% process the input
[myTDL, y] = process( myTDL, s );
```

After the completion of this code sequence the vector y will contain the results. Please note that, since the **process** member function changes the state of the TDL, in addition to the processed vector y, the function returns the new TDL –this is why myTDL is also on the left hand side of the assignment operator.

B.4 Creating a Set of Uncorrelated Complex Gain Generators

In order to create a set of uncorrelated Complex Gain Generators, we need to make sure that each new Complex Gain Generator we put into the set is uncorrelated to all other generators in the set. This is performed by calling the **isCorrelated** member function of the ComplexGainGen on the newly created generator and all other generators in the set.

The process outlined above has been gathered in a general-purpose function called createUncorGenerators. The following code excerpt shows how this function is used:

```
% create complex gain generator
gainGen = JakesGen( blockSize, fDts, Kfactor, randomSeed, Ns, wssThreshold );

% threshold value
threshold = 0.01;

% create array of generaros
v = createUncorGenerators( 10, gainGen, threshold )
```

Much like creating a TDL outlined in B.3, a sample generator is created first. Then, this sample generator is passed, along with the number of generators to be created, to the createUncorGenerators function where the required number of generators is created by “cloning” the sample generator.

The third parameter passed to createUncorGenerators is a threshold value used to determine if two generators are indeed correlated. For example, in the case of

JakesGen, the parameter is used to determine if the initial arrival angles of two generators are “too close”.

Appendix C: Source Code

C-1. ComplexGainGen\ComplexGainGen.m

```
%
% Complex Gain Generator base class
%


%
% Constructor.
%
function c = ComplexGainGen( blockSize, fDts )

%
% arguments are,
%   blockSize: determines the number of samples processed at one time.
%   fDts: (fD * ts) normalized sample spacing
%
   c.blockSize = blockSize;
   c.fDts      = fDts;

   c = class( c, 'ComplexGainGen');
```

C-2. ComplexGainGen\Display.m

```
%
% Complex Gain Generator base class
%

% display the contents of the object

function display( c )

fprintf( 'BlockSize: %d\n', c.blockSize );
fprintf( 'Normalized sample spacing (fD * ts): %d\n', c.fDts );
```

C-3. ComplexGainGen\generate.m

```
%
% Complex Gain Generator base class
%

%
% function 'generate': this function returns a vector of blockSize elements that
% contains complex gain samples.
%

% The baseclass provides no implentation. Derived classes should implement this
```

```
% function
%
function y = generate( c, blockNum )
```

C-4. ComplexGainGen\get.m

```
%
% Complex Gain Generator base class
%

%
% This function allows public members to be accessed
%
function val = get( c, prop_name )

switch prop_name
   case 'blockSize'
      val = c.blockSize;
   case 'fDts'
      val = c.fDts;

   otherwise
      error([prop_name,' is not a valid property.'])
end
```

C-5. ComplexGainGen\set.m

```
%
% Complex Gain Generator base class
%

%
% This function allows public members to be changed
%
function c = set( c, prop_name, val )

switch prop_name
   case 'blockSize'
      c.blockSize = val;
   case 'fDts'
      c.fDts = val;
   otherwise
      error([prop_name,' is not a valid property.'])
end
```

C-6. FWNGen\FWNGen.m

```
%
% The Filterred White Gaussian Noise Complex Gain Generator
%
```

```
%
% Constructor, performs initilization.
% The function has the following arguments:
%   blockSize: number of samples processed at one time, also determines the window size
used when
%              truncating the impulse response of the coloring filter
%   fDts: normalized sample spacing
%   beta: the beta parameter for the Kaiser window
%
function c = FWNGen( varargin )

switch nargin
case 3

   % get arguments
   %
   blockSize = varargin{1};
   fDts = varargin{2};
   beta = varargin{3};

   %
   % Initialize
   %

   fprintf( 'generating time domain coloring filter...\n' );
   fprintf( '0%%-------50%%-------100%%\n' );

   % get the truncated time domain coloring filter
   for i=[1:blockSize]
      timeFilter( 1, i ) = quad( @coloringfilter, 0, pi/2, [], [], (i-(blockSize-1)/2)*fDts );

      % print progress indicator
      if mod((i/blockSize), .05) == 0
         fprintf( '>' );
      end;
   end;

   fprintf( '\n' );

   % apply the Kaiser window
   timeFilter = timeFilter .* rot90(kaiser( blockSize, beta ));

   % normalize to unit sum of squares
   sumSquared = sqrt( sum( abs(timeFilter).^2 ) );
   dummy = timeFilter / sumSquared;
   timeFilter = dummy;

   % Get the FIR filter in frequency domain
   % Need to pad with zeros to make a block twice the size
   c.filter = fft( [ timeFilter zeros( 1, blockSize ) ] );

   % initialize postCursor (used in the overlap-and-add method)
```

```
    % to zero, so that it can be used the first time generate
    % is called.
    c.postCursor = zeros( 1, blockSize );

    % create base class
    gainGen = ComplexGainGen( blockSize, fDts );

    % create this class
    c = class( c, 'FWNGen', gainGen );

case 1

    %
    % Create an object using the same properties as the object
    % passed in
    %

    % source object
    source = varargin{1};

    % get base class properties
    blockSize = get( source, 'blockSize' );
    fDts = get( source, 'fDts' );

    % copy the filter
    c.filter = source.filter;
    c.postCursor = zeros( 1, blockSize );

    % create base class
    gainGen = ComplexGainGen( blockSize, fDts );

    % create this class
    c = class( c, 'FWNGen', gainGen );

otherwise
    error( 'Wrong number of input arguments' );

end
```

C-7. FWNGen\generate.m

```
%
%  Filtered White Gaussian Noise Generator class
%

%
% function 'generate': this function returns a vector of blockSize elements that
% contains complex gain samples.
%

%
% We're overriding the parent class's 'generate' function, providing a suitable
```

```
% implementation.
%
% NOTE: the blockNum paramter is ignored here because the Filterred WGN generator
% can only generate sequentially.
%

function y = generate( c, blockNum )

fprintf( 'generating block %d...\n', blockNum );

% get the block size
blockSize = get( c, 'blockSize' );

% generate gaussian samples, pad with zeros and take fft
noiseSpectrum = fft( [randn( 1, blockSize ) + j*randn(1, blockSize), zeros(1, blockSize) ] );

% filter the white gaussian noise samples
gainSamples = ifft( noiseSpectrum .* abs(c.filter))./sqrt(2);


% add the first half to the previous postcursor and return the result
y = gainSamples(1:blockSize) + c.postCursor;

% save the second half as the postcursor for the next time
c.postCursor = gainSamples(blockSize:blockSize+1);
```

C-8. TDL\TDL.m

```
%
% The Jakes Complex Gain Generator
%

%
% Constructor, performs initilization.
% The function has the following arguments, in the order they apear here:
%   blockSize: number of samples processed at on time
%   fDts: normalized sample spacing
%   numScat: number of equispaced scatterers to be simulated
%   wssThreshold: how close to +-pi/2 can the generated angles be without making
%                 the process non-WSS
%
function c = JakesGen( varargin )

switch nargin
case 4

   % get arguments
   %
   blockSize = varargin{1};
   fDts = varargin{2};
   c.numScat = varargin{3};
   c.wssThreshold = varargin{4};
```

```
case 1
   %
   % Create an object using the same properties as the object
   % passed in
   %

   % source object
   source = varargin{1};

   % get base class properties
   blockSize = get( source, 'blockSize' );
   fDts = get( source, 'fDts' );
   c.numScat = source.numScat;
   c.wssThreshold = source.wssThreshold;

otherwise
   error( 'Wrong number of input arguments' );
end

   %
   % Initialize
   %

   % random phases
   for j = [1:c.numScat]
      c.phase(j) = rand(1) * 2 * pi;
   end;

   % doppler shifts
   nonWSS = 1;
   while nonWSS

      % assume WSS
      nonWSS = 0;

      % the angle of the first scatterer
      theta0 = rand(1) * 2 * pi / c.numScat;

      % generate the other numScat scatterers (equispaced)
      for j = [0:c.numScat - 1]
         theta = theta0 + j * 2 * pi / c.numScat;

         % check WSS-ness
         if (abs(theta - pi/2) < c.wssThreshold)  |  (abs(theta + pi/2) < c.wssThreshold)
            nonWSS = 1;
            break;
         end;

         % calc doppler shift from angle
         c.dopplerShift( j + 1 ) = -2 * pi * cos( theta );
```

```
    end;

  end;

  % create base class
  gainGen = ComplexGainGen( blockSize, fDts );

  % create this class
  c = class( c, 'JakesGen', gainGen );
```

C-9. JakesGen\display.m

```
%
%  Jakes Complex Gain Generator class
%

% display the contents of the object

function display( c )

% display parent first
display( c.complexgaingen );

fprintf( '# of Scatterers: %d\n', c.numScat );
```

C-10. JakesGen\generate.m

```
%
%  Jakes Complex Gain Generator class
%

%
% function 'generate': this function returns a vector of blockSize elements that
% contains complex gain samples.
%

%
% We're overriding the parent class's 'generate' function, providing a suitable
% implementation.
%

function y = generate( c, blockNum )

  % get block size from parent
  blockSize = get( c, 'blockSize' );
  % ...and the normalized sample spacing
  fDts = get(c, 'fDts');

  % This is the time axis (sampling instants)
  u = [blockNum*blockSize*fDts:fDts:( (blockNum+1)*blockSize - 1 ) * fDts];

  % init to zero
```

```
    y = zeros( 1, blockSize );

    % generate gain samples
    for j = [ 1 : c.numScat ]
        y = y + exp( i * ( c.phase( j ) + u * c.dopplerShift( j ) ) );
    end;

    y = y / sqrt( c.numScat );
```

C-11. JakesGen\get.m

```
%
%  Jakes Complex Gain Generator class
%

% This function allows public members to be accessed
function val = get( c, prop_name )

switch prop_name

    % access number of scatterers
    case 'numScat'
        val = c.numScat;

    % access to parent class members
    otherwise
        val = get( c.complexgaingen, prop_name );
end
```

C-12. TDL\TDL.m

```
%
% class TDL: Tapped Delay Line
%

%
% Constructor
%       powerDelayProfile: array containing points seperated by
%       ts in the power delay profile. Each is the variance of the
%       corresponding tap in the tap delay line.
%
%       sampleGenerator: is a sample object of the desired type
%       of complex gain generator. The needed generators will
%       created using the same properties as this one.
%

function c = TDL( powerDelayProfile, sampleGenerator )


% save number of taps
c.numTaps = length( powerDelayProfile );
```

```
% block size
c.blockSize = get( sampleGenerator, 'blockSize' );

% save the power delay profile
c.powerDelayProfile = powerDelayProfile;

% get the handle of the constructor of the generator
consFunc = str2func(class( sampleGenerator ));

% create an array of generators
for i=[1:c.numTaps]
   c.taps( i ).gen = feval( consFunc, sampleGenerator );
   c.taps( i ).buffer =  generate( c.taps( i ).gen, 0 );

end;

% initialize the buffer pointer
c.gainBufferIndex = 1;

% the number of the next block
c.blockNum = 1;

% the buffer used for convolution
c.buffer = zeros( 1, c.numTaps );

c = class( c, 'TDL' );
```

C-13. TDL\process.m

```
%
% class TDL: Tapped Delay Line
%

%
% process, simulates the effects of fading on the input vector
% containing signal samples.
%

function [c, y] = process( c, s )

% length of input vector
iLength = length( s );

for i = [1:iLength]

   % shift everything to right
   c.buffer = circshift( c.buffer, [0, 1] );

   % put new value in
   c.buffer(1) =  s(i);
```

```
    % get new gain values
    [c g] = getNewGainValues( c );

    % multiply out the numbers
    y(i) = sum( c.buffer .* g );

end;
```

C-14. TDL\get.m

```
%
%  Tapped Delay Line class
%

% This function allows public members to be accessed
function val = get( c, prop_name )

switch prop_name

    % access number of taps
    case 'numTaps'
        val = c.numTaps;


    otherwise
        error([prop_name,' is not a valid property.'])
end
```

C-15. JakesTest.m

```
% generator parameters
Ns=25;              % odd
blockSize=10000;
blockCount = 100;
fDts=0.001;
beta = 5;
threshold=.0001;

% create complex gain generator
gainGen = JakesGen( blockSize, fDts, Ns, threshold );


%
% Compare pdf with that of a Gaussian rv
%


% bins for histogram (pdf)
x = -2.9:0.1:2.9;

y = [];
```

```
% generate block
for i=1:blockCount
   y = [y generate( gainGen, i )];
end;

blockSize = blockSize * blockCount;

% draw pdf, making variance equal to one
figure;
pdfActual = hist(real(y)/sqrt(var(real(y))),x);
hist(real(y)/sqrt(var(real(y))),x);
title( 'Actual distribution' );

% draw pdf of unit variance gaussian rv for comparison
figure;
pdfGaussian = hist(randn(1,blockSize),x);
hist(randn(1,blockSize),x);
title( 'Theoritical Gaussian rv' );

% plot the difference between the actual and the theoritical pdfs
figure;
plot( x, abs( pdfActual - pdfGaussian)/blockSize );
title( 'gaussian/actual difference' );


%
% Compare the autocorrelation with theoritical -i.e.
% zeroth order bessel function of the first kind.
%

% x axis for autocorrelatiohn
ax = -(blockSize-1)*fDts:fDts:(blockSize-1)*fDts;

% autocorrelation (normalized)
autocor = xcorr( y )/blockSize;

% bessel function
j0 = besselj(0, ax*2*pi);

% difference between actual and theoritical
diff = (real(j0 - autocor));

% plot the 400 samples in the middle
figure;
plot(ax(blockSize-10000:blockSize+10000),diff(blockSize-10000:blockSize+10000));
title( 'Error between J0 and autocorrelation' );

fprintf( 'The difference between Theoritical/Measured Corrleations: %f\n', mean(diff) );
fprintf( '\nMean: %15f', mean(diff) );
fprintf( '\nVariance: %15f\n', var(diff) );

%
```

```
% Make sure the real and imaginary parts are uncorrelated
%
crosscor = xcorr( real(y), imag(y) )/blockSize;

figure;
plot(ax(blockSize-10000:blockSize+10000),crosscor(blockSize-10000:blockSize+10000));
title( 'crosscorrelation between real and imaginary parts' );

fprintf( '\nThe crosscorrelation between real and imaginary components:' );
fprintf( '\nMean: %15f', mean(crosscor) );
fprintf( '\nVariance: %15f\n', var(crosscor) );
```

C-16. FWNTest

```
% generator parameters
blockSize=1000;
blockCount = 100;
fDts=0.01;
beta = 5;

% create complex gain generator
gainGen = FWNGen( blockSize, fDts, beta );


%
% Compare pdf with that of a Gaussian rv
%


% bins for histogram (pdf)
x = -2.9:0.1:2.9;

y = [];

% generate block
for i=1:blockCount
   y = [y generate( gainGen, i )];
end;

blockSize = blockSize * blockCount;

% draw pdf, making variance equal to one
figure;
pdfActual = hist(real(y)/sqrt(var(real(y))),x);
hist(real(y)/sqrt(var(real(y))),x);
title( 'Actual distribution' );

% draw pdf of unit variance gaussian rv for comparison
figure;
pdfGaussian = hist(randn(1,blockSize),x);
hist(randn(1,blockSize),x);
title( 'Theoritical Gaussian rv' );
```

```
% plot the difference between the actual and the theoritical pdfs
figure;
plot( x, abs( pdfActual - pdfGaussian)/blockSize );
title( 'gaussian/actual difference' );



%
% Compare the autocorrelation with theoritical -i.e.
% zeroth order bessel function of the first kind.
%

% x axis for autocorrelatiohn
ax = -(blockSize-1)*fDts:fDts:(blockSize-1)*fDts;

% autocorrelation (normalized)
autocor = xcorr( y )/blockSize;

% bessel function
j0 = besselj(0, ax*2*pi);

% difference between actual and theoritical
diff = (real(j0 - autocor));

% plot the 400 samples in the middle
figure;
plot(ax(blockSize-1000:blockSize+1000),diff(blockSize-1000:blockSize+1000));
title( 'Error between J0 and autocorrelation' );

fprintf( 'The difference between Theoritical/Measured Corrleations:');
fprintf( '\nMean: %15f', mean(diff) );
fprintf( '\nVariance: %15f\n', var(diff) );

%
% Make sure the real and imaginary parts are uncorrelated
%
crosscor = xcorr( real(y), imag(y) )/blockSize;

figure;
plot(ax(blockSize-1000:blockSize+1000),crosscor(blockSize-1000:blockSize+1000));
title( 'crosscorrelation between real and imaginary parts' );

fprintf( '\nThe crosscorrelation between real and imaginary components:' );
fprintf( '\nMean: %15f', mean(crosscor) );
fprintf( '\nVariance: %15f\n', var(crosscor) );
```

C-17. TDLTest

```
clear all;

% generator parameters
beta=10;
```

```
threshold=.0001;
Ns=25;              % odd
blockSize=1000;

modIndex = .4;
fD = 1;
fm = 10000;

fs = 32 * fm;
ts=1/fs;

numSamples = 100;
numTaps = 4;
td = ts * numTaps;

fDts = fD / fs;

% time axis
t = [0:ts:ts*(numSamples-1)];
tPowerDelay = [0:ts:(numTaps-1)*ts];

% input signal
s = exp( j * 2*pi*modIndex*cos(2*pi*fm*t) );

figure;
plot(s);
title( 'Input signal s(t)' );

% Standard deviations taken from the Power Delay Profile
Ps=sqrt( ts*(-2*tPowerDelay-ts+2*td)/td^2 );

% Plot the Power Delay Profile
stem(t(1:length(Ps)), Ps.^2);
title( 'Power Delay Profile' );
xlabel( 'time (s)' );
ylabel( 'Power' );

% create complex gain generator
gainGen = JakesGen( blockSize, fDts, Ns, threshold );
%gainGen = FWNGen( blockSize, fDts, beta );

myTDL = TDL( Ps, gainGen );

[myTDL, y] = process( myTDL, s );

figure;
plot( y );
title( 'Output signal y(t)' );
```

Appendix D: References

1. Cavers, James K., September 2000, *Mobile Channel Characteristics,* Kluwer Academic Publishers
2. Oppenheim, Allan V. and Schafer, Ronald W., 1989, *Discrete-Time Signal Processing,* Prentice Hall Signal Processing Series
3. Bruegge, Bernd and Dutoit, Alan H, 2000, *Object-Oriented Software Engineering*, Prentice Hall
4. Harry Lass and Peter Gottlieb, 1971, *Probability and Statistics,* Addison Wesley Publishing Company, Inc.