Predicting Microwave Diffraction in the Shadows of Buildings

by

Thomas A. Russell

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

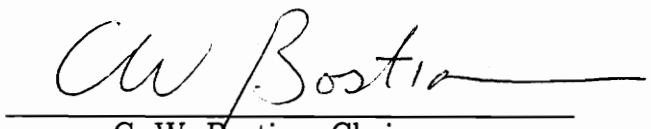
in partial fulfillment of the requirements for the degree of

Master of Science

in

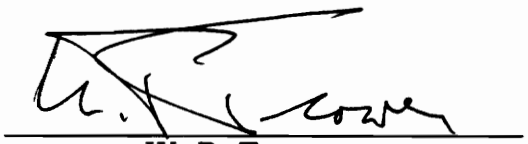
Electrical Engineering

APPROVED:



C. W. Bostian, Chairman

T. S. Rappaport



W. P. Trower

November, 1991

Blacksburg, Virginia

Predicting Microwave Diffraction in the Shadows of Buildings

by

Thomas A. Russell

Charles W. Bostian, Committee Chairman

Electrical Engineering

(ABSTRACT)

Designers of low-power radio systems for use in urban areas would benefit from the capability for accurate computer-based predictions of signal loss due to shadowing. This thesis is intended to fill a need for prediction methods that exploit a building database and consider the three-dimensional profile of the radio path. Models are presented that allow the application of Fresnel-Kirchoff diffraction theory to arbitrarily oriented buildings of simple shapes. Building location information used by the diffraction models is in a form compatible with a geographic information systems (GIS) database. Diffraction screens are constructed at all building edges, including those of both horizontal and vertical orientations, in order to consider all possible diffractions and to compute field contributions often ignored.

Multiple buildings and edges of the same building that introduce multiple successive diffractions are considered with a rigorous, recursive application of the diffraction theory that requires sampling the field distribution in each aperture. Robust and computationally efficient numerical methods are applied to solve the diffraction integrals. The software implementation of these methods is tested with example runs and comparisons with 914 MHz continuous-wave measurements taken on the Virginia Tech campus.

ACKNOWLEDGEMENTS

My sincere appreciation goes to my academic advisor, Professor Charles Bostian. His support, guidance, and friendship helped to make my research here an enjoyable experience. My thanks go to Professor Ted Rappapport for the opportunity to work in mobile radio research, and for his ideas and enthusiasm. I am grateful also to Professor Peter Trower, whose far-ranging lectures on physical optics and the physical world provided me with the inspiration to pursue this topic.

I am indebted to the Mobile and Portable Radio Research Group as a whole, in particular to Mike Keitz, Scott McCulley, and Sal Yusuf for making the measurements needed to validate the models, and most of all to Kurt Schaubach, not only for his help in processing the measured data, but for the many fruitful discussions of the behavior of radio waves set loose on the urban environment.

I am grateful to my family for their love and support, and for my Dad's insistence that I could do whatever I set my mind to. My deepest gratitude goes to my wife, Janet, a constant source of life, inspiration, and perspective, who has cheerfully accompanied me throughout this academic journey. Finally, I must also thank Thelonious Monk and John Coltrane for the spiritual lift their music provided when I needed it most.

TABLE OF CONTENTS

I. INTRODUCTION

1.1 Statement of the Problem

A possible future trend in urban-area mobile radio communications that is currently generating a great deal of interest is the use of reduced transmit power and coverage of smaller service areas. A low-power communication system allows for a reduction in size of the mobile unit, but it introduces new challenges to radio coverage prediction. With fewer buildings within the coverage area, each individual building can have a more significant impact on signal coverage, and the shadowing by these buildings will be an important consideration in the design of the proposed low-power radio systems. Measurements taken over distances of only a few city blocks in downtown Ottawa, Canada, by Whitteker [1] confirmed the importance of individual buildings in such an area, showing a 10 to 20 dB impact on received signal strength at 910 MHz.

Designers of low-power radio systems for use in urban areas would benefit from the capability for accurate computer-generated predictions of signal level, based on the effects of the particular configuration of buildings in the area. The application of environmental databases to the prediction of signal strength has been attempted successfully in the past, using large-scale digital terrain maps of mainly rural environments to develop two-dimensional radio path profiles (see

Figure 1.1, reprinted from Palmer [2]). Such a profile neglects the horizontal dimension (directed into the page in Figure 1.1) that is transverse to the line connecting the transmitter and receiver, requiring the assumption that there is no significant variation in this dimension. This is an assumption that clearly does not hold in the general urban environment where the principal obstacles are buildings and the dominant propagation path may be around the side of a building or over the roof.

This thesis is intended to fill a need for signal coverage prediction methods that exploit a building database and consider the three-dimensional profile of the radio path. As this is a complex task, the work in this thesis addresses only a single mechanism for signal transmission in the urban environment: diffraction. It is intended that this work will be combined with the work of other researchers on scattering and reflection to yield a comprehensive prediction tool.

In attempting to predict signal levels, we must distinguish between signal level variations with two different sets of characteristics. Small-scale fluctuations (fast fading) with a spatial period on the order of a half wavelength, arise from interference caused by the existence of multiple radio paths over which the transmitted wave may travel to the receiver through a complex environment. Signals may arrive at the receiver from different directions, so, as the receiver moves, the relative path lengths change. The phase of each incoming signal

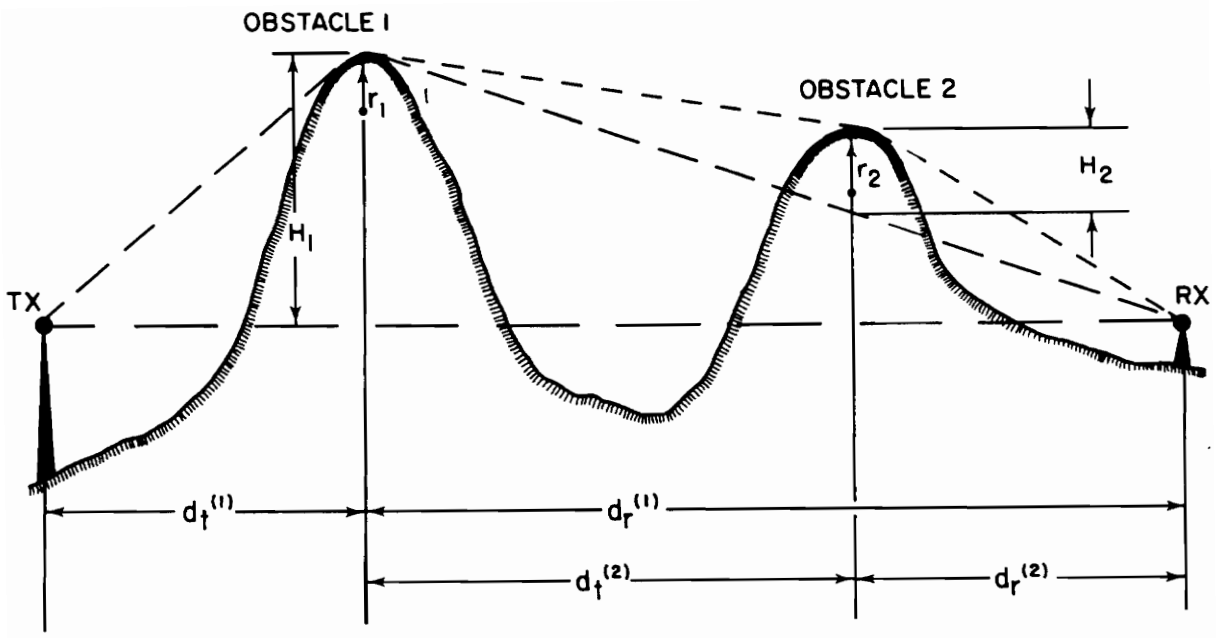


Figure 1.1. Two-Dimensional Radio Path Profile and Geometry used to Compute Terrain Diffraction (Figure 4 in Reference [2])

depends on the distance traveled, so the relative phases of the waves arriving by multiple paths will change as the receiver moves, causing a pattern of constructive and destructive interference. Two waves of the same amplitude arriving perfectly in phase by different paths will be perfectly out of phase when their relative path length has changed by a half of a wavelength, and they will then cancel each other out. At UHF frequencies the wavelength is less than a meter, which is smaller than the resolution of the terrain and building information available at this time. Accordingly, it is not feasible to predict the precise locations of peaks and nulls in the small-scale fluctuations of UHF mobile radio signals, and statistical models of these variations, based on measurements, would be more appropriate. Nevertheless, attempts to predict deterministically the small-scale fluctuations, even with imprecise geographical information, may shed some additional light on their characteristics in a given environment. Therefore, we will consider small-scale fluctuations but without emphasizing their accurate prediction.

The mean of the small-scale fluctuations is known as the local average or local mean field strength. Lee [3] showed that it can be estimated by taking a running average of signal variations over a distance of 20 to 40 wavelengths, based on a Rayleigh distribution of the small-scale fluctuations. The local average varies with changes in the amplitudes, rather than in the relative phases, of the waves incident on the receiving antenna [4]. Changes in the wave amplitude are produced by sufficient movement of one of the antennas to change

the radio path such that the propagation mechanisms are altered, or by a path length change sufficient to affect the free space loss. It is this type of signal variation that should be amenable to accurate prediction through the use of computer databases and propagation prediction models that make use of specific, rather than categorical, information about the environment. However, with interest in low-power communications coming about only recently, these approaches are only now beginning to be researched and exploited.

1.2 Overview of Thesis

The remainder of this chapter summarizes the results of this research and the contributions made in this thesis. Chapter 2 provides an overview of the literature on the various topics addressed in this work, and describes where this research fits in with published work. The suitability of the Geographic Resources Analysis Support System (GRASS), a Geographic Information Systems (GIS) package, for propagation prediction is investigated in Chapter 3. Algorithms for searching the database for diffracting buildings are also described in Chapter 3.

Chapter 4 presents the development of the simplified form of the Kirchoff diffraction integral, methods for modeling building edges, the interaction with the database, example calculations, and finally some comparisons with measurements. Chapter 5 extends the methods of Chapter 4 to include the possibility of multiple successive diffracting obstacles. Examples are presented

and comparisons are made with measurements. The numerical methods used to implement the models of Chapters 4 and 5 are presented in Chapter 6. Chapter 7 addresses some software issues, and Chapter 8 briefly considers reflections and the interaction of reflection models with these diffraction models. Conclusions are presented in Chapter 9, references in Chapter 10, and a program listing is included in the appendix.

1.3 Summary of Results

The following is a summary of the results obtained when the models and methods proposed in this thesis were implemented in software and tested:

1. In all of the building configurations tested, the diffracting building edges were successfully located and the multiple diffractions were computed in 5-10 seconds per edge on a 386SX computer. The most likely potential problems with geometries not yet tested and possible solutions have been identified.

2. The computational precision in the tests was about ± 1 dB, and error due to modeling assumptions may be up to 5 dB.

3. Measurements of the effects of buildings on 914 MHz propagation show good agreement with the onset of signal loss due to shadowing, but in two cases tested, the predicted diffraction field was much weaker than the measured signal strength in deeply shadowed regions. It is likely that this is due to alternate,

low-power sources of energy, such as scattering from trees, that overwhelm the diffraction field when the latter has dropped to very low values (20-30 dB down from free space).

1.4 Contributions Made in Thesis

The contributions made in this thesis include proposed methods for:

1. Accessing a building database
2. Identifying the diffracting buildings
3. Identifying and ordering the diffracting edges of a building according to whether the edges diffract the field multiple times or diffract different portions of the wavefront a single time
4. Modeling arbitrarily oriented building edges in a form that allows application of Fresnel-Kirchoff diffraction theory
5. Ensuring convergence of the semi-infinite integrals necessary in the evaluation of the diffraction field

Contributions of other types include the application of Fresnel-Kirchoff theory in its three-dimensional form to buildings, and the modification of the multiple diffraction evaluation methods of Whitteker to allow for prediction of diffraction by edges of two different general orientations: vertical and horizontal.

II. OVERVIEW OF THE LITERATURE

2.1 Propagation Prediction Based on Geographical Information

The model published by Okumura [5] in 1968 for predicting mobile radio propagation over land includes the effects of many geographical characteristics, such as terrain slopes, isolated ridges, land-use classifications (such as urban or rural), and street orientations, all in the form of graphical correction curves. Hata [6] derived simple formulas based on Okumura's curves, allowing direct computation by computer, although, as in the original model, the inputs were manually derived. Durkin [7] used a computer database to store elevation data in a grid pattern, with a spacing of 500m per grid cell, and employed an interpolation method to construct the path profile for any transmitter and receiver locations. Durkin's computer program used this profile to determine if there was sufficient clearance over the intervening terrain, and if not, the program applied the empirical methods of Longley and Rice [8] to compute the shadowing loss. By choosing many points along each of many radials surrounding a possible transmitter location, an area coverage prediction was generated.

Durkin's technique for utilization of a computer database in area coverage prediction was expanded upon in the work of Palmer [2] and Whitteker [9] of the

Communications Research Center (CRC), among others. The CRC model is the most sophisticated of its type according to a recent comparative study [10], and uses ground cover codes attached to each grid cell to provide a large amount of information about the path profile. The ground cover codes are used to estimate the complex reflection coefficient at each point along the path, to determine by how much the terrain elevation at each profile point should be increased to account for buildings or trees in the path, and to determine overall path profile characteristics such as the percentage of water along the path and a general classification of the path, such as "low-density urban". This information is applied to empirical formulas, such as Hata's, derived from extensive measurements in various environments, to determine the propagation effects. The impact of individual terrain peaks are accounted for in a rigorous fashion with diffraction models, and specific, small-scale obstacles, such as buildings, near one of the antennas are accounted for in an approximate way, "subject to large uncertainties." [9]

Recently the resolution of environment-specific modeling has shrunk, and the prediction methods for the effects of buildings have been improved. Walfisch and Bertoni [11] presented a rigorous theoretical solution for propagation past many infinitely-wide, evenly-spaced rows of buildings characterized by an average height, based on diffraction theory. Saunders and Bonar [12] modified Walfisch and Bertoni's method to account for the individual heights of the buildings nearest the receiver. Ranade [13] developed a computer-based method

for tracing reflected ray paths, assuming geometrical optics (no diffraction), with a database including all of the buildings in the area modeled with simple geometrical shapes, and specified by location and height. Ikegami et al. [14] used a similarly detailed database and computed both multiple diffractions and multiple reflections, basing the diffraction predictions on assumptions that the buildings are both very wide (only roof-top diffraction is considered) and very far apart (diffraction loss due to each edge is simply summed) and that the transmitter is much higher than the buildings (only a single edge of each building diffracts).

2.2 Prediction of Diffraction

The development of the Fresnel-Kirchoff theory of diffraction by a thin, two-dimensional absorbing screen is described in texts on optics such as those by Born and Wolf [15] and Goodman [16]. The application of this theory to the diffraction of radio waves by a mountain peak, modeled as an infinitesimally thin "knife edge" of infinitely wide extent, a practice dating to the 1930's [17], is discussed by Carlson and Waterman [18]. Comparisons with measurements in [18] indicate prediction error when the mountain falls off sharply in the direction transverse to the direction of propagation.

Bachynski and Kingsmill [19] conducted scale-model measurements of diffraction by thin, knife-edge obstacles with various transverse profiles, and saw

a strong dependence on transverse profile. Agreement was shown with predictions based on an application of the Fresnel-Kirchoff theory that included an exact description of the transverse profile, rather than the typical infinitely-wide knife-edge model.

Millington et al. [20] formulated a rigorous solution for knife-edge diffraction by two successive infinitely wide edges as a two-dimensional "Fresnel surface integral" that is computationally efficient but not easily extended to more than two edges. Whitteker [21] and Haslett [22] recently modified Millington's formulation to include the energy reflected from a connecting surface between the two infinite knife edges. They each presented experimental measurements that agree with predictions to within 3 dB over a broad range of diffraction angles.

Deygout [23] and Epstein and Petersen [24] have presented techniques for combining the effects of two or more successive infinite knife edges, where Fresnel-Kirchoff theory is applied to each obstacle taken individually. These are simple, useful approximations derived by comparison with measurements taken on radio paths many tens of kilometers in length, involving diffraction by large terrain peaks; they are not necessarily applicable to the dense urban environment.

The geometric theory of diffraction (GTD) was applied to propagation through hilly terrain by Luebbers [25]. The theories of GTD and its more general form, the uniform theory of diffraction (UTD), describe the scattering from a perfectly conducting wedge, and inherently include both reflected and diffracted energy. Luebbers introduced an approximate correction for the diffraction by finitely conducting wedges, and applied the single obstacle result repeatedly to approximate the effect of multiple peaks. This approach was shown to work well in predicting the effect of large-scale terrain features [25].

In recent years Fresnel-Kirchoff theory has been applied in a rigorous way to multiple knife edges. Vogler [26] produced a general solution in a series form that includes repeated integrals of the error function. Sharples and Mehler [27] extended Vogler's result to treat diffraction by multiple cylinders, as an alternative model of hilly terrain. In [28], Whitteker derived a rigorous method of applying Fresnel-Kirchoff theory to diffraction by hilly terrain. The field is computed as a function of height above each terrain peak, and an integration is performed over this field distribution to find the field above the next peak. In this way any number of peaks can be considered. An efficient method of integrating over the field distribution is presented in [28], using local quadratic phase and amplitude interpolating polynomials, and a single step to infinity. The peaks are modeled as knife edges extending to infinity in the transverse

direction. Reflected energy from the intervening terrain is modeled with a reflecting plane bridging each pair of knife edges, reflecting a portion of the field diffracted from the previous edge and contributing energy to the diffraction field above the next edge. Successful results were obtained when even smoothly varying terrain was modeled by piecewise linear reflecting planes with a knife edge at either end.

The published literature does not address the application of Fresnel-Kirchoff diffraction theory in its three-dimensional form to the problem of building shadowing. With the possible exception of the GTD method, there are no published approaches that consider diffraction by multiple edges of both horizontal and vertical orientation. The GTD method, however, is not designed for application to multiple obstacles that are near each other. The dense urban environment mandates the use of rigorous methods for considering multiple obstacles. In this work, we combine ideas of several authors in extending rigorous multiple diffraction methods to consider transverse variations. We then develop appropriate building models and apply the diffraction methods to an arbitrary building database.

III. USE OF A GIS BUILDING DATABASE

The propagation models described here require a digital description of the heights and locations of all of the buildings in the service area. A more comprehensive prediction tool, one that includes street reflections and obstructions by terrain undulations, will also need terrain elevation information. In anticipation of this, we have investigated the use of computer software packages known as Geographic Information Systems as an aid to database management and analysis. Geographic Information Systems (GIS) are sets of software utilities that facilitate the storage and manipulation of digital geographic information of any type. The specific GIS package we are using is the Geographic Resources Analysis Support System (GRASS), which runs on a Sun workstation.

3.1 GRASS Philosophy

GRASS was developed by the U.S. Army Corps of Engineers with the help of private contractors. It is a C language program in the public domain. Users are provided with the source code and encouraged to write new functions for their own applications and then to pass them on to the rest of the GRASS community through a clearinghouse. In this way, the program is always growing.

GRASS stores, displays, and analyzes data in two different forms: cells and vectors. A cell is a portion of a grid, and contains a single attribute in the form of an integer. This integer may signify elevation in meters or it may map to a type of vegetation cover, for example. The meaning of the cell value is stored in an attribute file and can be called up and displayed on the screen. Each grid of cells, containing information of a certain type, is known as a cell layer. There are no inherent limits on the size of cells, but the maximum lateral resolution available for elevation data on a broad geographic scale is 30m; that is, each cell in the elevation layer is 30m on a side and has a single elevation value attached to it. The elevation data are available from the U.S. Geological Survey (U.S.G.S.) in the form of digital elevation models and are easily input to the GRASS database through existing routines.

The other basic form of data representation in GRASS is the vector. This is a line defined by its endpoints. Curved lines are represented with piecewise linear sections. Vector information can be easily overlaid on a cell map for display. An example application is an overlay of roads (stored as vectors) on a cell map showing by use of different colors which cells have a line of sight to a particular transmitter location. With the roads overlay it is possible to see where a mobile on the road will have a line of sight. (Roads and streets locations for many areas are available in digital form from the U.S.G.S., as well as other sources.) A routine to compute the line of sight based on the terrain is supplied with the GRASS package.

3.2 Storage of Building Information in GRASS

Vectors can be used to define an area, which can then be given an attribute. This is similar to a cell except that when vectors are used there is not the resolution limit set by the grid pattern, as the boundaries of the vector area are defined by the endpoints, whose precision is only limited by the original information source. Therefore, we are representing buildings in GRASS with vector areas, with the attribute representing the building height in meters.

Input of building information can be accomplished in several ways. If the information is already stored in a different database, conversion to the GRASS storage format should be possible. GRASS stores the vector information in files composed of both ASCII and binary representations, and utilities are provided for conversion from one to the other. The format is very simple, consisting of a table of endpoints of the lines that make up each area, and the value of the attribute for the area. If the building information is in paper map format, the data can be manually entered with a digitizer. There are digitizer support routines within GRASS that automatically create the necessary files.

The building information stored in the GRASS ASCII vector files can be easily read in to the diffraction prediction routines and contain all of the information necessary to model the buildings: the coordinates of the four corners,

and the height of the roof.

3.3 Identification of Diffracting Buildings

The diffraction computation procedures detailed in this thesis require only the coordinates of the nearest building obstacle to an observation point. Any other diffracting buildings are considered one at a time by placing new observation points on the far edges of the nearest building and repeating the process. This recursive procedure, described in detail in Chapter 5, has the benefit of allowing for a straightforward interaction with the database.

The coordinate system of the database is first translated such that the horizontal coordinates of the transmitter are (0,0). For each new observation point, a new coordinate system is defined by rotating the axes such that the observation point lies on one of the axes. In the software implementation of the models in this thesis (detailed in Chapter 7 and included in full in the appendix), the original coordinate system, translated to the transmitter point, is the (z,x,y) system, where y is the vertical axis (which is not affected in these manipulations) and (z,x) describes a horizontal plane. The new coordinate system is specified in terms of (u,v,y) where (u,v) is a horizontal plane defined as a rotation of the (z,x) plane (see Figure 3.1) such that the u coordinate of the observation point, P, is the horizontal component of the distance to the point, computed by

$$u_p = \sqrt{z_p{}^2 + x_p{}^2} \tag{3.1}$$

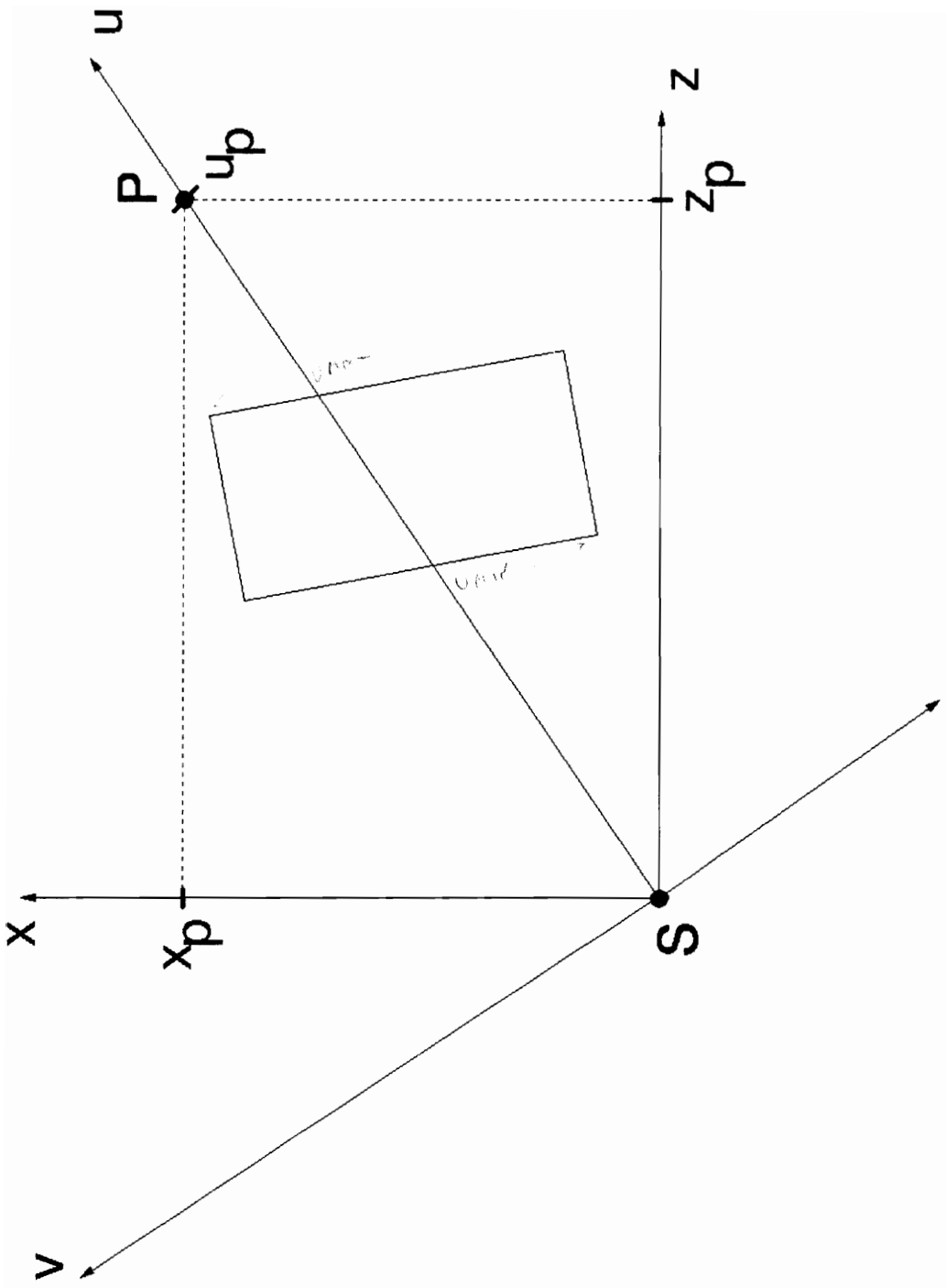


Figure 3.1. Coordinate System Transformation

where (z_p, x_p) are coordinates of P. By definition, then, the v coordinate of P is zero. All of the buildings in the database are then rotated to this new set of axes according to

$$u = z(z_p/u_p) + x(x_p/u_p), \tag{3.2}$$

$$v = -z(x_p/u_p) + x(z_p/u_p). \tag{3.3}$$

The transformations are performed without costly calls to trigonometric functions and are performed very quickly. If the building database is very large and the coordinate transformations begin to take a large amount of computer time, the database should be reduced by defining a window about the area of interest and deleting buildings outside the window, before input to the prediction program.

After the (u,v,y) coordinates of the buildings are found, each building is bounded by its maximum and minimum values of u and v. A building is then flagged for further consideration with OBSTRUCT = TRUE if all of the following are true:

$u_{max} > 0$,

$u_{min} < u_p$,

$v_{max} >$ -MAXDIST,

and

$$v_{min} < \text{MAXDIST},$$

where MAXDIST is set large enough ($\sim 10\lambda$) such that if it is remotely possible for the building to obstruct the signal significantly, the building will be flagged. The flagged buildings are then searched to find the nearest one by checking the values of u_{max}. All of the manipulations so far, which are performed on many buildings, require minimal computer time. With a building identified that obstructs the horizontal projection of the signal to some degree, the vertical component of the signal path, the building height, and Fresnel zone clearances are all considered in the final determination. If the building does not meet the criteria (discussed in Chapter 7), it is marked with a flag to remove it from consideration, and the group of buildings flagged with OBSTRUCT are searched again for the next nearest building.

IV. SINGLE DIFFRACTION MODEL

4.0 Introduction

In a general urban situation, electromagnetic waves will bend, or diffract, past vertical and horizontal building edges. A general diffraction model for buildings, therefore, must include both of these mechanisms, as the model proposed here does. This will be referred to as three-dimensional diffraction, where the obstacle face is assumed to be two-dimensional (planar), and the direction of propagation constitutes the third dimension. With this model, diffraction field contributions due to roof and side edges are computed individually and summed at the receiver.

The assumption of single diffraction limits the application of this model to situations where the field in the vicinity of each diffracting edge has not experienced any significant diffraction due to previous edges and does not suffer any further diffraction before reaching the receiver, so that each field component is diffracted by only one edge. The techniques presented here for computing single diffraction constitute the building blocks of a more general multiple diffraction model described in Chapter 5. An example of a single diffraction situation is given in Figure 4.1, where the transmitter is higher than both the obstacle building and the receiver, and diffraction is introduced only by the left vertical corner and the two roof edges nearest the receiver. The right corner is

Figure 4.1. Typical Single Diffraction Situation

too distant relative to the other edges to contribute relatively significant energy, and the roof edges away from the receiver do not obstruct the path.

4.1 Fresnel-Kirchoff Diffraction Theory

The amplitude and phase of a wave, u, diffracting through an aperture in a thin, two-dimensional obstruction, measured at point P on the other side, is given by the Kirchoff diffraction integral [15],

$$u(P) = \frac{-i}{2\lambda} \int\int_R \frac{1}{r_s\, r_p} [\cos(\alpha) - \cos(\beta)]\, e^{ik(r_s + r_p)}\, dR\,, \tag{4.1}$$

where the region of integration, R, is the surface of the aperture, point Q is a secondary Huygens source in the aperture, and r_s and r_p are the distances from point Q to the source and observation points, S and P, respectively. This geometry is presented in Figure 4.2. The surface normal, $\vec{n}$, pointing towards the half-plane containing the source point, makes an angle α with the vector $\vec{PQ}$, and an angle β with the vector $\vec{SQ}$. The wavelength is represented by λ and the propagation constant, k, is equal to $2\pi/\lambda$. Equation (4.1) is normalized to unit amplitude at the source.

The Kirchoff integral provides a scalar description of any wave: acoustic, water, etc. Since we are interested in electromagnetic waves, we will refer to the electromagnetic field, E(P), instead of u(P). Specifically, we are concerned with

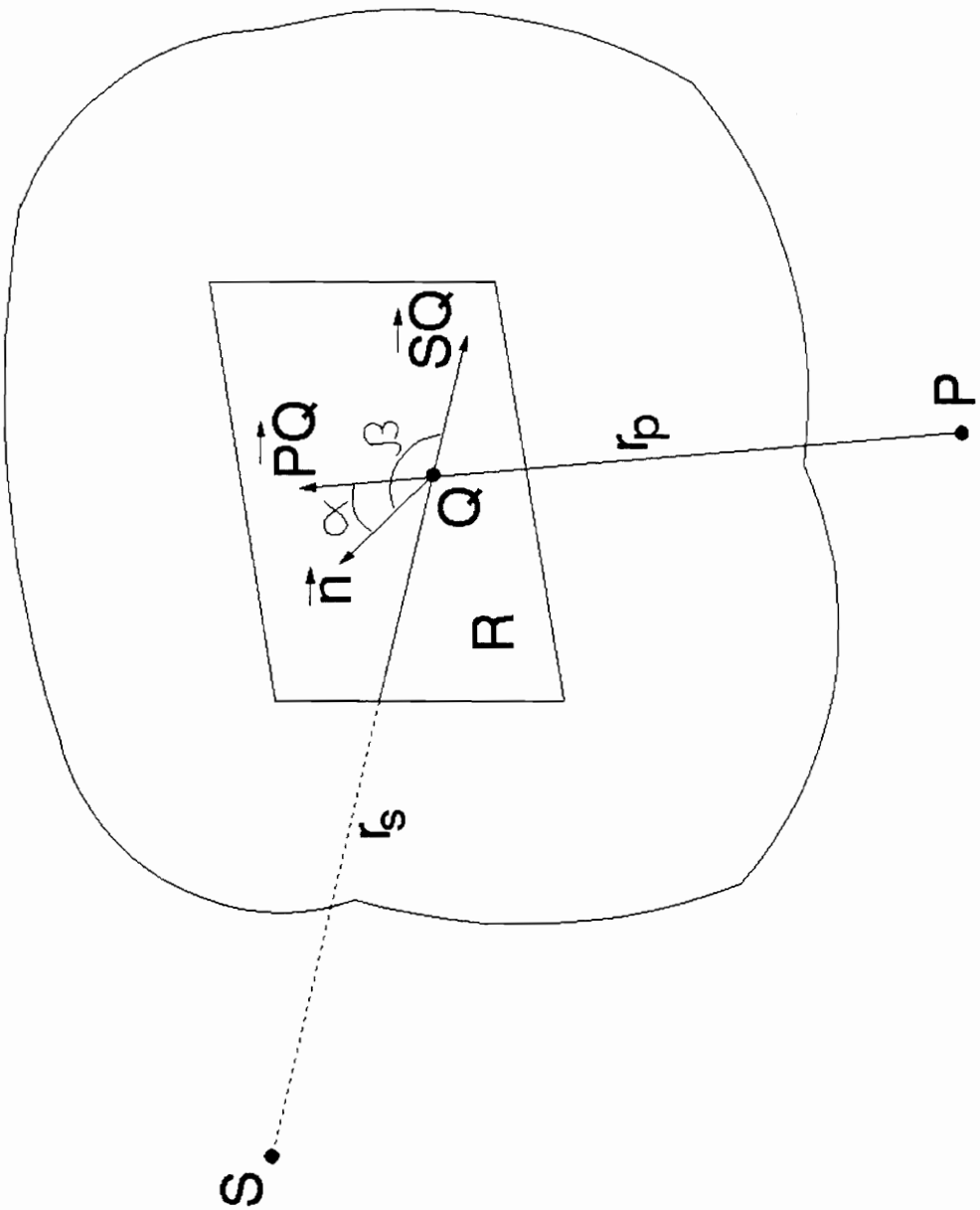


Figure 4.2. Diffraction through aperture in an infinite obstacle plane

polarized electromagnetic waves. By applying a scalar solution to a vector phenomenon we simplify the problem but lose the ability to model any dependence on polarization. Experiments on large-scale terrain obstacles generally show little impact of polarization on diffractive shadowing [29], but polarization effects are known to increase as the distances (relative to wavelength) to the diffracting edge decrease [30]. Scale-model measurements of polarization dependence using aluminum sheets by Neugebauer and Bachynski [30] show that diffraction by a conducting knife edge exhibits a maximum 0.4 dB difference between the results for vertical and horizontal polarizations at a distance of 50 wavelengths, and a maximum 2 dB difference at a distance of 20 wavelengths, for situations involving up to 30 dB of diffraction loss. At 900 MHz, a typical mobile radio frequency, 20 wavelengths is equal to 6.7 m or 22 ft, a reasonable distance to assume between obstacles and termination points in an outdoor radio system. Thus, for up to 30 dB diffraction loss, we can expect a maximum error due to polarization effects of 2 dB.

The Fresnel-Kirchoff approach requires the assumption of perfectly absorbing material in the obstacle edges. With real buildings that have some finite conductivity, there will be some reradiation, or scattering, from the edges. This source of diffuse radiation may be important but is neglected with this approach. It should be included, as necessary, along with other sources of scattered and reflected energy, in a more comprehensive model of urban propagation.

When Fresnel-Kirchoff diffraction theory is applied to microwave radio propagation, typically the aperture is semi-infinite, extending up into space, unlike the finite aperture of Figure 4.2. When the diffracting obstacle, such as a building or a hill, is modeled as extending to infinity in the transverse dimension, we have the simple case of a one-dimensional diffracting edge and two-dimensional single diffraction. The geometry for this special case, known as infinite knife-edge diffraction, is shown in Figure 4.3. The line SP is drawn connecting the source and observation points. The xy plane is constructed to include the plane of the obstacle; and the origin, O, is defined as the point of intersection with SP. The region of integration, R, is the aperture, or the unobstructed area of the xy plane, which in this case is the semi-infinite plane above the obstacle edge. R is bounded by y_1 as indicated on the graph, and by $y_2=\infty$, $x_1=-\infty$, and $x_2=\infty$. The z-axis is anti-parallel to the surface normal, n, and the angles α and β seen in Figure 4.2 are presented in Figure 4.3 in a different way, though they are the same angles. The lengths s, p, and q, are the distances from the origin to points S, P, and Q. The angle between the z axis and the line SP is denoted δ, and the angle between the x axis and SP is denoted θ.

A central concept in Fresnel-Kirchoff diffraction theory is that of Fresnel zones (see optics texts such as [15] or [16]). A Fresnel zone is an annular region

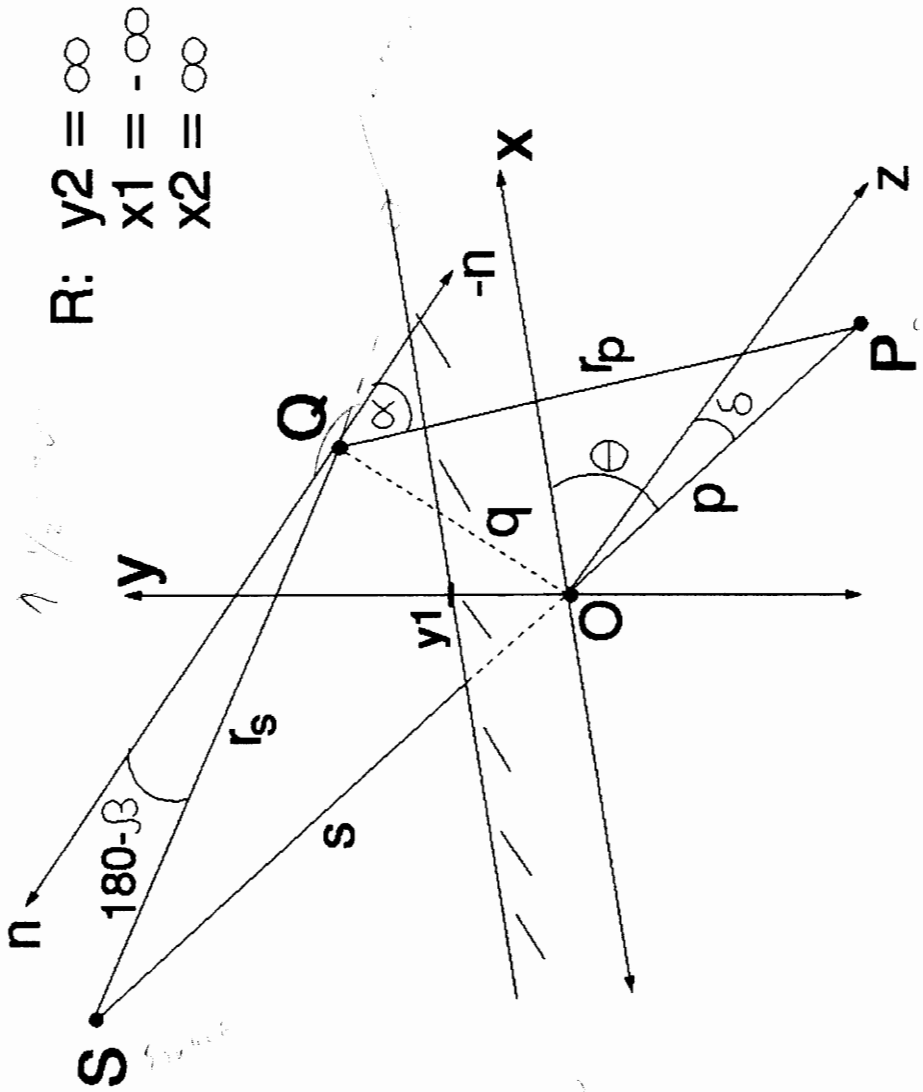


Figure 4.3. Infinite Knife Edge

of a plane transverse to SP defined such that all of the energy passing through it interferes constructively at the observation point. The first two Fresnel zones are illustrated in Figure 4.4, where the outer boundary of the n^{th} zone is the radius, R_n, where

$$R_n = \sqrt{\frac{n\lambda sp}{(s + p)}}. \tag{4.2}$$

The distance from the source to a zone boundary to the observation point is exactly $\lambda/2$ greater than that for the previous zone boundary. Although neighboring Fresnel zones contribute energy exactly out of phase with each other, they don't completely cancel out because the energy contribution decreases with increasing distance from the origin. This model of wave propagation includes the notion of Huygens sources of secondary spherical radiation to explain the change of direction of energy.

When a portion of the radio path is obstructed, as in Figure 4.4, and some zones are blocked, the theory predicts diffracted, or bent, energy arriving at the observation point by way of the unobstructed Fresnel zones, at a lower power level because higher order zones contribute less energy. When the line of sight is clear, but some zones are still obstructed, both enhancement and degradation of the signal are possible, depending on the relative obstruction of zones whose contributions interfere constructively and destructively with that of the first, dominant, Fresnel zone.

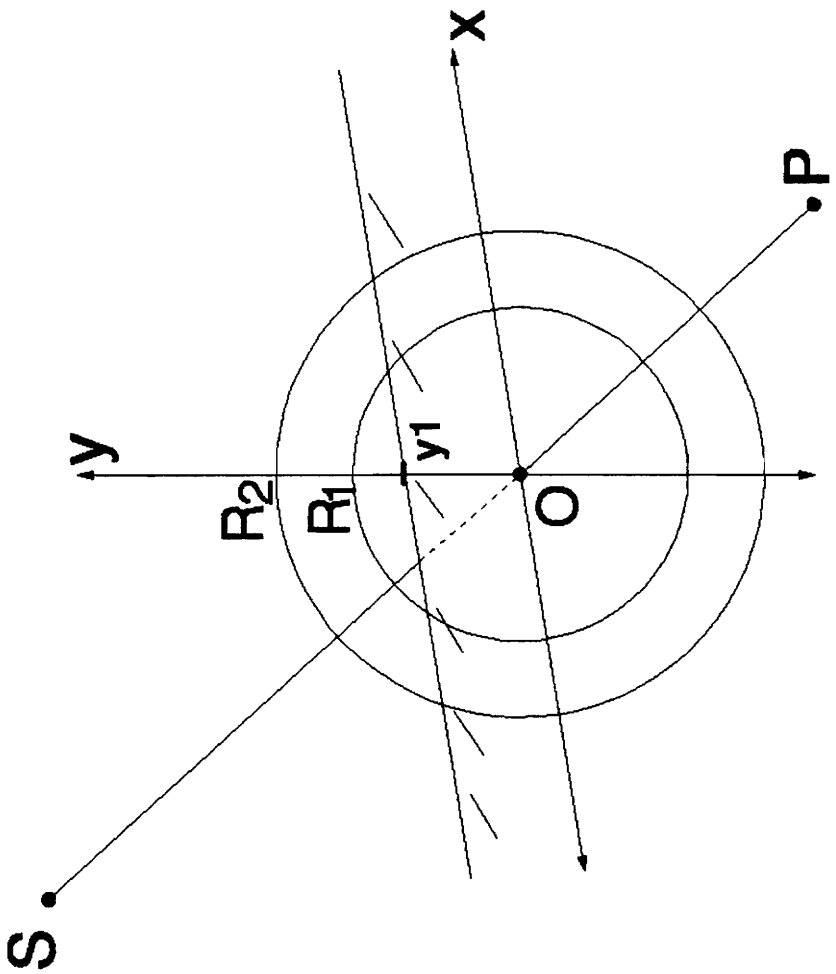


Figure 4.4. Fresnel Zones

4.2 Simplification of Diffraction Integral

The integral (4.1) does not have an analytic solution, and as it is a double integral, it is impractical to evaluate numerically. There are a standard set of approximations made in order to separate the integrand into independent functions of x and y, which allows the evaluation of two single integrals; see for example Born and Wolf [15], Goodman [16], or Whitteker [28]. The procedure is repeated here to ensure an understanding of the assumptions before attempting to extend the theory to new applications.

4.2.1 Small Angle Approximation

Simplification of (4.1) requires the approximation that the angles α and (180-β) in Figure 4.3 are always small compared with unity. Two assumptions are inherent in the small angle approximation. The first is that the distance, q, from the origin to each point, Q, in the unobstructed area is much smaller than the distances s and p. This assumption clearly never holds in the case of semi-infinite integration regions, as in Figure 4.3, where Q may be at infinity. However, the approximation can still be made with good results if s and p are much larger than both the wavelength and the distance from the origin to the nearest edge (y_1 in Figure 4.3) [15], [28] .

The significance of the wavelength arises because, although the entire

region R must be considered in the integration, the dominant portion of R is that nearest the edge, where most of the energy passes. The size of the dominant portion is measured in Fresnel zones. In general, the majority of the energy at P originates from secondary sources, Q, within the first few unobstructed Fresnel zones. Thus the distances, q, to the most dominant sources are less than some R_n, defined in (4.2). The constraint on q with respect to s and p therefore constrains R_n with respect to s and p. Since R_n depends on λ, s, and p, the constraint on R_n with respect to s and p in turn constrains the wavelength with respect to s and p. Thus, transmission at a wavelength much smaller than s and p ensures that q for the most dominant sources Q will be small compared with s and p, provided that the nearest distance to the edge, y_1, is also small compared to s and p.

The second assumption necessary in the small angle approximation is that the angle δ in Figure 4.3 is always small compared with unity, meaning that the building face is always nearly perpendicular to SP. A large δ would always invalidate the small angle approximation. The second assumption obviously does not always hold in reality, where a diffracting building face may be oriented at any angle with SP. We will address this in later sections.

The preferred method of validation of this approximation and those in the following sections would be comparison of the results before and after making the approximations, but we have not yet been able to evaluate the double integral in

(4.1) and so can only compute results with the approximations.

4.2.2 Fresnel Phase Approximation

The phase term of the integrand in (4.1), $e^{ik(r_s+r_p)}$, can be separated into a product of independent functions of x and y with the Fresnel phase approximation [16], [15]; where x and y define the obstacle plane, as in Figure 4.3. With the small angle approximation, expressions for r_s and r_p are given by

$$r_s = \sqrt{s^2 + x^2 + y^2} \tag{4.3}$$

$$r_p = \sqrt{p^2 + x^2 + y^2} \tag{4.4}$$

or

$$r_s = s\sqrt{1 + \left(\frac{x}{s}\right)^2 + \left(\frac{y}{s}\right)^2} \tag{4.5}$$

$$r_p = p\sqrt{1 + \left(\frac{x}{p}\right)^2 + \left(\frac{y}{p}\right)^2}\,. \tag{4.6}$$

Approximating (4.5) and (4.6) with the first two terms of the binomial expansion leaves

$$r_s \simeq s + \frac{x^2}{2\,s} + \frac{y^2}{2\,s} \tag{4.7}$$

$$r_p \simeq p + \frac{x^2}{2\,p} + \frac{y^2}{2\,p}\,. \tag{4.8}$$

After arranging terms we may write

$$e^{ik(r_s+r_p)} \simeq e^{ik(s+p)}\, e^{ikx^2(\frac{1}{2s}+\frac{1}{2p})}\, e^{iky^2(\frac{1}{2s}+\frac{1}{2p})}, \tag{4.9}$$

which is a product of independent functions of x and y, and a constant term.

4.2.3 Obliquity Factor

The bracketed term in (4.1), [cos(α)-cos(β)], is known as the obliquity factor [15] because it takes into account oblique incidence of the wave on the obstacle plane. The small angle approximation allows the obliquity factor to be approximated as 2sinθ, where θ is the angle that the building face makes with SP in Figure 4.3. Since the factor is now independent of x and y, it can be moved outside of the integral.

4.2.4 Amplitude Approximation

The integral, (4.1), is less sensitive to errors in amplitude than errors in phase, so while the phase was approximated with a quadratic function of x and y, a coarser amplitude approximation will be made. The amplitude factor, $\frac{1}{r_s r_p}$, is approximated simply as a constant term, $\frac{1}{sp}$, and moved outside the integral [15], [28].

4.2.5 Simplified Kirchoff Diffraction Integral

With the approximations described in Sections 4.2.2 - 4.2.4, the integral (4.1) can be written

$$E(P) = \frac{-i}{\lambda} \frac{e^{ik(s+p)}}{sp} \sin\theta \int_{x_1}^{x_2} e^{ik\frac{x^2(s+p)}{2sp}} dx \int_{y_1}^{y_2} e^{ik\frac{y^2(s+p)}{2sp}} dy, \qquad (4.10)$$

where (x_1,x_2) and (y_1,y_2) bound the region R in Figure 4.3. After a change of variables, (4.10) becomes

$$E(P) = \frac{-i}{2} E_{fs} \sin\theta \int_{\xi_1}^{\xi_2} e^{i\pi\xi^2/2} d\xi \int_{\eta_1}^{\eta_2} e^{i\pi\eta^2/2} d\eta \qquad (4.11)$$

where

$$E_{fs} = \frac{e^{ik(s+p)}}{(s+p)} \; ; \qquad (4.11a)$$

$$\xi_1 = x_1 \sqrt{\frac{2(s+p)}{\lambda sp}} \; ; \qquad \xi_2 = x_2 \sqrt{\frac{2(s+p)}{\lambda sp}} \; ; \qquad (4.11b,c)$$

$$\eta_1 = y_1 \sqrt{\frac{2(s+p)}{\lambda sp}} \; ; \qquad \eta_2 = y_2 \sqrt{\frac{2(s+p)}{\lambda sp}} \; . \qquad (4.11d,e)$$

In Eqn. (4.11), E_{fs} is the free-space field at P (with no diffraction); the new variables ξ and η are known as the diffraction parameters, which are dimensionless variables of integration. These parameters are used in order to simplify the numerical evaluation, but the bounds on the geometrical region R (the aperture) over which the integral is computed, will be defined as limits on x and y. These limits transform into limits on the diffraction parameters through (4.11b,c,d,e). Equation (4.11) contains the product of two single integrals in the Fresnel form, which allows for greatly simplified evaluation as compared with

(4.1). The methods of evaluation are discussed in Chapter 6.

The infinite knife edge case of Figure 4.3 is used as an example, where x_1=-∞, x_2=∞, y_2=∞, and y_1 is the height of the edge relative to the origin, O. The origin is always the point at which the line SP (the line of sight) intersects the xy plane., With a wavelength of 0.33m and source and observation points situated such that s is 500m, p is 50m, and θ is 90°, the height of the edge relative to the origin is increased from 20m below the line of sight (y_1= -20m) to 20m above the line of sight (y_1 = 20m). The results of applying (4.11) are plotted in Figure 4.5, where the abscissa is the power relative to free space, which explicitly shows the impact of diffraction. The oscillations at low y_1 are due to self interference in the wave. At y_1=0, the shadow boundary, the relative power is -6 dB, because at that point exactly half of the electromagnetic field is prevented from passing the obstacle. In the limit, as the edge drops far from the line of sight and y_1 becomes a large negative number, the computed power converges to the free space value as expected.

4.3 Three Dimensional Single Diffraction Model

The method of Section 4.2.5 for computing diffraction by a single edge can be modified to apply to the case of a real building with several diffracting edges, as in Figure 4.1. Assuming the wave in the vicinity of each edge is not diffracted by any of the other edges (the single diffraction assumption), the total

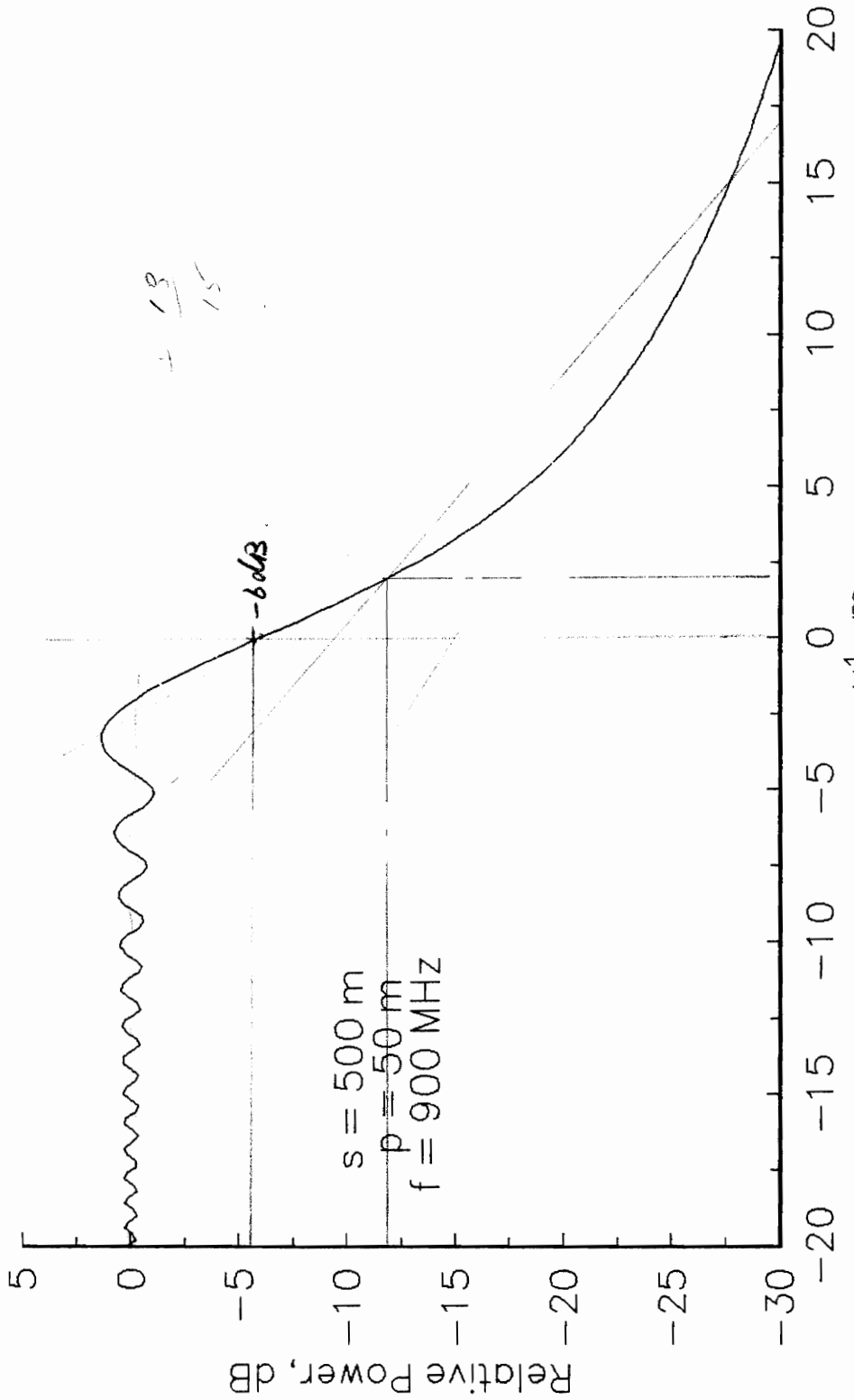


Figure 4.5. Diffraction by Infinite Knife Edge

field at P is found as the sum of the field contributions due to each edge by

$$E(P) = \sum_{n} E_n(P) , \tag{4.12}$$

where each $E_n(P)$ is found according to (4.11), by integrating over surfaces that are subsets of the total area of integration. The procedure for defining the integration region for each $E_n(P)$ in terms of x and y is described below.

4.3.1 Representation of Real Buildings

In order to simplify the geometrical manipulations, we have assumed that all building faces are vertical and that all roof edges are horizontal. Also, we assume the vertical angle that the line SP in Figure 4.3 makes with the horizontal plane is small compared to unity. This requires that the horizontal distance between the transmitter and receiver is greater than the vertical distance.

The second assumption required in the small angle approximation, discussed in Section 4.2.1, is that δ, the angle between the z axis and the line SP in Figure 4.3, is small compared with unity. We have already assumed that the vertical component of δ is small, so we will neglect it and consider δ as the angle in the zx plane between the projection of SP and the z axis. Now, θ, which we have defined (Section 4.1) as the angle between SP and the x axis, also resides in the zx plane, and is equal to (δ +90°).

Our first attempt at modeling building faces at oblique angles to the propagation vector was a direct application of (4.10) and (4.11), where the xy plane necessary for evaluation of each $E_n(P)$ was constructed to contain the corresponding building face, which may be oriented at any angle δ. However, the approximate form for the obliquity factor included in (4.11) (see Section 4.2.3) is intended only for small values of the angle δ. As δ increases, the obliquity factor has an increasingly large effect and, as it was not meant to be used at large angles, there is no reason to believe the large impact is legitimate.

The impact of the obliquity factor is illustrated in Figure 4.6 in the limiting case where the diffracting obstacle has receded far from the line-of sight (y_1 goes to -∞). The ordinate of Figure 4.6 is θ (see Figure 4.3). Equation (4.11) should predict the free space value in this case (as in Figure 4.5 at small y_1) but Figure 4.6 shows that the obliquity factor introduces a 6 dB error for the case of a 150° angle between SP and the obstacle face, and the loss increases steeply as the angle increases further towards 180°.

The obliquity factor introduces a very small correction (< 1 dB) at the small values of δ (< 30 °) for which it was intended, as seen in Figure 4.6. Therefore, to remove the large incorrect effect of the obliquity factor at large angles from normal, we remove the obliquity factor from the diffraction

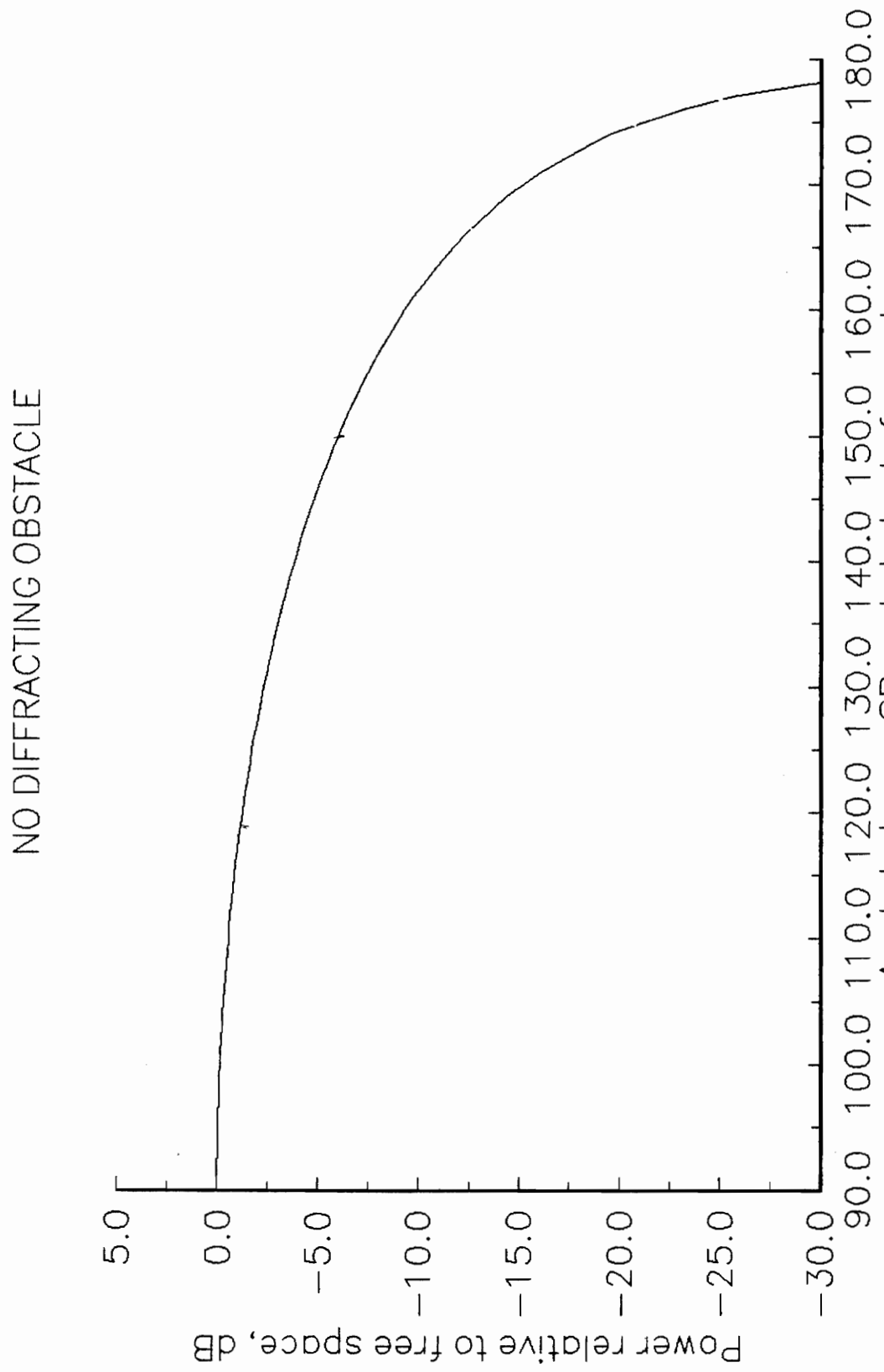


Figure 4.6. Effect of Obliquity Factor

expression and thereby sacrifice the small, approximately correct effect at small angles. The obliquity factor is neglected by approximating $2\sin\theta$ simply as 2.

Removal of the obliquity factor is equivalent to making the assumption that diffracting obstacles are perpendicular to SP, so we now use the projection of the obstacle face on a plane normal to SP to obtain the region of integration.

4.3.2 Roof-Edge Diffraction Fields

With the above modification, the xy plane, or aperture plane, no longer contains the building face. We have already made the assumption that SP has a small vertical angle, so we now can neglect it, and instead of defining the xy plane as a plane normal to SP, we take it as a vertical plane, oriented such that the y axis is vertical, and the x axis is horizontal and perpendicular to the projection of SP on the horizontal plane. An example is provided in Figure 4.7, where a single roof edge is shown from above. The roof edge is indicated with a bold line that corresponds to the bold line on the building shown in full perspective view in the inset. In the top view, only the x axis of the aperture plane is visible, as the y axis is directed out of the page. The inset figure includes the xy plane superimposed.

The inset of Figure 4.7 duplicates the building of Figure 4.1, an example of a building obstacle which introduces single diffraction from several edges, as

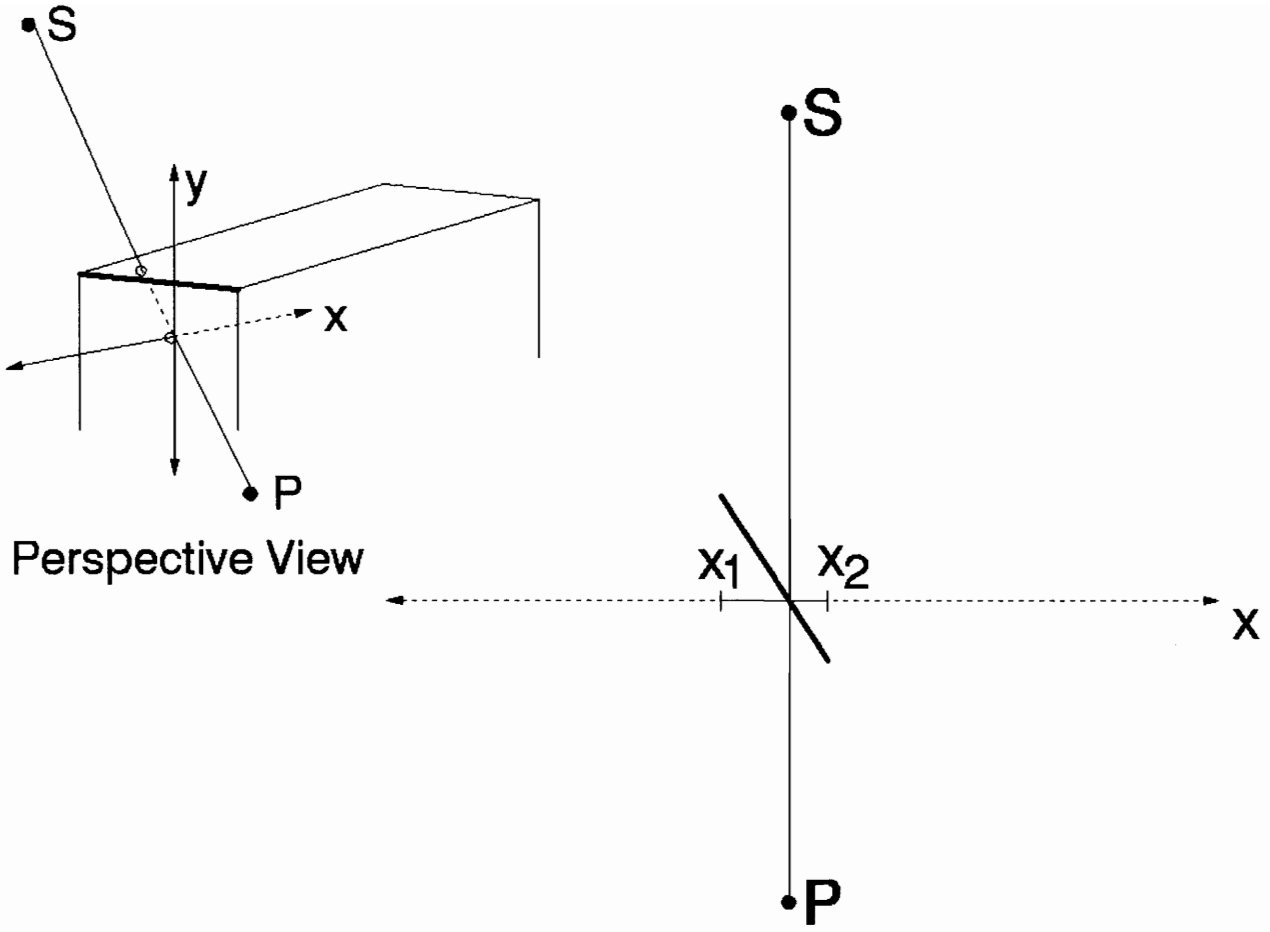


Figure 4.7. Top View of Left Side Roof Edge and Construction of Aperture Plane

discussed in Section 4.1. Consider first the visible building face on the left, through which SP passes, and the field diffracting over the roof. The xy plane is oriented according to the rules above, and located such that the origin is at the point of intersection of the building face with SP, as in Section 4.1. In order to find the integration limits, the building face is projected onto the xy plane, yielding the bounds on integration region R in terms of x and y: (x_1, x_2), and (y_1, y_2). The solution for x_1 and x_2 is shown in Figure 4.7, y_1 is found as in the infinite knife edge case of Section 4.1, and $y_2 = \infty$. These distances transform into the limits of integration according to (4.11).

Now consider the second diffracting building face visible in Figure 4.1, the front of the building. This face, seen as a bold line in Figure 4.8, does not intersect SP, though it is near enough to SP to contribute diffracted field to P. The xy plane on which the building face is to be projected is defined as the vertical plane that intersects the building face at the point nearest to SP, and is oriented, as above, such that the x axis is perpendicular to the horizontal projection of SP, and the origin is located where the plane intersects SP. This is illustrated in Figure 4.8, with the bounds on x_1 and x_2 indicated, and a perspective view included as an inset.

4.3.3 Ground Clearance

As a first approximation, the ground is taken to be at negative infinity for

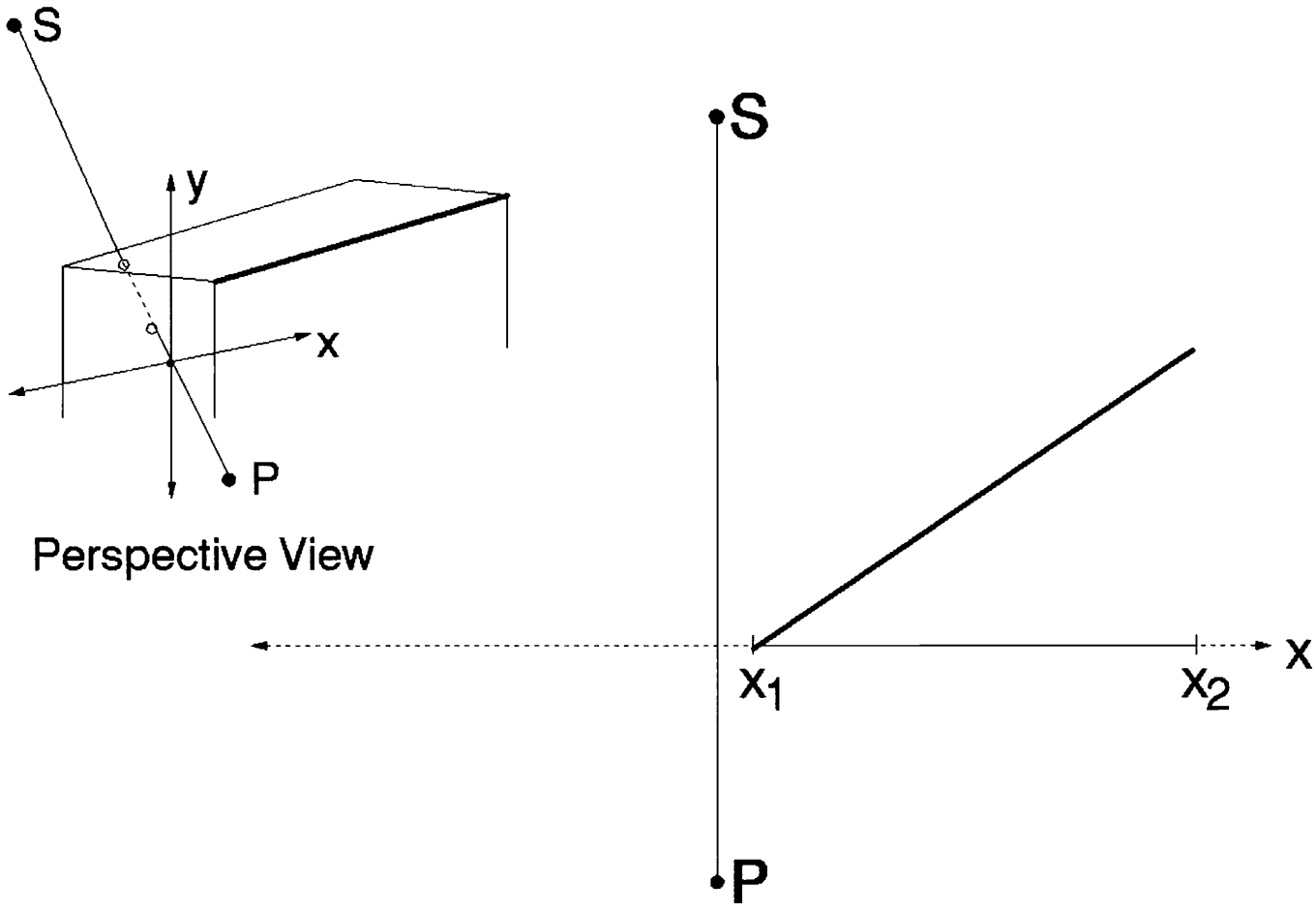


Figure 4.8. Top View of Front Roof Edge and Construction of Aperture Plane

purposes of computing diffraction. If the ground clearance (the clearance over the terrain, or in our case, the streets and sidewalks) is at least 0.55 Fresnel zones, a criterion traditionally used in radio engineering [31], this approximation incurs no significant error (< 2 dB). Now we must look at the validity of the ground clearance assumption. These modeling techniques are designed with small portions of urban or semi-urban areas in mind. We will make the assumption that the signal coverage is small enough, or the ground is flat enough, that no significant terrain variations occur over the course of the radio path. Now if we assume that the transmitting antenna will be at least 5 m above ground, and receiving antennas at least 2 m high, the slanted path will be nearest the ground when it is near the receiver. The radius of the Fresnel zone ellipsoid [31], computed by (4.2), is maximum when s and p are equal. For λ = 0.33 m, and a short link of 100 m, the maximum value of $0.55R_1$ is 1.5 m, providing sufficient clearance all along the path. For a longer path of 400 m, the maximum value of $0.55R_1$ is 3.2 m at s=p=200 m, at which point the ray is 2 + (5-2)/2 = 3.5 m above ground, and we have sufficient clearance at the center of the path. At a point near the receiver, where the ray is only 2 m high, such as at s=395 m and p=5 m, we compute $0.55R_1$ = 0.70 m, again giving sufficient clearance. Thus, for the above assumptions: flat terrain, transmitter height at least 5 m, receiver height at least 2 m, and path less than 400 m, we will have ground clearance of 55% of the first Fresnel zone at all points along the path.

A useful extension to this work would be to include methods to input

terrain data, identify the obstructing terrain peaks, and model them as knife edges. The effect of a ground reflection could also be included where deemed necessary. The method of combining diffractions and reflections using image theory is discussed in Chapter 8.

4.3.4 Corner Diffraction Fields

Refer again to the situation in Figure 4.1, reproduced in the inset of Figure 4.9, but this time consider the diffraction field bending around the left rear corner of the building. The corner is drawn in top view in Figure 4.9. The xy plane is defined as the plane normal to SP that intersects the corner. This is presented in the figure as a dashed line. The corner diffraction field is computed by integrating over the region of the xy plane abutting this corner edge. This is the region through which the wave diffracted by the corner (the vertical edge) will propagate. In this case, $x_1 = -\infty$, x_2 is identified in the figure, $y_1 = -\infty$ (where the ground is neglected), and $y_2 = \infty$.

4.3.5 Overall Single Diffraction Building Model

The three diffraction field contributions discussed above are treated as though they diffract through separate apertures in the same plane. This is the requirement of the single diffraction assumption. The regions of integration that yield the three diffraction field contributions are presented together in Figure 4.10, where the apertures are marked by bold lines. Since these regions are not apertures in the same plane, the resultant fields are not strictly the result of

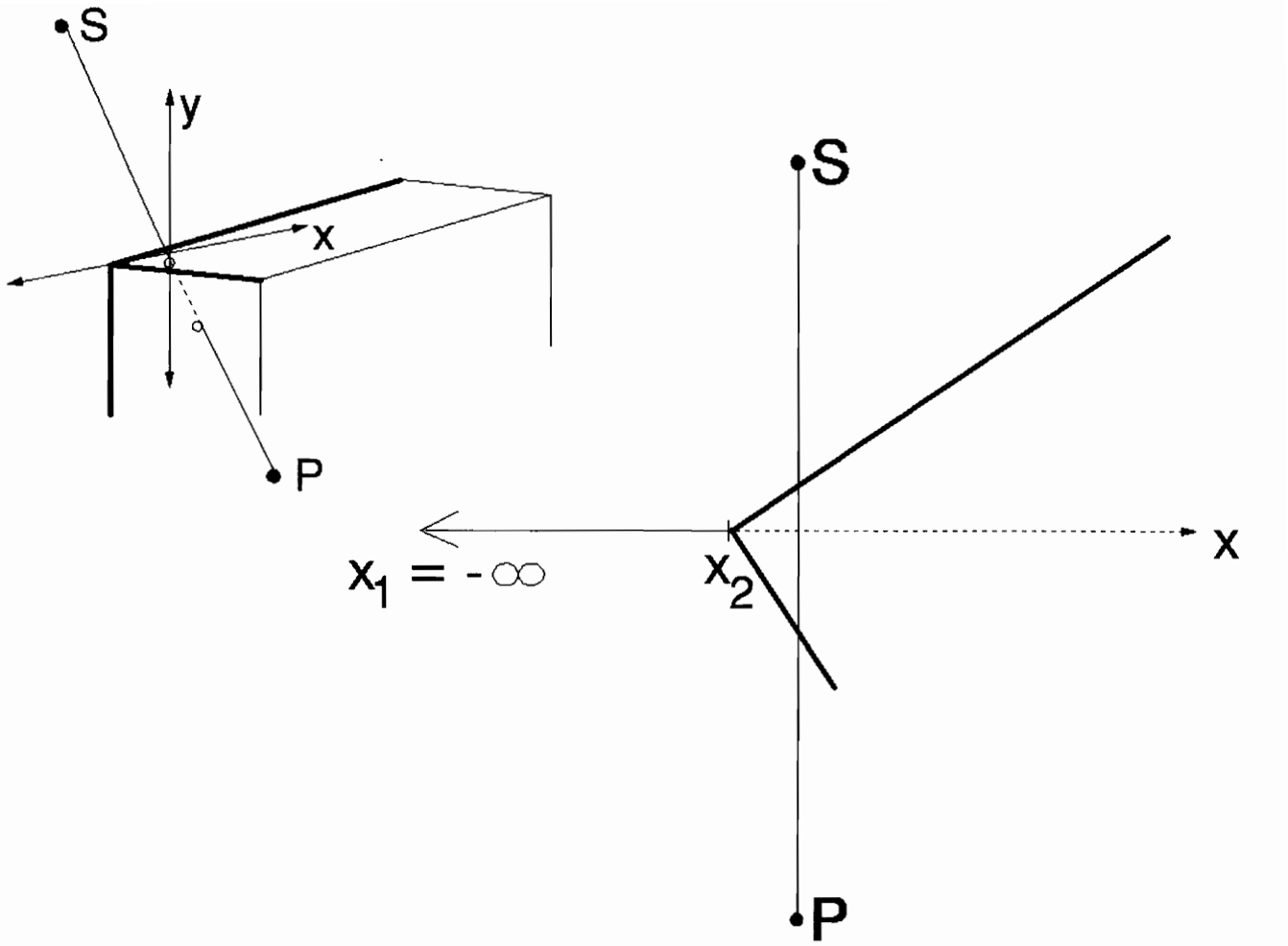


Figure 4.9. Top View of Left Rear Building Corner and Construction of Aperture Screen

Figure 4.10. Single-Diffraction Building Model in Top View

single diffraction. As Fresnel-Kirchoff theory requires the aperture plane to be approximately transverse to the direction of propagation, and it seems most accurate to define the aperture introduced by each building edge according to the point on that edge nearest the direct line SP, we are left with the situation in Figure 4.10 where segments of a total aperture plane are separated into vertical, non-overlapping segments on staggered xy planes whose origins are translated along SP. As evidenced by the inset of Figure 4.10, this translation has both vertical and horizontal components. When viewed along a line of sight from P to S, we see that these aperture plane segments form a single aperture plane transverse to the horizontal component of the direction of propagation. Only when the apertures introduced by different building edges overlap when viewed in this way do we treat it as a multiple diffraction situation.

4.3.6 Interaction with the Database

The building information is stored in a database containing the three-dimensional coordinates of each corner. As introduced in Section 3.3, the software implementation of these models resolves the building database into a local coordinate system (u,v,y), where (u,v) defines a horizontal plane such that the coordinates of the transmitter and observation points are (0,0) and (u_p,0), respectively. The v coordinates of the corners defining each edge provide the set of values for (x_1, x_2) needed in the diffraction model, the distances s and p are easily determined from the u coordinates, and the y values are found through a simple application of the principle of similar triangles.

4.3.7 Example Calculations

The methods described in this chapter are further illustrated here by simulating the movement of a receiver past the building examined in the previous sections. The top and side views, provided in Figure 4.11, show the geometry of the situation. At each computation point along the mobile receiver track, and for each edge that contributes diffraction field, the xy plane is constructed and the limits on x and y are determined. These limits are plotted as a function of distance along the track in Figures 4.12, 4.13, and 4.14 for the left rear corner, the left roof edge, and the front roof edge, respectively, as viewed from the receiver. Figure 4.15 presents the results of applying the diffraction integral (4.11) to each edge, showing that there are regions where each component is dominant, and demonstrating the necessity of computing each component. While the error in the power estimation would usually be less than 3 dB if only the most dominant diffraction component is computed, this component often can not be reliably identified beforehand, and never in the general case of multiple diffraction. In Figure 4.16 the total diffraction field is represented both as a phasor sum and as a power sum, or mean field strength, computed as described in Ikegami, et al. [4].

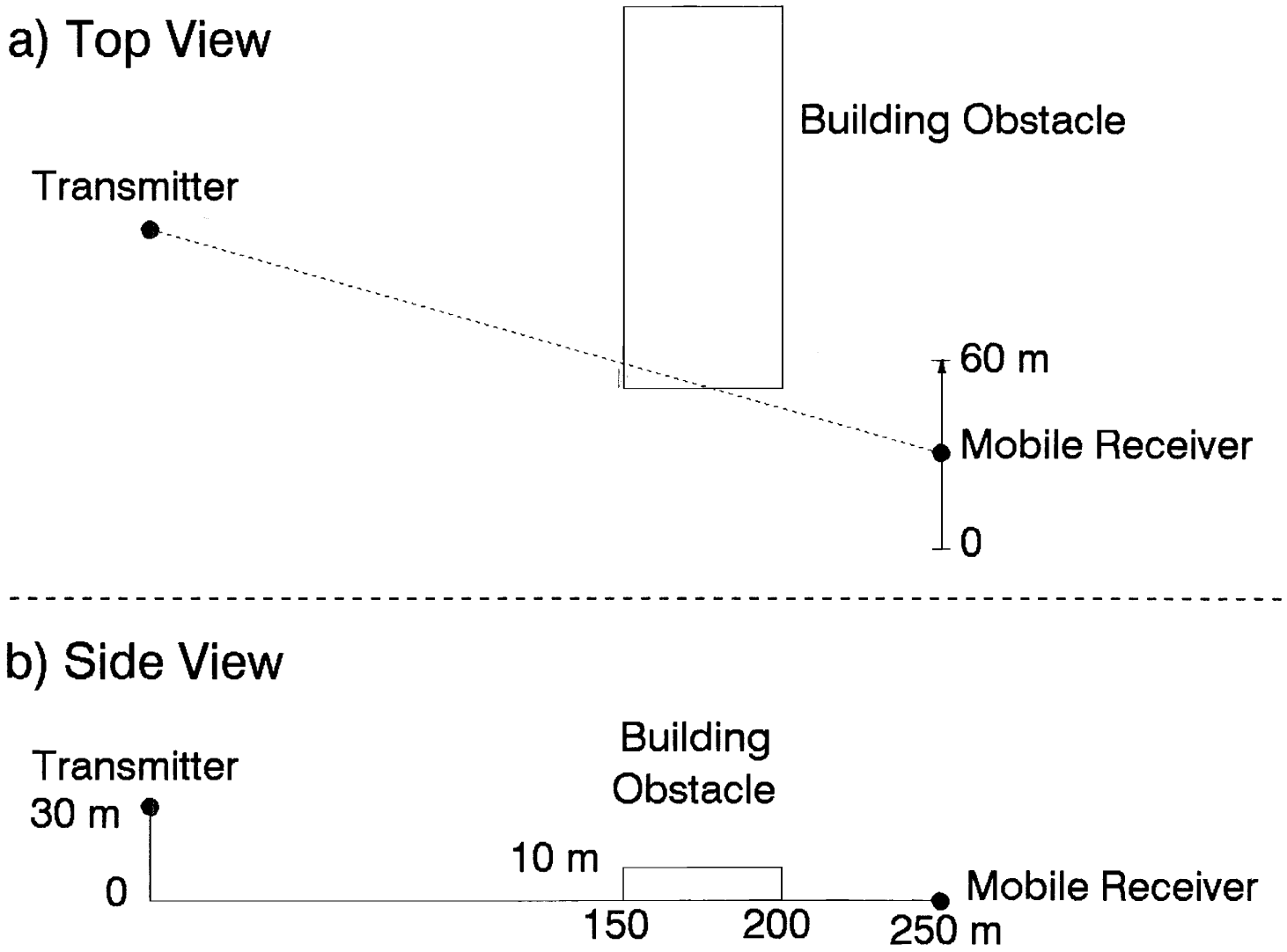


Figure 4.11. Three-Dimensional Single Diffraction Example

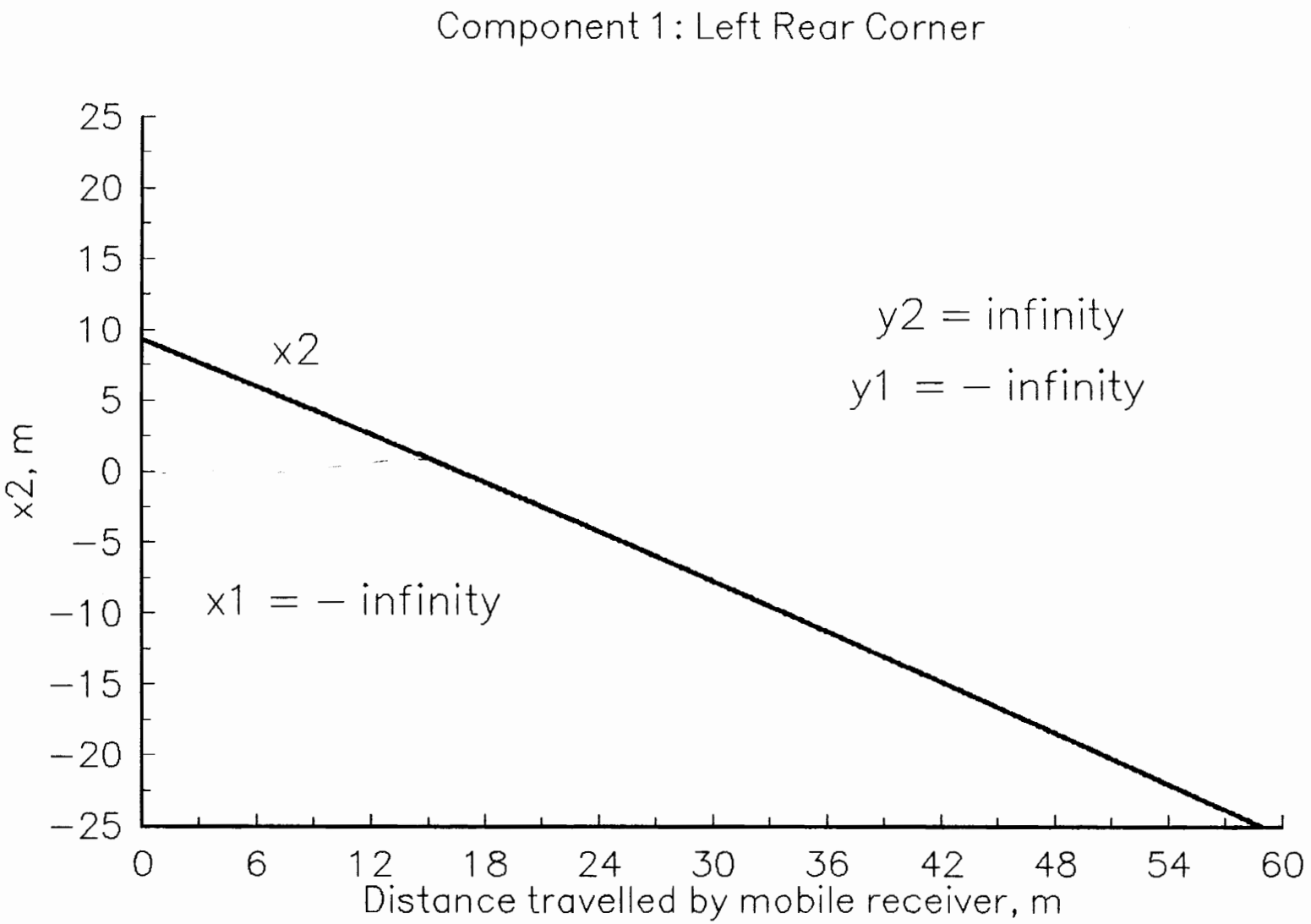


Figure 4.12. Bounds on Region of Integration: Left Rear Corner

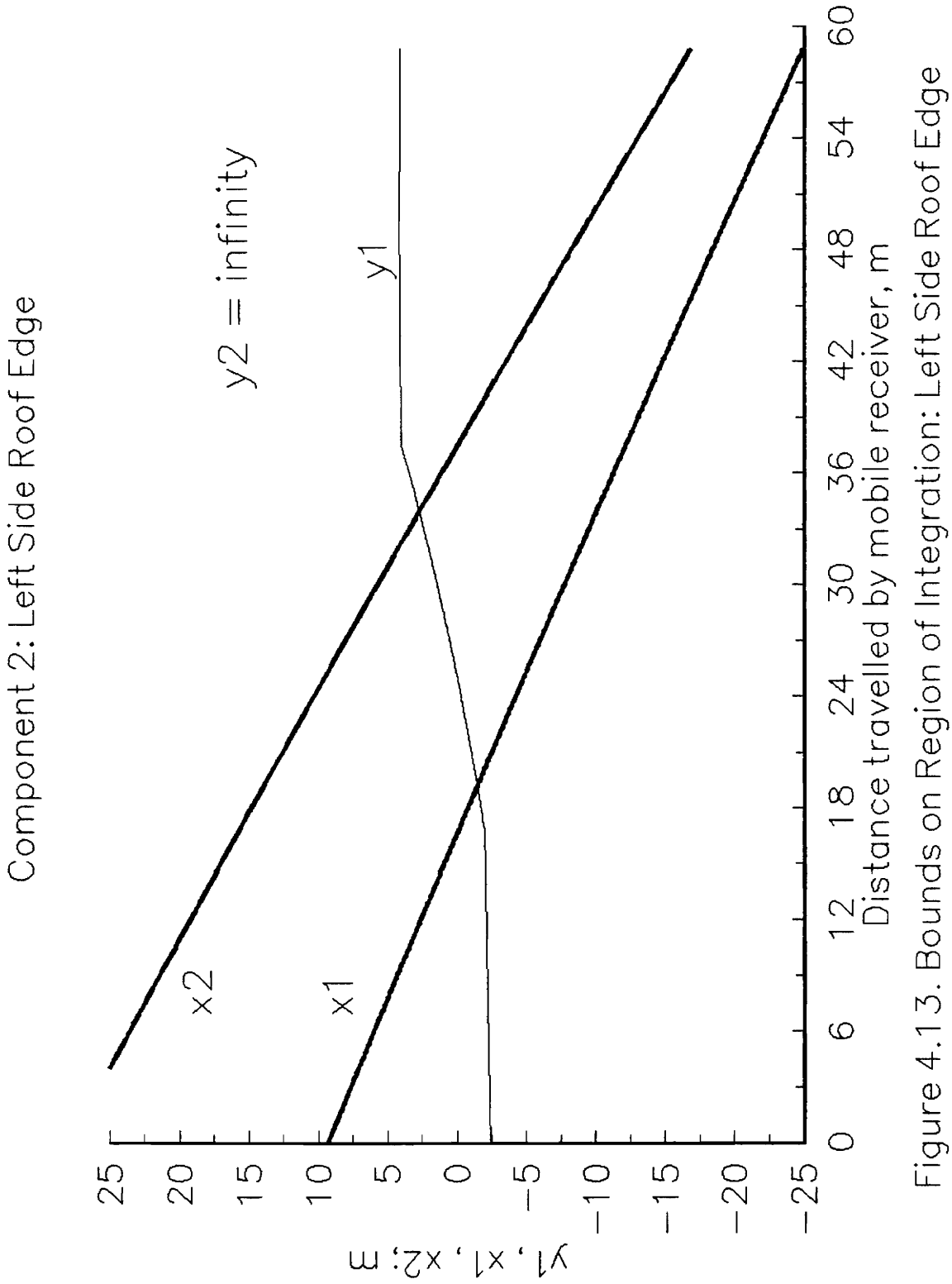


Figure 4.13. Bounds on Region of Integration: Left Side Roof Edge

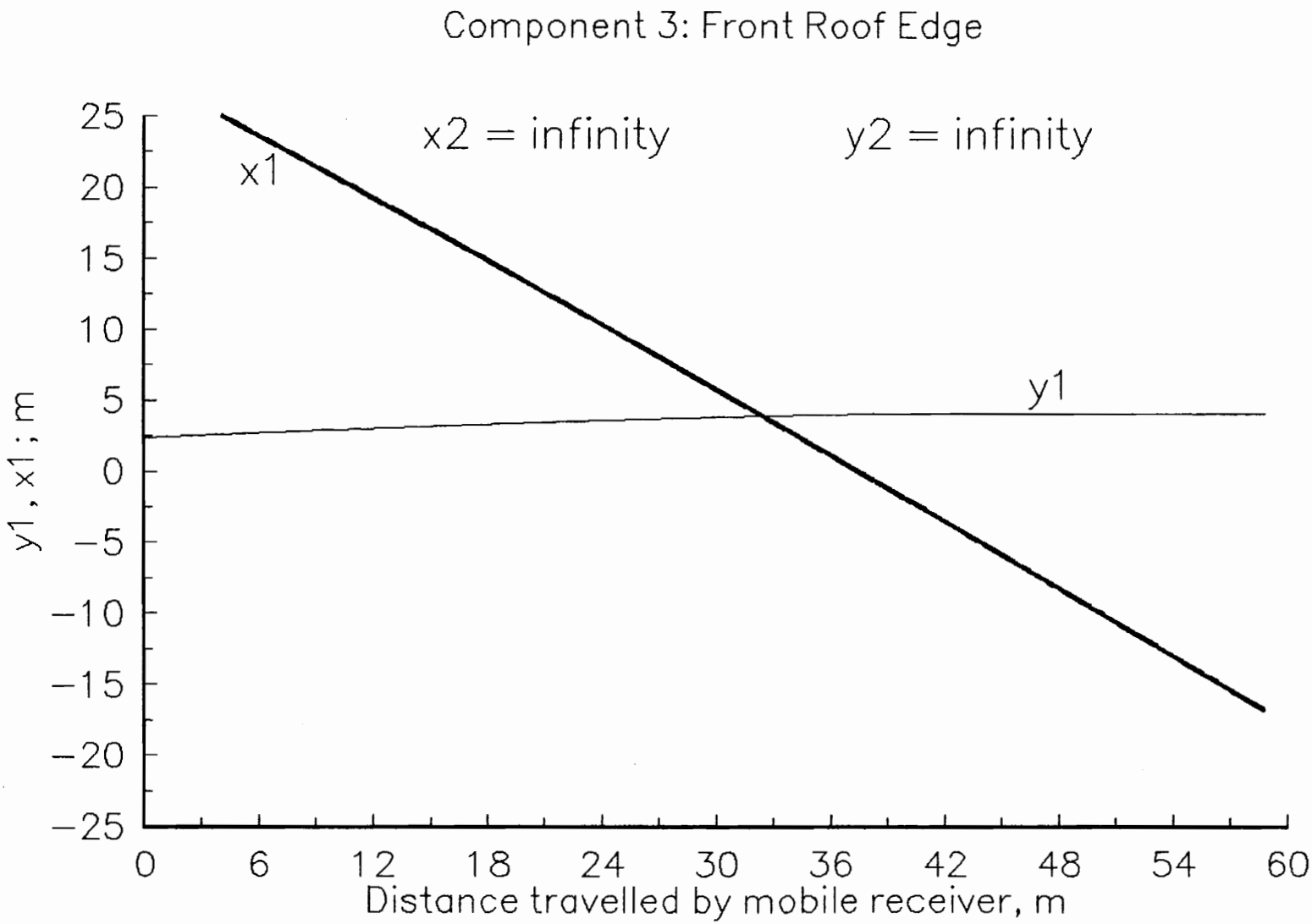


Figure 4.14. Bounds on Region of Integration: Front Roof Edge

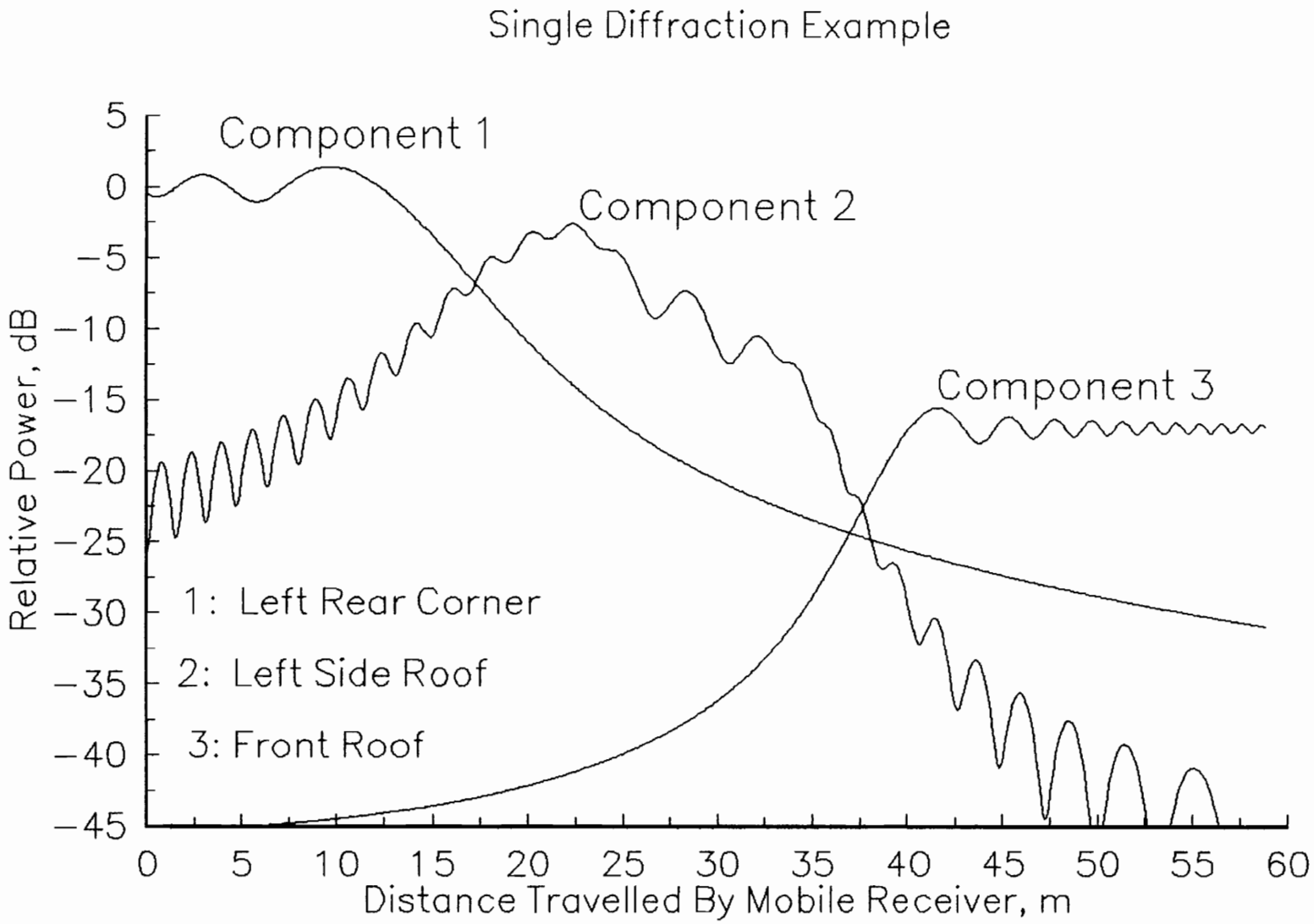


Figure 4.15. Diffraction Field Components

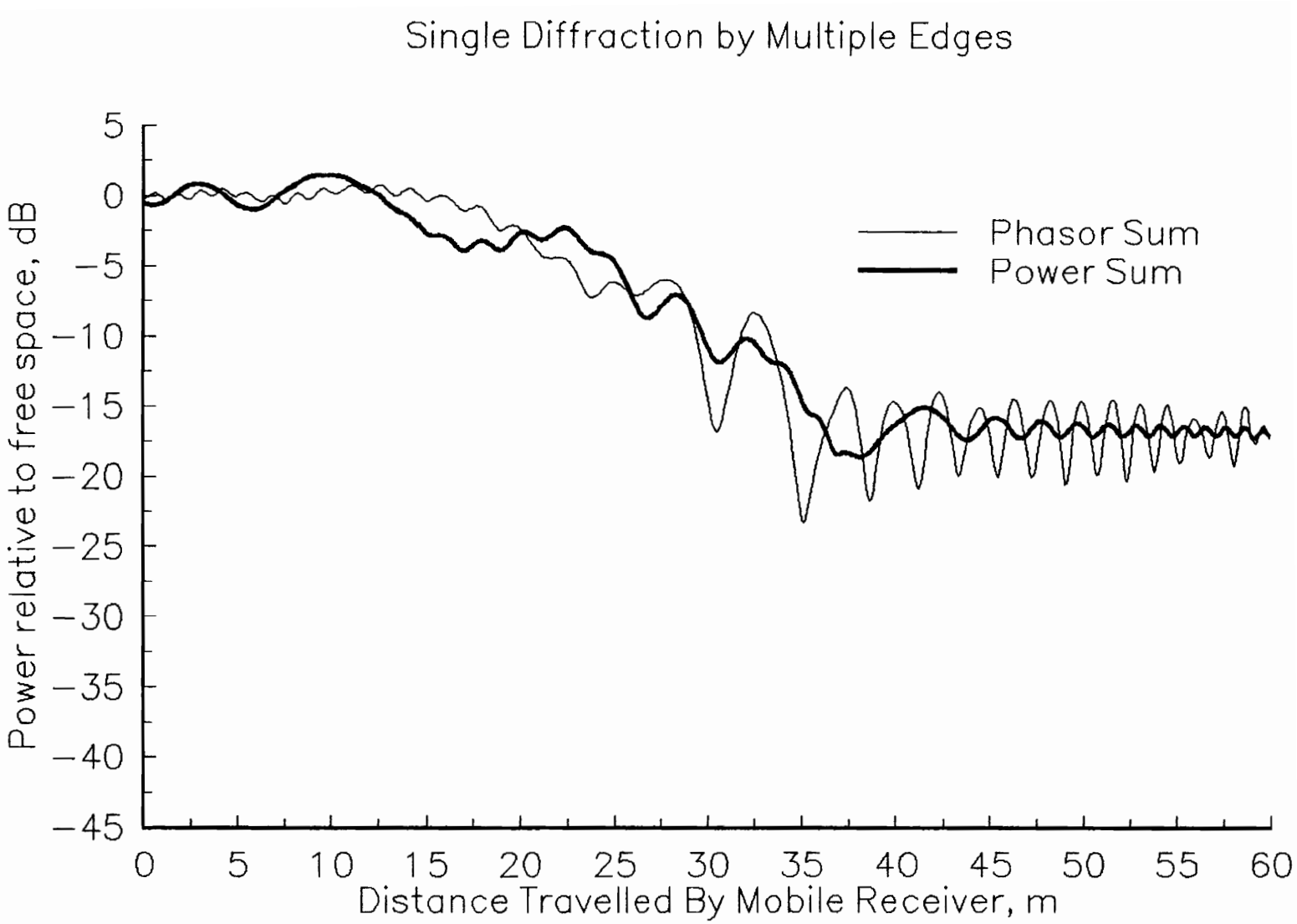


Figure 4.16. Total Diffraction Field

4.4 Model Validation By Comparison With Measurements

4.4.1 Experimental Measurements

Continuous-wave measurements made on the Virginia Tech campus at 914 MHz were used for validation of predictions throughout the development of these models. The antennas in all cases were omni-directional and vertically polarized. Figure 4.17 shows the geometry of the area where the first of two measurement campaigns was performed. The transmitter was on top of Whittemore Hall, the building on the left of the picture, and the receiver was on a mobile cart that was pushed down several sidewalks on campus. The two measurement runs of this campaign that are used here for model comparison are shown in the figure. These two tracks were chosen because they exhibit interesting diffraction characteristics. The building that diffracts field down to measurement track SL22 is divided into two sections of different height, while the obstacles shadowing track SL32 are of several different heights. Since the field incident on track SL22 has undergone primarily single diffraction, we will compare those measurements with the single diffraction model. We will consider track SL32 in Chapter 5 as it includes multiple diffraction effects. At that point we will also consider the second measurement campaign, which attempted to isolate the effects, including multiple diffraction, of single buildings.

4.4.2 Predictions Compared with Measured Data, Track SL22

Figure 4.18 presents a comparison of predicted signal strength and

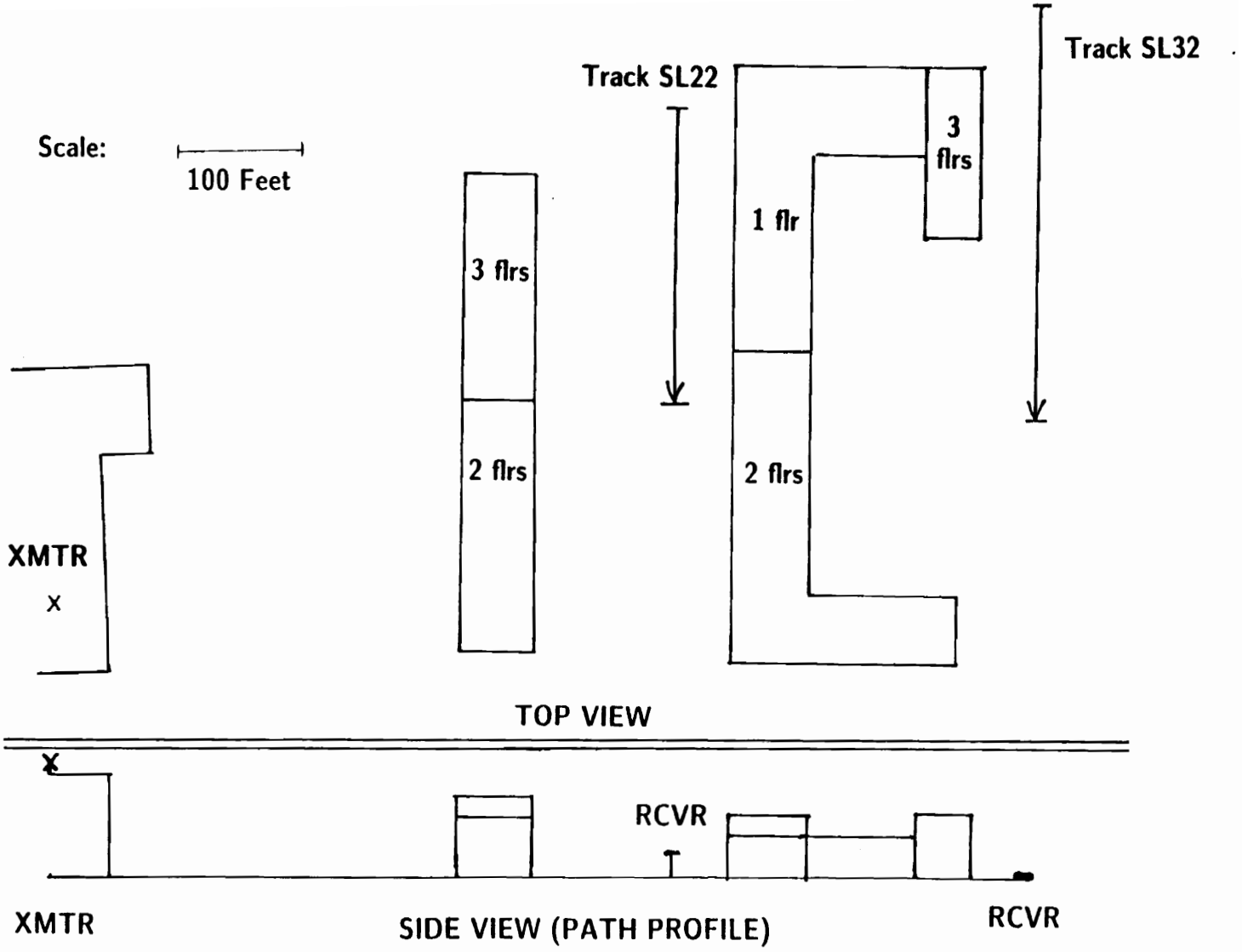


Figure 4.17. Virginia Tech Campus Measurement Area

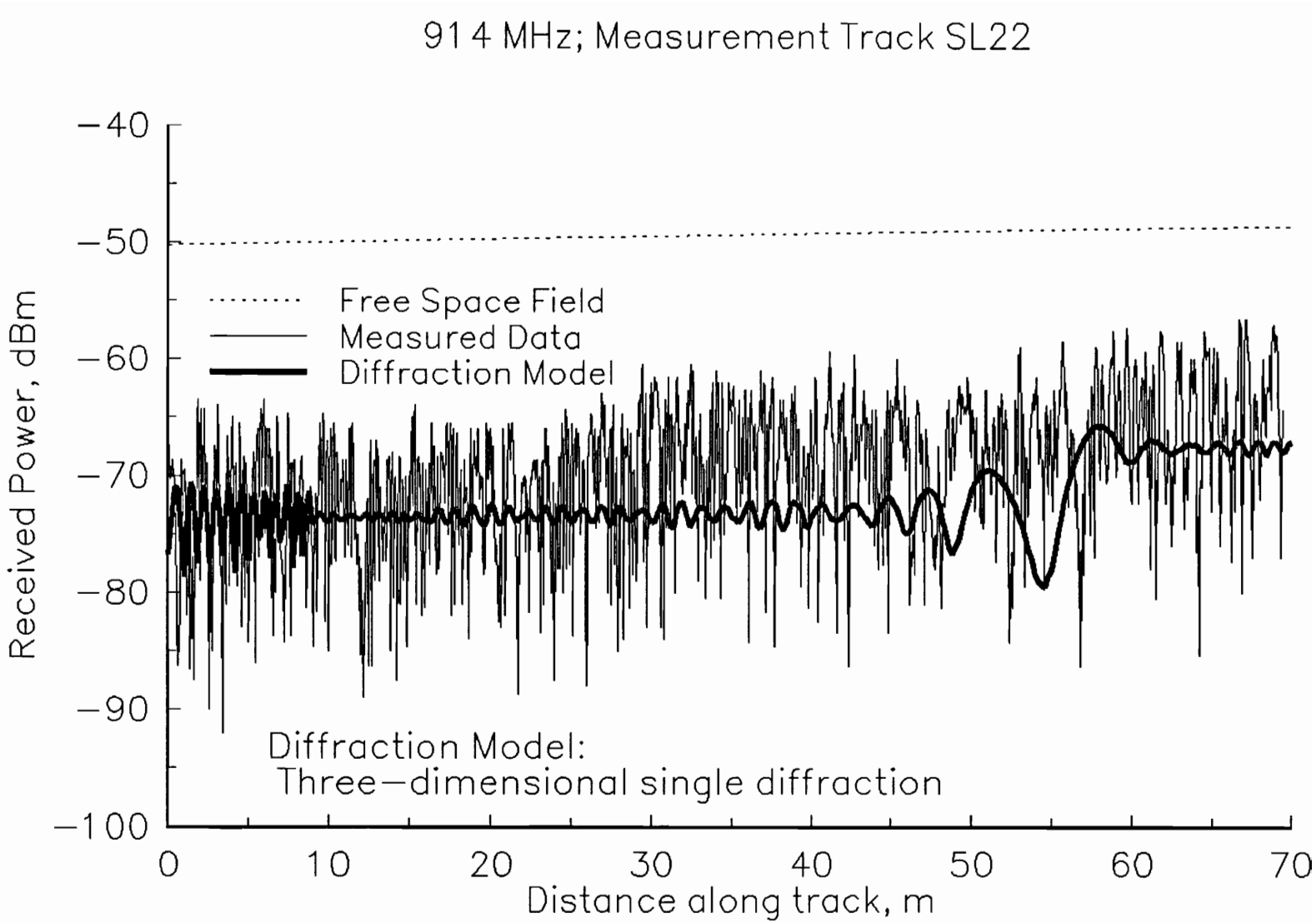


Figure 4.18. Predictions Compared with Unfiltered Measurements

measured data on track SL22. The prediction curve represents the complex phasor sum of the individual diffraction field components, and exhibits some interference. The strengths of these field components are presented in Figure 4.19. Although the diffraction model predicts interference, it is not expected that the predicted variability will always match measured data because much of the actual signal variations are due to reflected and scattered energy that the diffraction model does not include. It is hoped, however, that the mean predicted signal strength will match well with the mean measured signal strength and that some of the character of the measured signal variability can be reproduced.

In the region between 40 m and 55 m in Figure 4.18, the wavelength and amplitude of spatial variations in the model prediction increases until the radio path passes the 3 floor - 2 floor discontinuity in the obstacle building, and then decreases again, as the field settles to a new, higher value. This same tendency can be seen in the measured data, although the measured peaks and nulls do not line up with the predictions, and there are extra variations superimposed on the measured signal.

4.4.3 Comparison with Filtered Measurements, Track SL22

In order to validate the prediction of mean field strength, it is useful to apply a sliding window filter to the measured data. This provides an approximation to the local mean by filtering out the higher frequency variations.

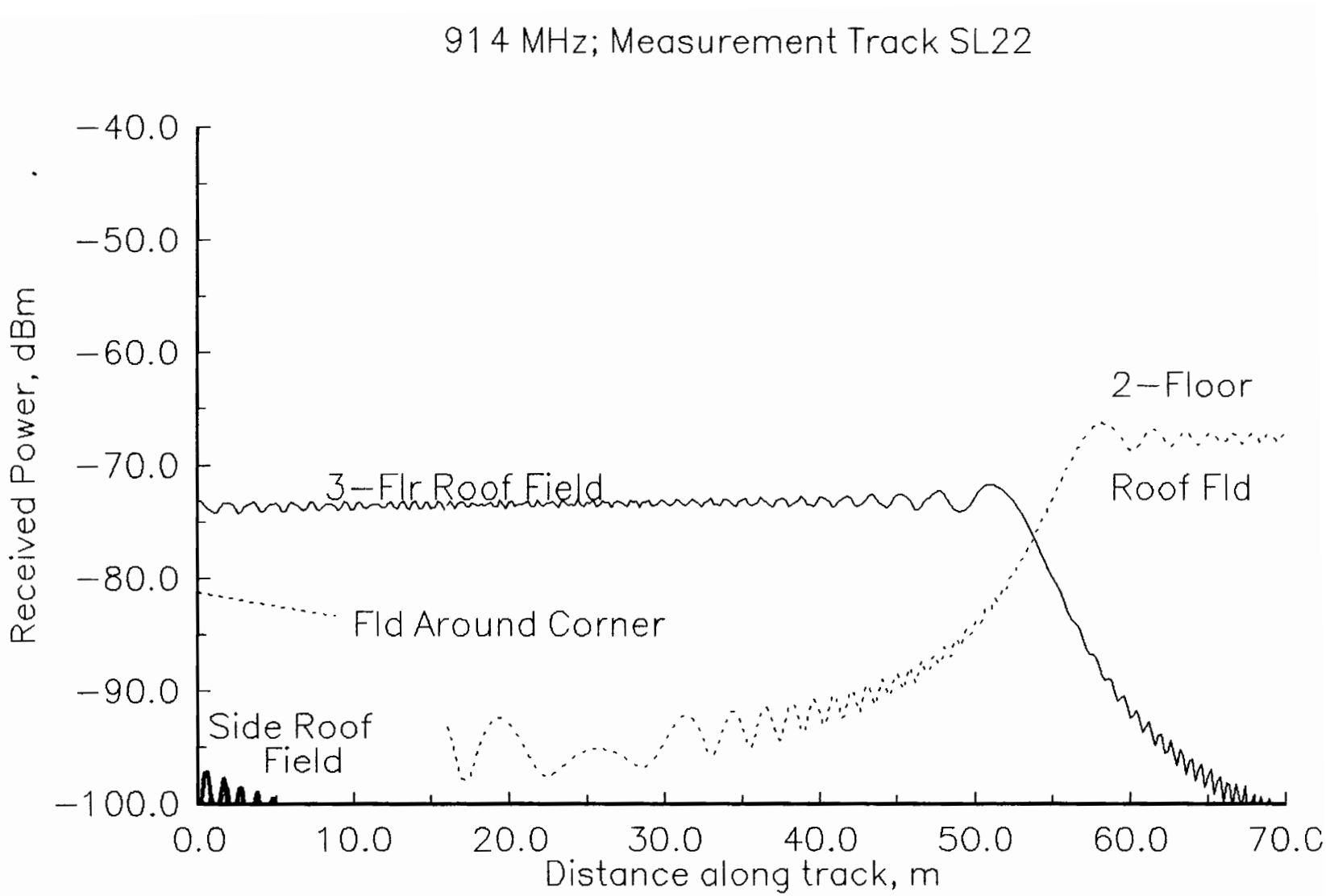


Figure 4.19. Predictions of Individual Field Components

The result of applying a Hamming window with size equal to 5 wavelengths is presented in Figure 4.20, along with the model predictions. The filtered measurements seem to experience two general signal enhancements as the receiver moves down the track, each greater than the slow increase seen in the free space value. The first, at 25-30 m, is not matched by an increase in the predicted strength. This suggests the appearance of reflected or scattered field that dominates over the diffracted field. Possible sources include reflections off of the building behind the measurement track or the parking lot in front of the track, and scattering off of cars in the lot.

The second noticeable increase occurs at about 55 m and coincides with an increase in the predicted signal strength, due to less severe diffractive shadowing by the shorter portion of the building. The two curves agree to within 5 dB after this point, indicating that the direct diffraction field again dominates over other sources. These results indicate that the use of actual building heights and locations can provide for accurate predictions, but a diffraction model alone will only provide good results when the diffracted field is dominant over other sources of field.

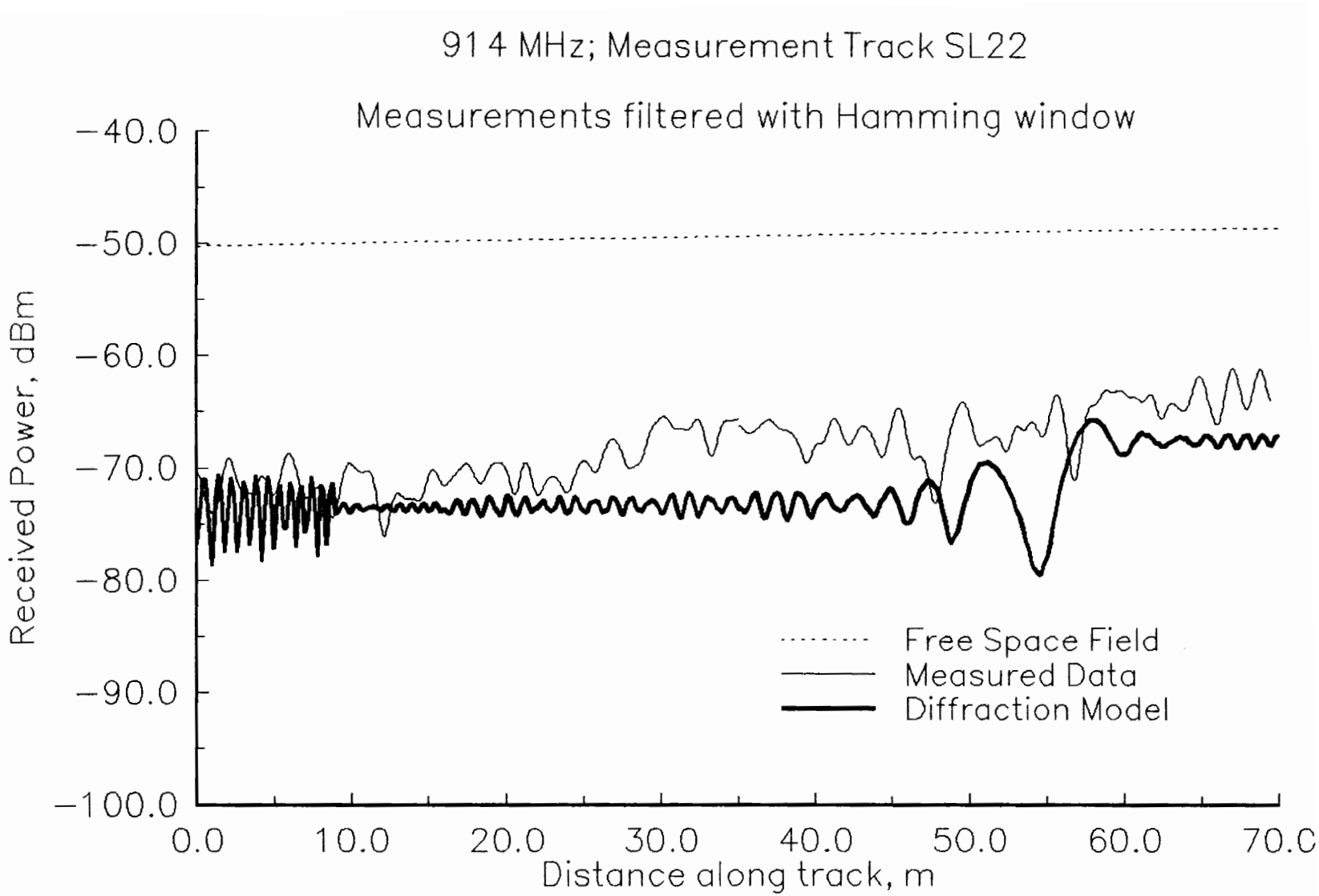


Figure 4.20. Predictions Compared with Filtered Measurements

V. MULTIPLE DIFFRACTION MODEL

5.0 Introduction

The three-dimensional single diffraction model is extended here to include multiple successive diffracting edges. The technique of recursively computing diffracted fields past successive obstacles is based on that of Whitteker [28], who applied it to large terrain obstacles such as mountains and hills that were assumed to extend to infinity in the horizontal direction. We refer to this technique of Whitteker as two-dimensional multiple diffraction. However, as discussed above, in an area in which buildings may be the obstacles to a line of sight, signal may be received in a shadow region by diffraction past a horizontal roof edge or a vertical corner edge, necessitating the use of a three-dimensional model. We have therefore combined Whitteker's multiple diffraction model with our three-dimensional single diffraction model, giving us a three-dimensional multiple diffraction model. This model can be applied to simulate the proposed microcellular system where the transmitter and the receiver are both well below the rooftops. Most importantly, the model applies in the most general situation where diffraction can take place past several rooftops as well as around corners, and the dominant mechanism is not obvious.

5.1 The Multiple Diffraction Integral

The single diffraction model requires that the wave incident on each secondary Huygens source, Q, in the unobstructed area surrounding an obstacle has undergone only spherical wave expansion in free space. If, instead, the incident wave has experienced diffraction by a preceding edge, as in Figure 5.1, a multiple diffraction model must be used. The method we will use involves the evaluation of the complex field at many points in the xy plane in order to approximate the field distribution, the same approach used by Whitteker [28] and Walfisch and Bertoni [11]. However, while the solutions in both [28] and [11] included only an integral over the vertical dimension, our solution involves the product of integrals over each of the two dimensions of the xy plane, in order to include the effects of transverse variations.

5.1.1 Derivation of the Multiple Diffraction Integral

We will begin with the single diffraction approach of Chapter 4 and reformulate it to allow for the consideration of a diffracting edge between S and Q. Refer to Figure 4.3, Equation (4.1), and the accompanying text for geometry and variable definitions.

Consider (4.1), the general integral for diffraction past a single edge, and assume, as discussed in Section 4.3, that the obliquity factor, $\cos(\alpha)$ - $\cos(\beta)$, is always equal to 2. The field at Q (Figure 4.3) is a function of free space wave

Figure 5.1. Multiple Diffraction

expansion,

$$E(Q) = \frac{e^{ikr_s}}{r_s}\,, \tag{5.1}$$

and can be separated out of the integrand, leaving

$$E(P) = \frac{-i}{\lambda} \int\int_R E(Q)\, \frac{e^{ikr_p}}{r_p}\, dR\,. \tag{5.2}$$

The first term of the integrand of (13b), E(Q), describes the propagation from S to a point Q in region R, and the remaining terms, e^{ikr_p} and $\frac{1}{r_p}$, describe the phase shift and amplitude change, respectively, introduced between Q and P.

The expression for the field at Q can be broken down as follows. With the Fresnel phase approximation, (4.7), equation (5.1) becomes

$$E(Q) \simeq \frac{e^{ik(s + \frac{x^2}{2s} + \frac{y^2}{2s})}}{r_s}\,. \tag{5.3}$$

When there is an intervening horizontal edge (parallel to the x axis), as in Figure 5.1, where the x axis is directed into the page, the field at Q is a function of the diffraction past that edge. However, the x dependence is approximately the same as that for free space expansion, as variation in x affects the path length but not the degree of shadowing. (Actually, a change in path length does change the degree of shadowing, as it enters into the determination of the diffraction

parameters; however, it is a small effect compared with that of variation in the other direction and will be neglected in the interests of reducing the number of dimensions of the problem.) This is an important distinction, as it allows us to separate out the dependence on x in (5.3) and to characterize the field distribution in the xy plane with discrete function evaluations in only the y dimension, and with an analytic description of the field variation in the x dimension. We can write

$$E(Q) \simeq E_Q(y)\, e^{ik\frac{x^2}{2s}} \tag{5.4}$$

where

$$E_Q(y) \simeq \frac{e^{ik(s + \frac{y^2}{2s})}}{r_s} \tag{5.5}$$

if there is no obstacle between S and Q. In the multiple diffraction case, we will substitute explicit evaluations of the field at Q as a function of y (at $x = 0$) for the relation (5.5).

The terms of the integrand in (5.2) that represent the amplitude and phase change incurred between Q to P can be simplified using the amplitude and phase approximations of Chapter 4 such that

$$\frac{e^{ikr_p}}{r_p} \simeq \frac{e^{ik(p + \frac{x^2}{2p} + \frac{y^2}{2p})}}{p}. \tag{5.6}$$

With the substitutions of (5.4) and (5.6) in (5.2), and rearrangement of terms, the solution for the field at P can be written

$$E(P) = \frac{-i}{\lambda p} e^{ikp} \int\int_R E_Q(y)\, e^{ik\left(\frac{x^2}{2s} + \frac{x^2}{2p}\right)}\, e^{ik\frac{y^2}{2p}}\, dR\,, \tag{5.7}$$

and the double integral can be separated into the product of two single integrals,

$$E(P) = \frac{-i}{\lambda p} e^{ikp} \int_{x_1}^{x_2} e^{ik\frac{x^2(s+p)}{2sp}}\, dx \int_{y_1}^{y_2} E_Q(y)\, e^{iky^2/2p}\, dy. \tag{5.8}$$

The integral over x in (5.8) may be written in the form of a Fresnel integral through a change of variable (as in the single diffraction solution),

$$E(P) = -i\, e^{ikp} \sqrt{\frac{s}{2\lambda p(s+p)}} \int_{\xi_1}^{\xi_2} e^{i\pi\xi^2/2}\, d\xi \int_{y_1}^{y_2} E_Q(y)\, e^{iky^2/2p}\, dy \tag{5.9}$$

where

$$\xi_1 = x_1 \sqrt{\frac{2(s+p)}{\lambda sp}}\,; \qquad \xi_2 = x_2 \sqrt{\frac{2(s+p)}{\lambda sp}}\,. \tag{5.9a,b}$$

Equation (5.9) is the multiple diffraction integral. This result is similar to (9) of Whitteker, [28], or (1) of Walfisch and Bertoni, [11], except that in [28] and [11], the limits on x extend to $\pm\infty$, in accord with the infinite knife edge model. The limits on x and y in (5.9) and (5.9a,b) measure the projection of the

diffracting edge on the xy plane, derived according to the techniques described in Chapter 4 (Sections 4.3 - 4.3.4). The upper limit on y, the vertical dimension, is always taken as infinity.

5.1.2 Alternate Form of the Multiple Diffraction Integral

In the derivation of (5.9) and the characterization of the field at points Q, a free space expansion is assumed in the x (horizontal) dimension and the variation in the y (vertical) dimension is explicitly evaluated. This formulation is based on the assumption that the previous diffracting edge is horizontal, causing the field at points Q in the xy plane to vary strongly in amplitude and phase with variation only in the y dimension, as this affects the degree of shadowing caused by the previous edge. If, on the other hand, the edge introducing diffraction at Q is a vertical edge, the corner of a building, then the opposite case is true: variation with x affects the degree of shadowing and variation with y can be assumed to be only a function of free space expansion. Thus, by separating out the y dependence instead of the x dependence in (5.3), we obtain

$$E(P) = \frac{-i}{\lambda p} e^{ikp} \int_{y_1}^{y_2} e^{ik\frac{y^2(s+p)}{2sp}} dy \int_{x_1}^{x_2} E_Q(x)\, e^{ikx^2/2p}\, dx \qquad (5.10)$$

or

$$E(P) = -i\, e^{ikp} \sqrt{\frac{s}{2\lambda p(s+p)}} \int_{\eta_1}^{\eta_2} e^{i\pi\eta^2/2}\, d\eta \int_{x_1}^{x_2} E_Q(x)\, e^{ikx^2/2p}\, dx \qquad (5.11)$$

where

$$\eta_1 = y_1 \sqrt{\frac{2(s+p)}{\lambda sp}} \; ; \qquad \eta_2 = y_2 \sqrt{\frac{2(s+p)}{\lambda sp}} \; . \tag{5.11a,b}$$

The upper limit on y and η is always taken as ∞. The limits on the range of x are (x_1,∞) or $(-\infty,x_2)$, for field propagating around the right or left side of the building, respectively.

Note that the need for (5.11) instead of (5.9) is determined not by the nature of the diffracting edge at Q, but by the nature of the diffraction field at Q, which is a function of the previous edge.

5.1.3 Evaluation of the Multiple Diffraction Integral

The first integral in both (5.9) and (5.11) is a Fresnel integral, which is relatively straightforward to evaluate; we use the techniques described in Section 6.1. The second integral involves evaluations of the field at distinct points, Q. To compute the field at these intermediate points, each point Q is treated as a new observation point, P′, and the diffraction field is computed at P′ according to (5.9) or (5.11). The process is repeated in this recursive fashion until there is only one obstacle between S and P′, and only free space wave expansion between S and Q′. Then the single diffraction at P′ is computed according to (4.11) and

the multiple diffraction integrals are evaluated. The geometry for the case of double edge diffraction is shown in Figure 5.2, where points Q in the aperture of the edge nearest to P are obstructed by a prior edge and must be considered as new observation points, P′.

Before discussing the evaluation of (5.9) and (5.11) we will investigate the behavior of $E_Q(y)$ in (5.9) with an example. Let us consider the simple case of a single building introducing diffraction by two roof edges in succession, as in Figure 5.3, where the building is long enough (in the x direction) that diffraction around the sides is negligible and the edges can be assumed to be infinitely long. The two diffracting roof edges correspond to the edges in Figure 5.2, where the field must be sampled at points Q or P′ above the trailing roof edge (the second edge encountered by the propagating wave). We compute the field E(Q) with the single diffraction integral, (4.11), as a function of y (at x = 0) for three different transmitter heights. The signal power variations relative to free space are plotted in Figure 5.4. In this single-building situation, the field at Q is the well-known result of a single diffracting edge, and the field at P is found as a double application of the knife-edge integral (4.11). This appears as though it should have an analytical solution, which would eliminate the need for explicitly evaluating the field at points Q. Indeed, Millington et al. [20] presented an analytical solution, but showed that the extension of that approach to more than two edges was not practical. In addition, variations in the transverse dimension were not easily included, unlike in the more flexible numerical solution described

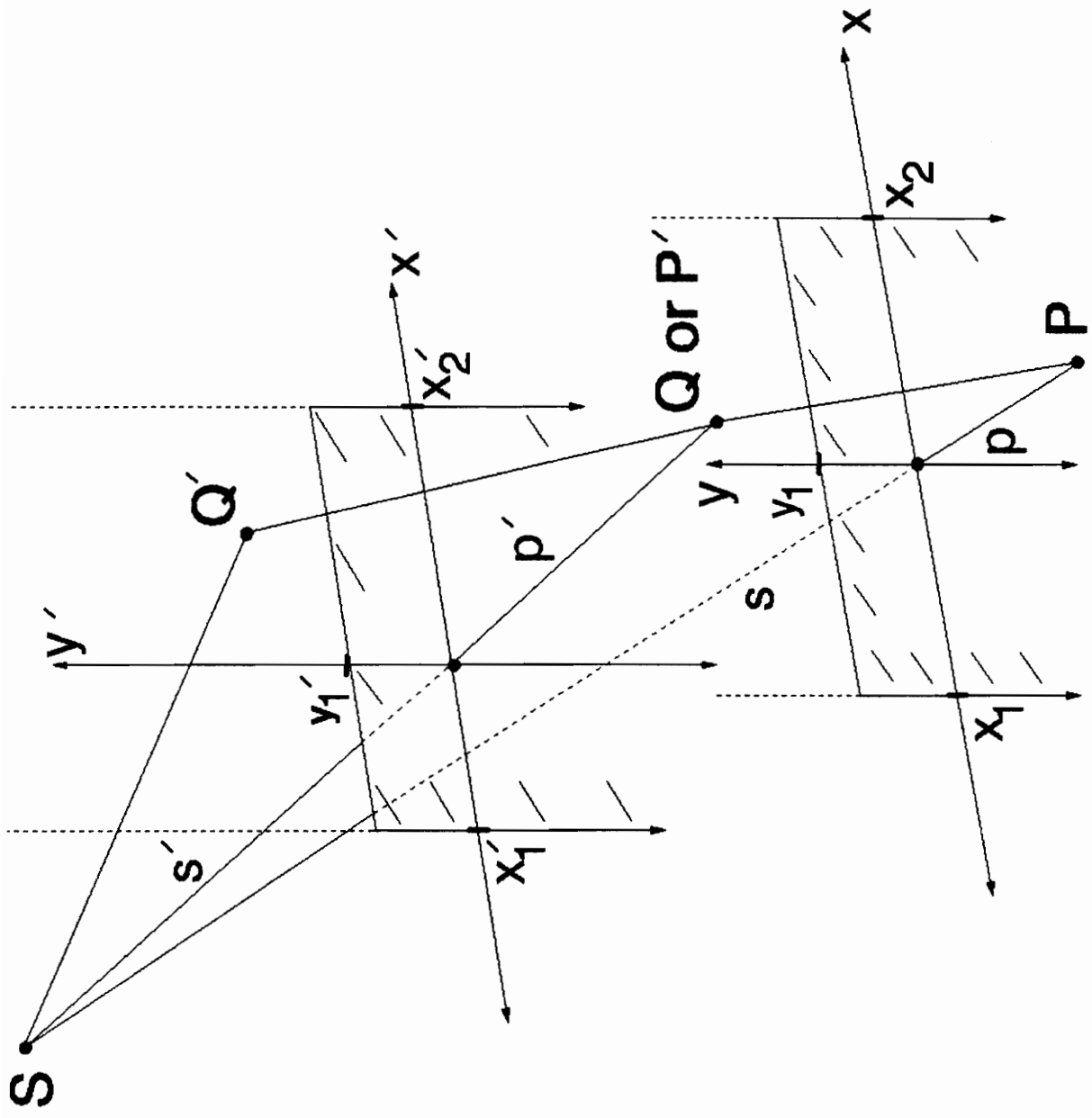


Figure 5.2. Double Building Edge Diffraction Geometry

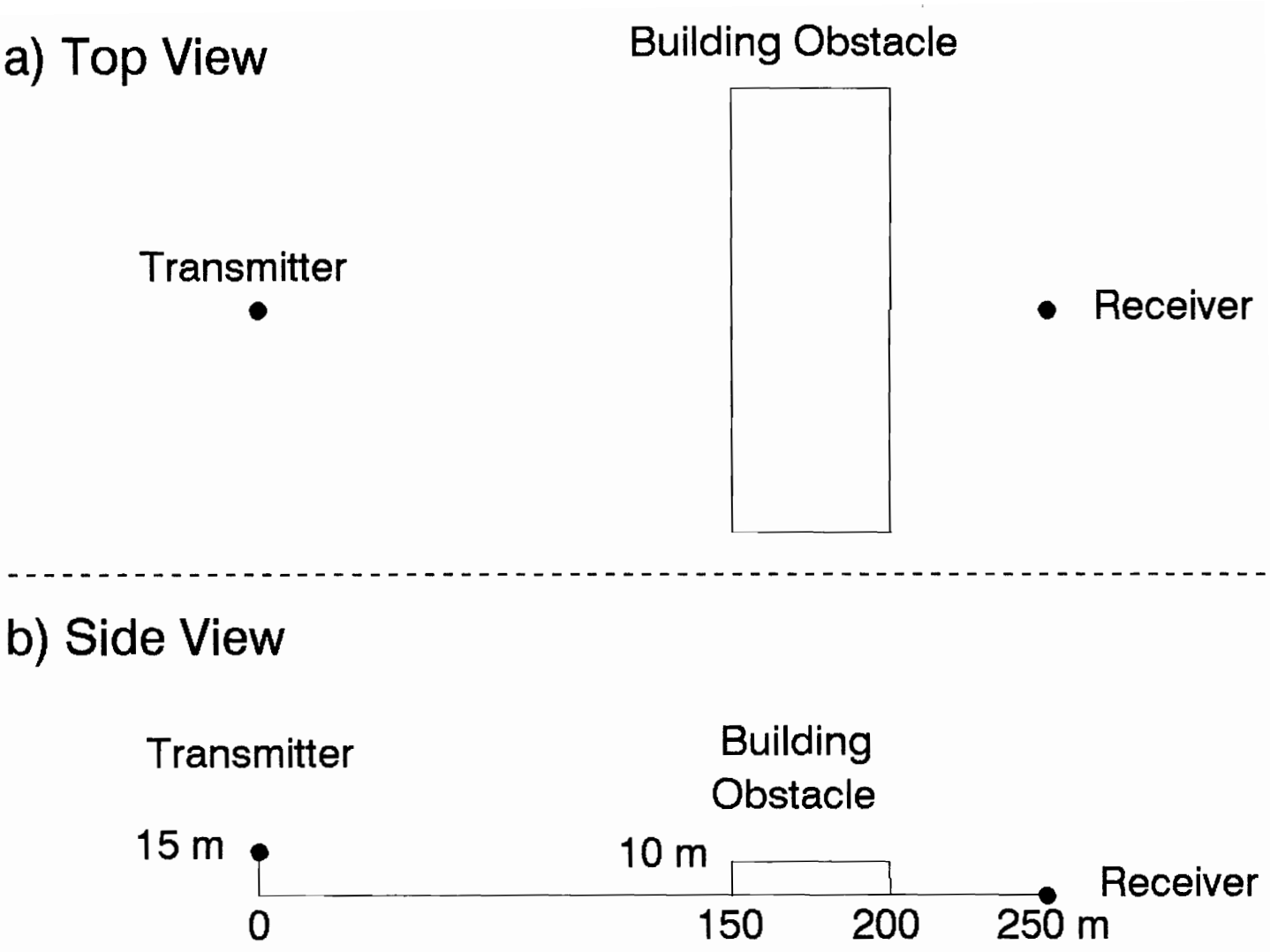


Figure 5.3. Single-Building Multiple Diffraction Example

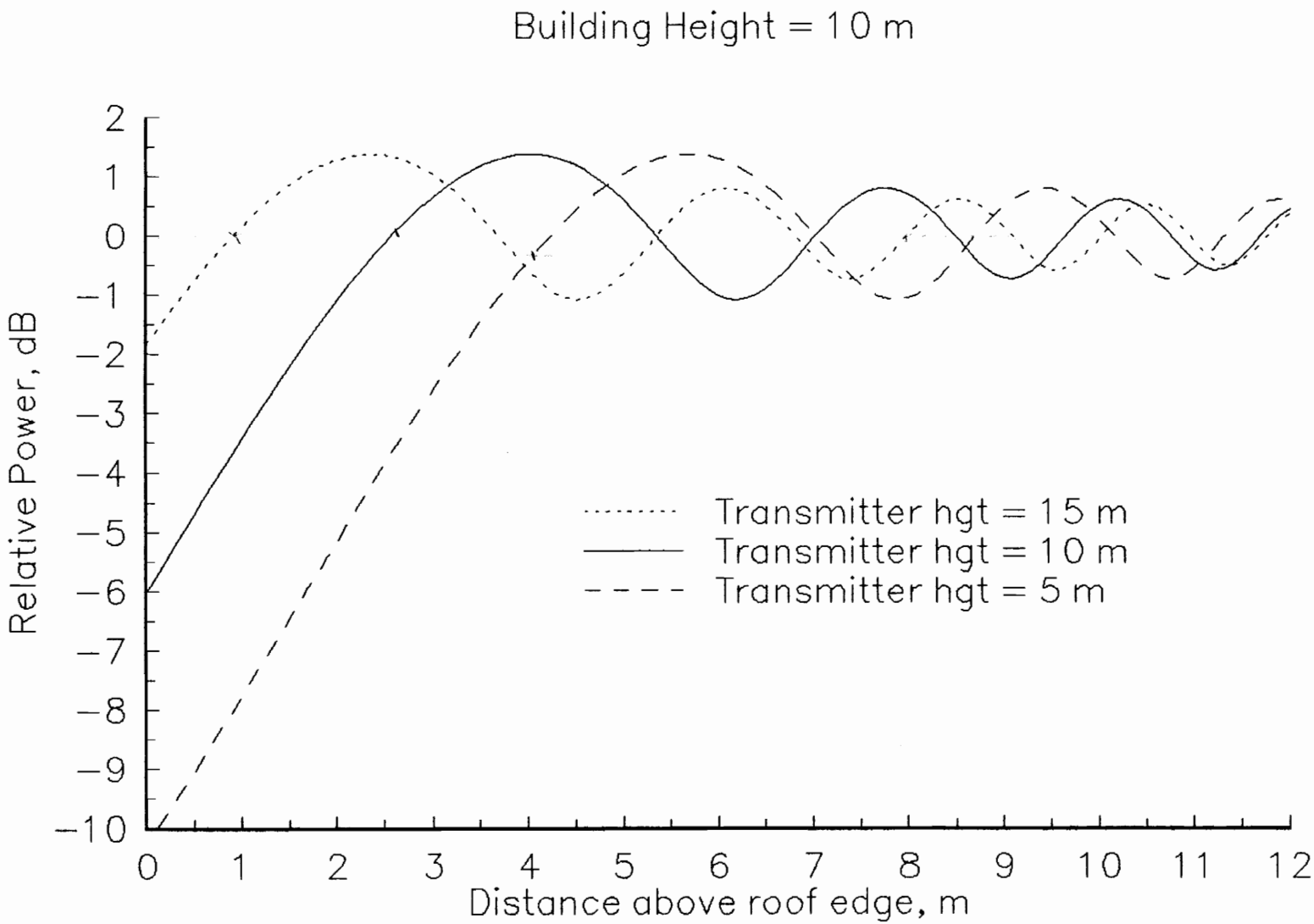


Figure 5.4. Field Distribution Above Trailing Roof Edge

here.

Of the three situations depicted in Figure 5.4, the dotted line describes propagation from a source above the rooftop, the solid line illustrates the case of grazing incidence (as seen at P′), and the dashed line describes the case where the source is below the rooftop. Only the case of the 5 m transmitter height, described by the dashed line, undergoes any obstruction of the line of sight. All of the curves show the characteristic amplitude oscillations (seen also in Figure 4.5) that converge at infinity to the free space level; and the curves for the two lower transmitters show a distinct region of smooth increase in power with decrease in shadowing or increase in clearance. The transition point between the two regions can be defined at different places; we will use the point of grazing clearance where a line of sight is just visible at the observation point, P′.

In order to evaluate the multiple diffraction integral over y (from y_1 to ∞) in (5.9), we divide the total integration range into two parts, following the approach of Whitteker [28]. The first part is a finite region, beginning at y_1, that extends out at least as far as the transition point defined above, while the second part averages over the interference fringes extending to infinity. The integral over the first region is found through sub-division into many small intervals. The integral over the second region uses only a few, widely-spaced sample points, and a single integration step to infinity. Further details of the numerical methods used to evaluate the integrals, and the optimization of point spacing

and number are included in Chapter 6.

5.1.4 Example Calculations

The behavior of this multiple diffraction model is illustrated first with some examples of an idealized two-dimensional case, where the building edges are long enough to be modeled as infinite and the building is oriented with its long edges transverse to the radio path. We will use the situation shown in Figure 5.5 and consider various combinations of the individual elements. First, taking only the wider building on the right (removing the first building completely), we set both the transmitter and receiver at a 2 m height and raise the height of the building from 0 to 20 m. The results of computing the multiple diffraction by the two roof edges of the building and of computing only the single diffraction by the trailing edge are plotted in Figure 5.6. Whereas the single diffraction assumption is that of an infinitely thin building, this figure demonstrates the relative importance of considering the width of the building. The error incurred in neglecting the width ranges from 3 dB at 10 dB diffraction loss to 12 dB at 35 dB loss.

Next, considering both buildings, with heights of 15 and 10 m for Buildings 1 and 2 (Figure 5.5), respectively, and a receiver height of 0 m, we drop the transmitter from a height of 30 m to only 5 m. The solid line in Figure 5.7 is the result of computing the total impact of all 4 edges of the 2 buildings (where the edges act in succession). The other curves demonstrate the impact of

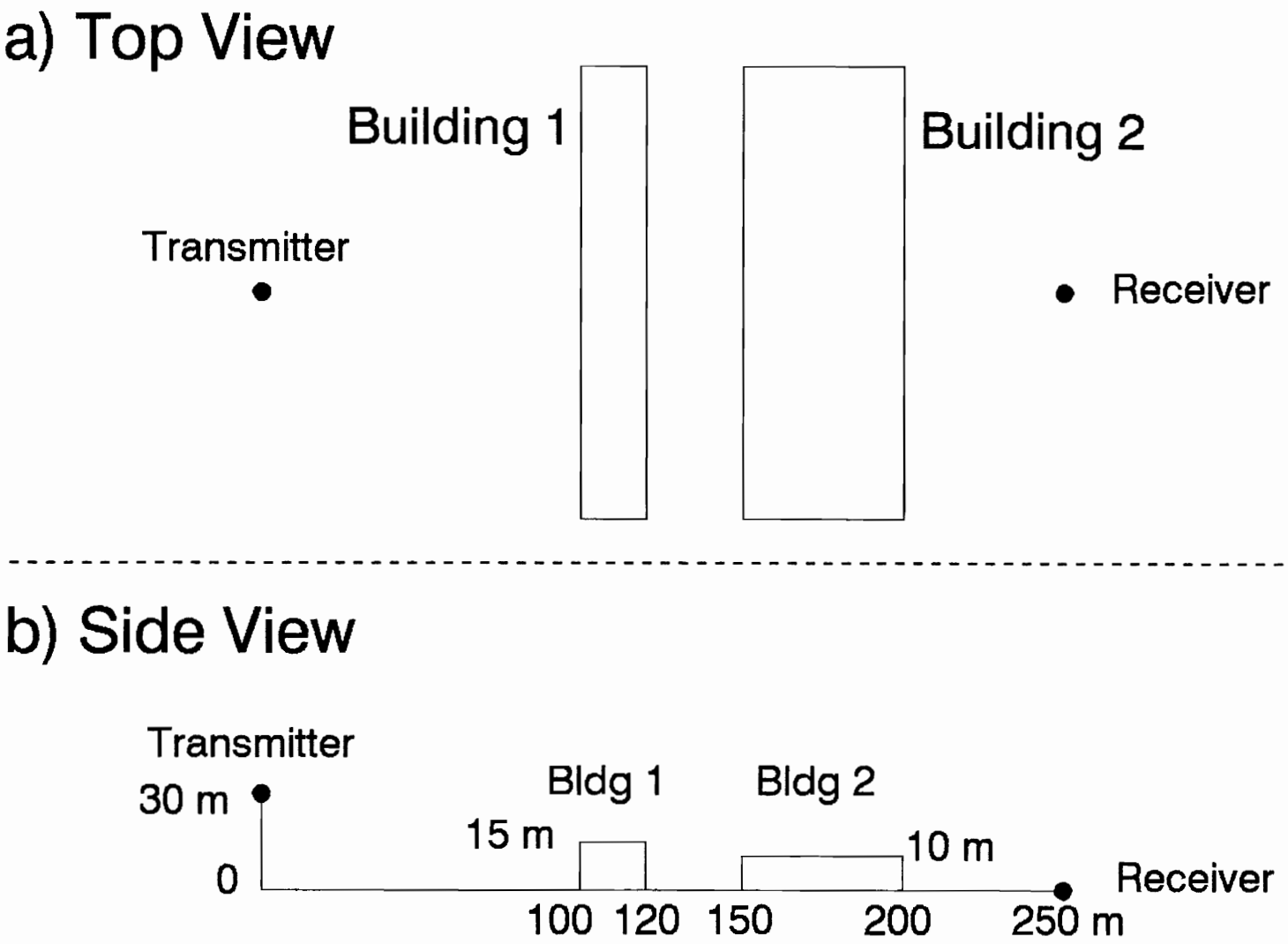


Figure 5.5. Multiple Diffraction Example

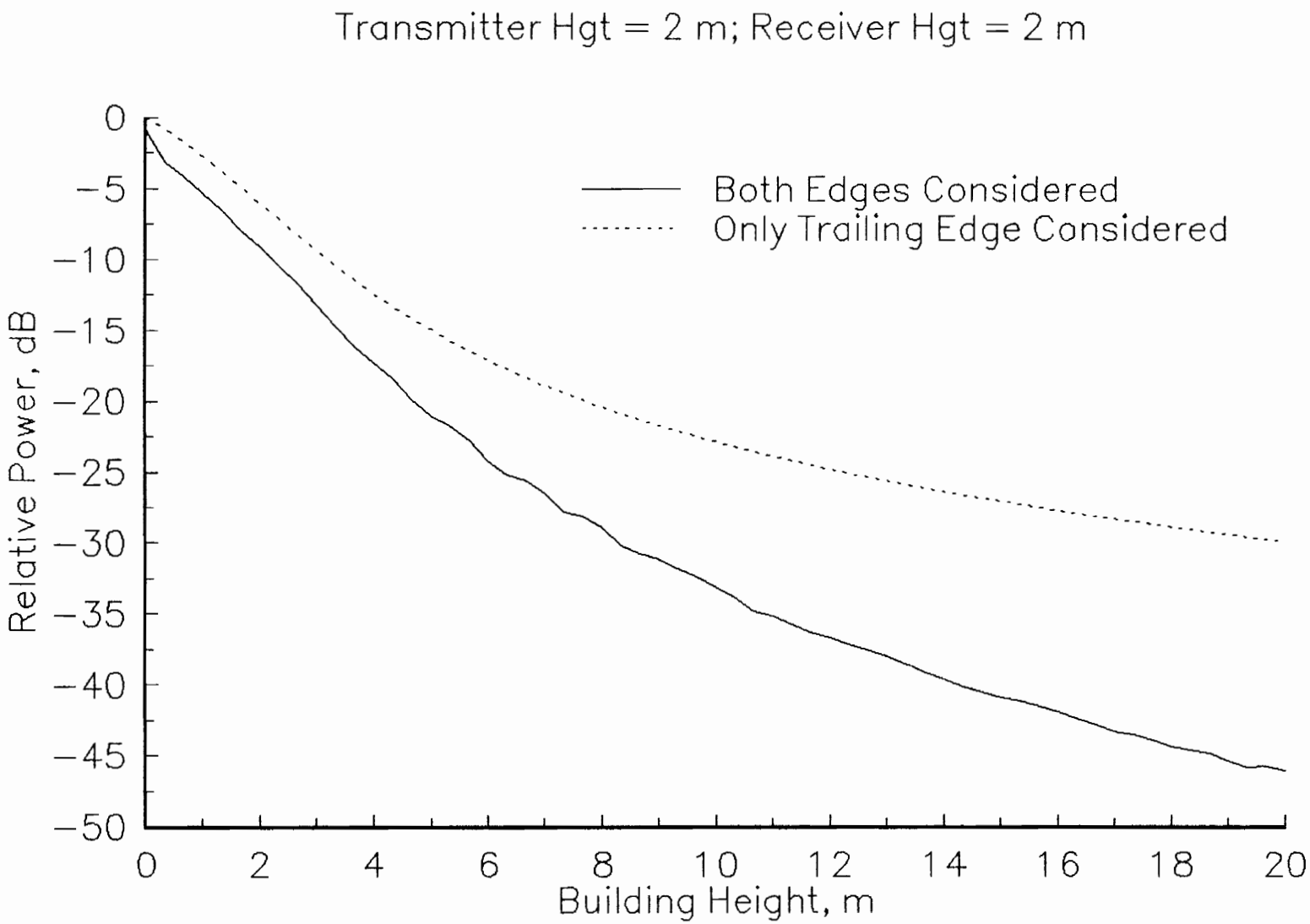


Figure 5.6. Multiple Diffraction, Dependence on Building Height

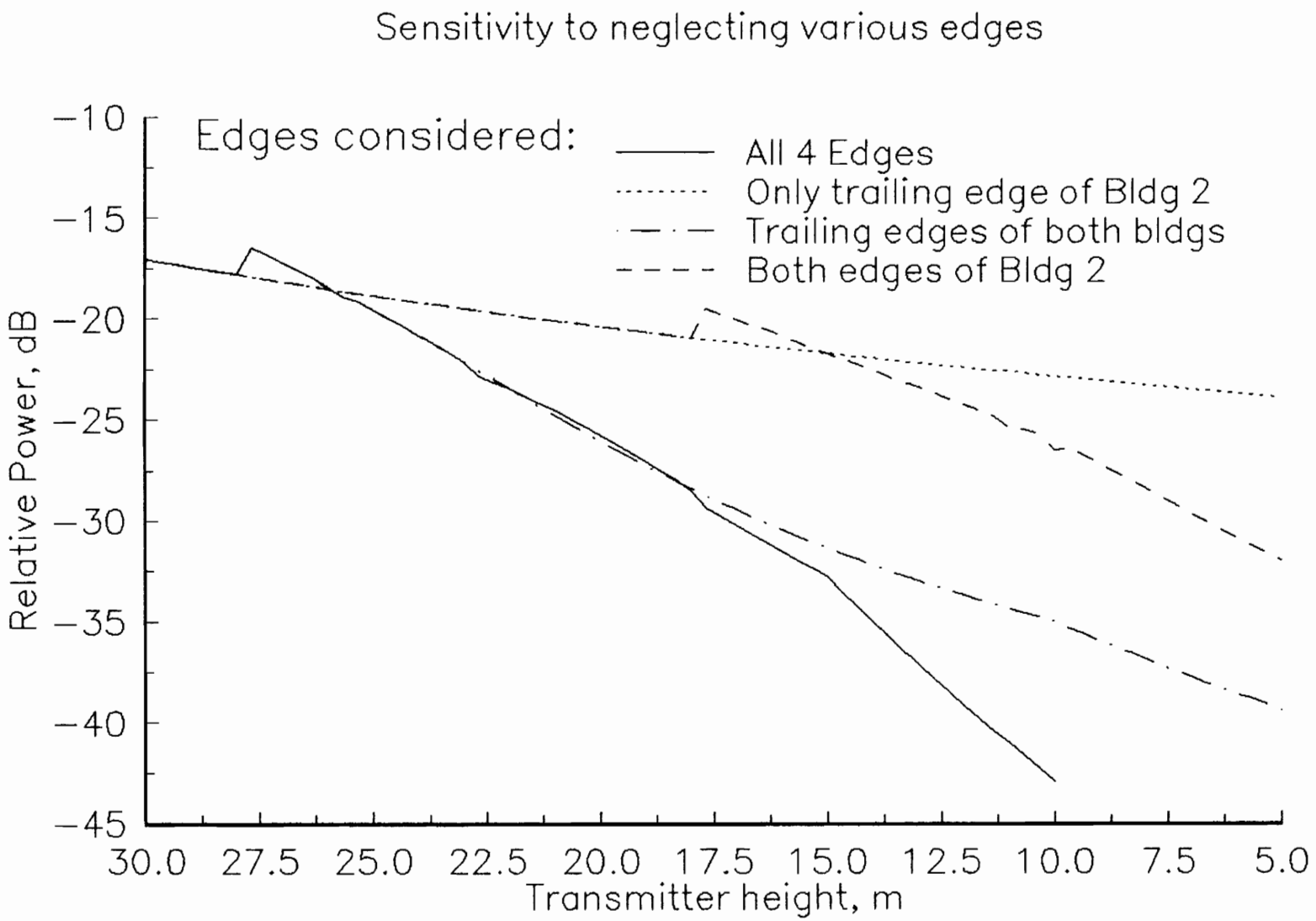


Figure 5.7. Multiple Diffraction by Two Buildings

neglecting various edges. For example, if only the last edge before the receiver (the trailing edge of Building 2) is considered, a technique often used in predicting approximate field strength, the error is on the order of 20 dB.

The discontinuities seen in some of the curves of Figure 5.7 are the result of our modeling assumption that edges providing clearance of at least 55% of the first Fresnel zone can be neglected. Referring to the curves in Figure 5.4, where the 55% R_1 point can be identified as the point where each curve first crosses 0 dB (free space loss), we see that as clearance increases past this point, the power continues to increase and then to oscillate about 0 dB. Apparently in the multiple diffraction situation, where the field is integrated over a vertical region at the trailing edge, the field in the shadow of the second edge encountered by the wave can be near a maximum of the oscillations at the point where the first edge has a 0.55 R_1 clearance. The sudden consideration of the first edge can then cause a sharp increase in power. This effect is confirmed in Figure 5.8, in which the solid line reflects our current modeling assumption regarding clearance, the dashed line the forced neglect of the leading edge, and the dotted line the consideration of both edges at all times, regardless of clearance. The dotted line, a portion of which is hidden by the solid line, represents the true situation where both edges affect the signal level even when there is substantial clearance, and we can see the oscillatory effect clearly. The above modeling assumption in this case causes an error of up to 5 dB. The case considered in Figure 5.8 includes only Building 1, which has a width of 20 m, much less than

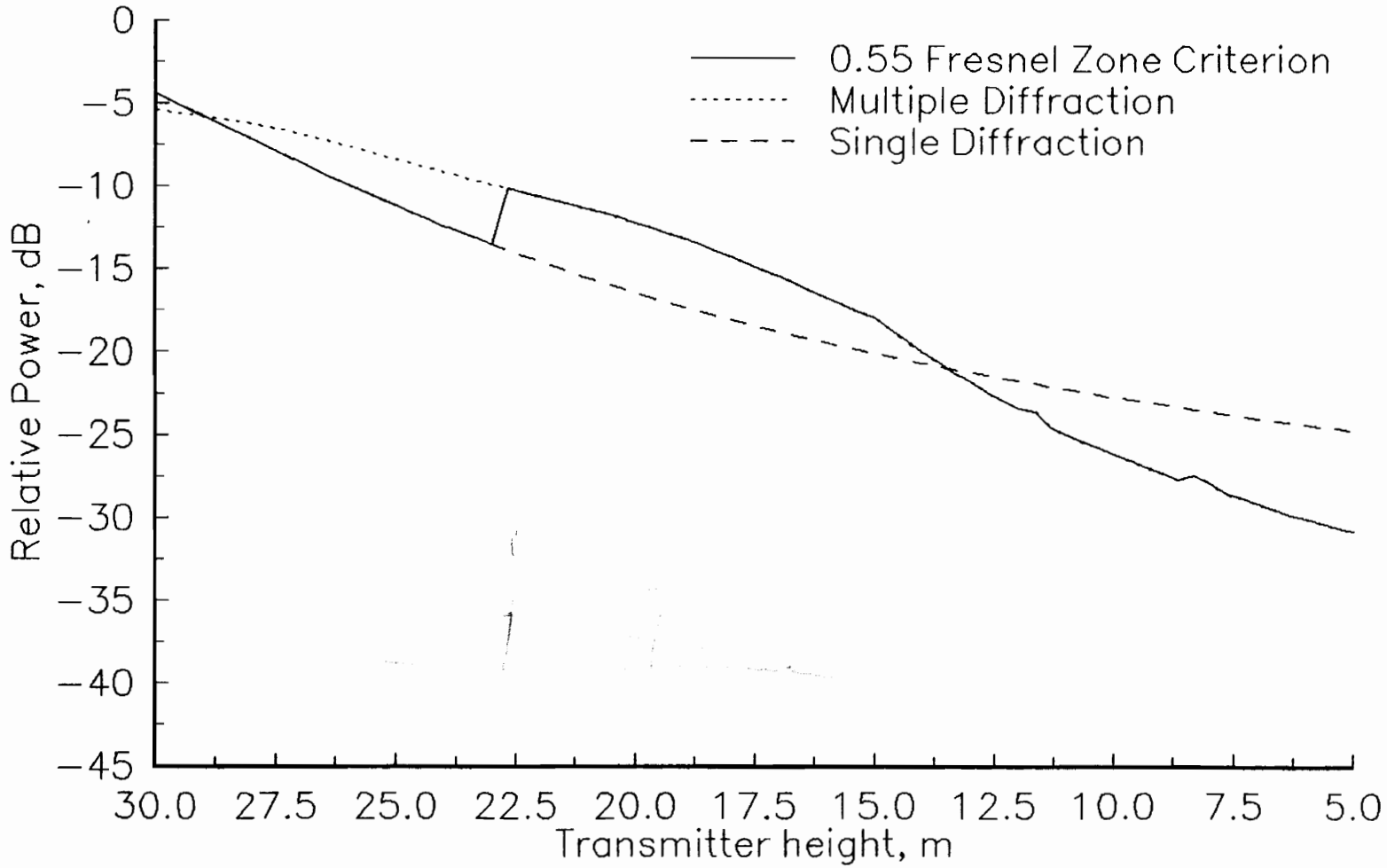


Figure 5.8. Effect of Criterion for Neglecting Edges

the 50 m width of the second building, which is examined in isolation in two of the curves of Figure 5.7. The greater discontinuity seen in the case of Figure 5.8 (5 dB compared with 2 dB) may be related to the smaller distance between the two edges of the first building. In general, decreasing the distance between diffracting edges enhances the interaction in their diffractive effects.

5.2 Three Dimensional Multiple Diffraction Model

The multiple diffraction model is applied to diffraction past several edges of a single obstacle in a manner similar to the three dimensional single diffraction model, with some differences noted here. At each observation point, whether it be the receiver point, P, or an intermediate point, P′, the diffraction is computed for each edge that diffracts energy directly to P (or P′) without encountering further diffraction. The apertures constructed for each of these edges are translated segments of a total aperture plane, as described in Section 4.3.5. The major difference is that in the multiple diffraction model, the field is found explicitly at sample points in the apertures to allow consideration of prior diffracting edges, those whose apertures are not part of the same aperture plane. Another difference is that, while at the receiver point, P, the field contributions are summed in either a complex phasor sense or as a power sum, at intermediate points, P′, the fields are compared and only the largest field source is used to find the diffraction at the next point. This is done in order to avoid the possibly large interference effects between the fields in the aperture and the consequent difficulties in integrating over the phase and amplitude variations. The power

sum method used to approximate the mean field strength at the receiver can not be used at intermediate points because it does not predict phase.

5.2.1 Representation of Real Buildings

We consider relations between buildings with two classifications: isolated and connected. If two buildings do not share an edge, they are assumed to be isolated, meaning that if any parts of the two buildings are behind each other the buildings will be treated as multiple successive diffractors, and if they are next to each other and separated, only the building explicitly blocking the signal will be considered. Thus the diffraction field contributions from edges on separate isolated buildings are never summed together, and each building can be considered individually in the recursive multiple diffraction procedure. (Connected buildings are treated as a single building with different heights. This special case will be discussed in Chapter 7.)

We now consider the case of a single isolated building, with the intent of applying these principles to multiple buildings individually. These models require the idealization of buildings as rectangular volumes, where each face is either horizontal or vertical and all angles are right angles. The most general model of such a building has eight possible diffracting edges: four roof edges and four corners. All of these are considered. The techniques described in Section 4.3 are applied to determine the plane of integration and limits of integration for each edge, given source and observation points. The results of this application in

a general case where both the transmitter and receiver are low enough relative to the roof height that the roof edges introduce multiple diffraction are illustrated in Figures 5.9 (roof edges) and 5.10 (corners), where one of the two dimensions of each aperture is indicated with a bold line, and the other dimension is directed out of the page.

In Figure 5.9 it is clear that the four roof edges define apertures that are translated segments of two separate aperture planes, producing double diffraction by two edges at each plane. This is the general roof-edge model used in our computer program. For each building, the four roof edges are grouped into two diffraction levels, where edges that belong to the same level define apertures that derive from a single, albeit segmented and translated, aperture plane, and the levels are numbered according to the order in which they affect the wave. For example, in Figure 5.9, the two apertures at the top of the building are part of the first aperture plane, so the two corresponding roof edges are considered level 1. The maximum and minimum bounds on the transverse building dimensions are used to distinguish the two levels in the following way. Each edge is defined by two corners, one of which must be one of the two bounding corners. Edges that extend in the direction of S from one of the bounding corners are part of diffraction level 1, while those that extend towards P cause the second diffraction, and are part of level 2.

Assuming the building under consideration is the nearest to the receiver,

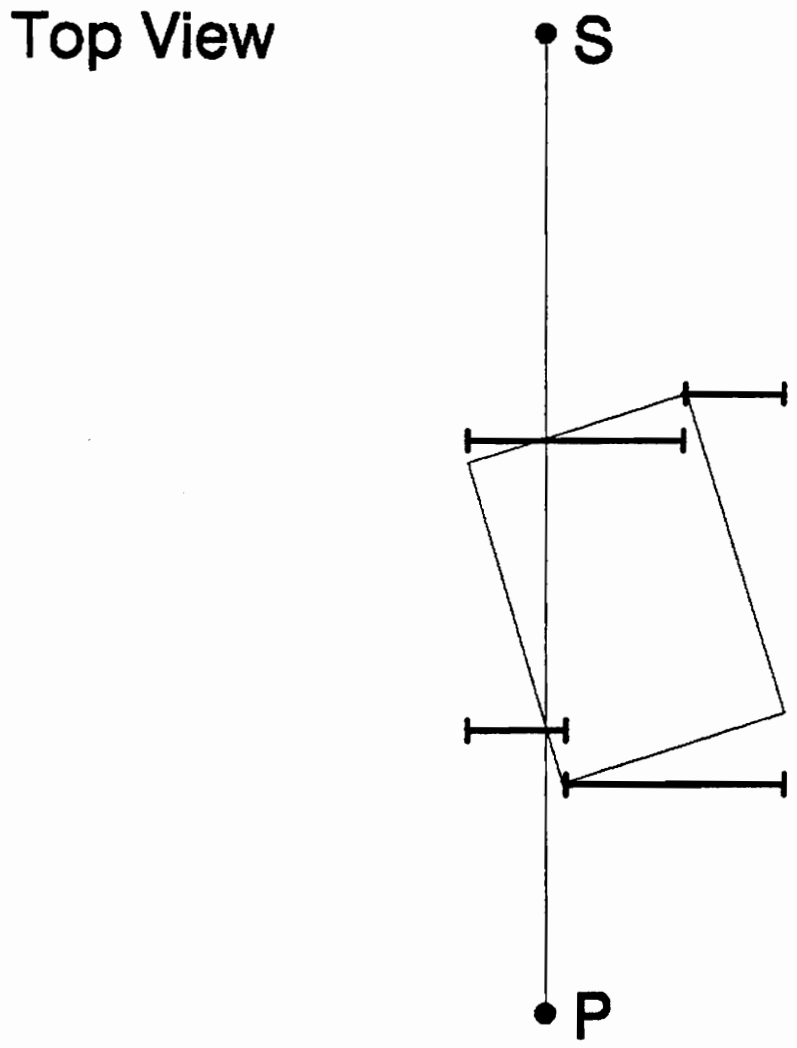


Figure 5.9. Multiple Rooftop Diffraction Model

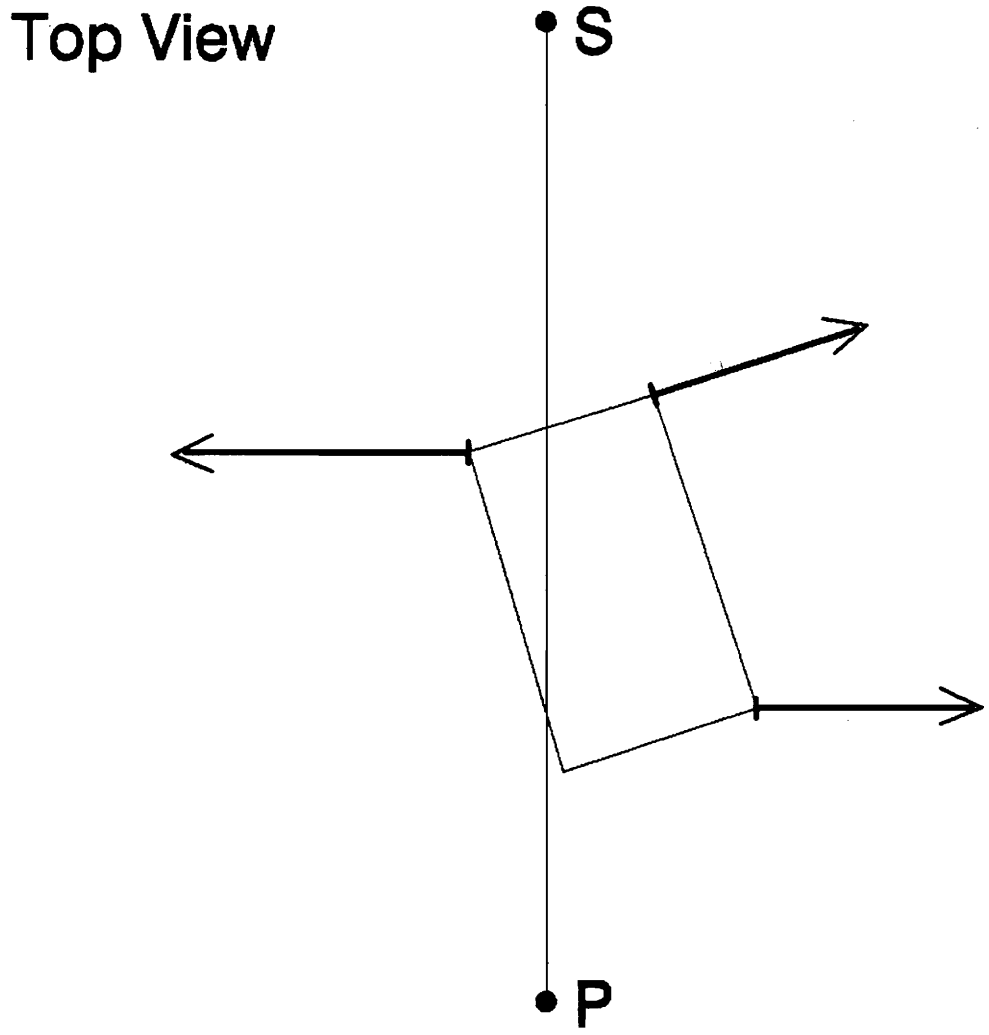


Figure 5.10. Multiple Corner Diffraction Model

the field contributions from the two edges in level 2 are summed together. The field distribution in the aperture at each of the level 2 edges is computed individually by finding the diffraction by the two level 1 edges, and then neglecting the level 1 edge which contributes the smaller field amplitude. The field in the apertures at the level 1 edges in this case are not affected by prior obstacles, but in general when the program reaches a level 1 edge, it returns to the top and searches the database again for another diffracting building.

The diffraction fields contributed by the building corners, while summed with the rooftop fields to get the total building diffraction field (or compared with the other fields to identify the dominant component if the building under consideration is not the nearest to the receiver), are considered separately from the rooftop fields with regard to ordering the diffraction levels. Figure 5.10 illustrates an example. In this general case, where SP passes through the horizontal projection of the building as opposed to next to the building, the corners are grouped according to whether they are on the left of SP or the right, and then ordered within each group. For each building we compute two diffraction fields, one around the left of the building and one around the right. Multiple corners in a single group are considered as possible multiple diffractors. Of course, in many cases, not every corner truly diffracts the signal significantly, so they are all checked first for clearance. The elimination of these corners is discussed in Chapter 7. We note in Figure 5.10 only one diffracting corner on the left, and two on the right (there is sufficient clearance at the corner on the

right that is nearest the receiver). Also, we note that the aperture plane next to the first corner on the right is at a different angle. This arises through the procedure whereby we sample the field in the aperture of the diffracting, right-side corner nearest the receiver by considering sample points as new observation points, P′, and drawing a line to S. The aperture plane at the preceding diffracting corner is then defined relative to this new line, SP′.

5.2.2 Example Calculations

The implementation of these algorithms is demonstrated with an example. In Figure 5.11 is shown a building obstacle taller than both the transmitter and receiver. The transmitter, which may be thought of as microcellular base station antenna on a lamppost, is at 10 m. The receiver is 2 m high and is a model of mobile unit, traveling on a 150 m track past the building. The transmitter and receiver can be considered to be on perpendicular streets, and the receiver begins its journey at the street corner, in sight of the transmitter. At the end of the track the receiver is again in sight of the transmitter. The individual field components computed at the receiver are plotted in Figure 5.12. In Figure 5.13 is plotted the phasor sum, showing the diffraction multipath effects, and the power sum, an estimation of the mean field strength.

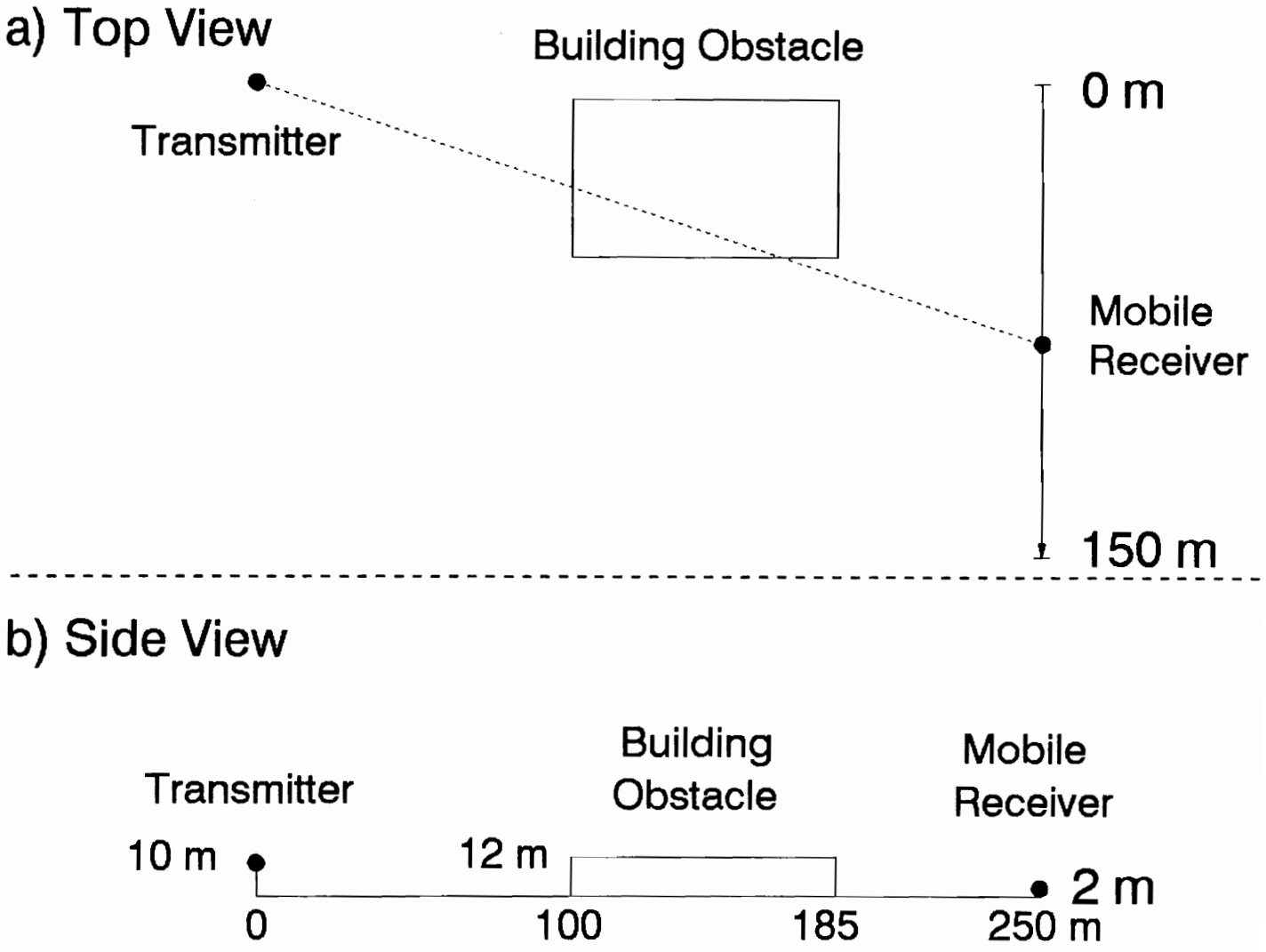


Figure 5.11. Three-Dimensional Multiple Diffraction Example

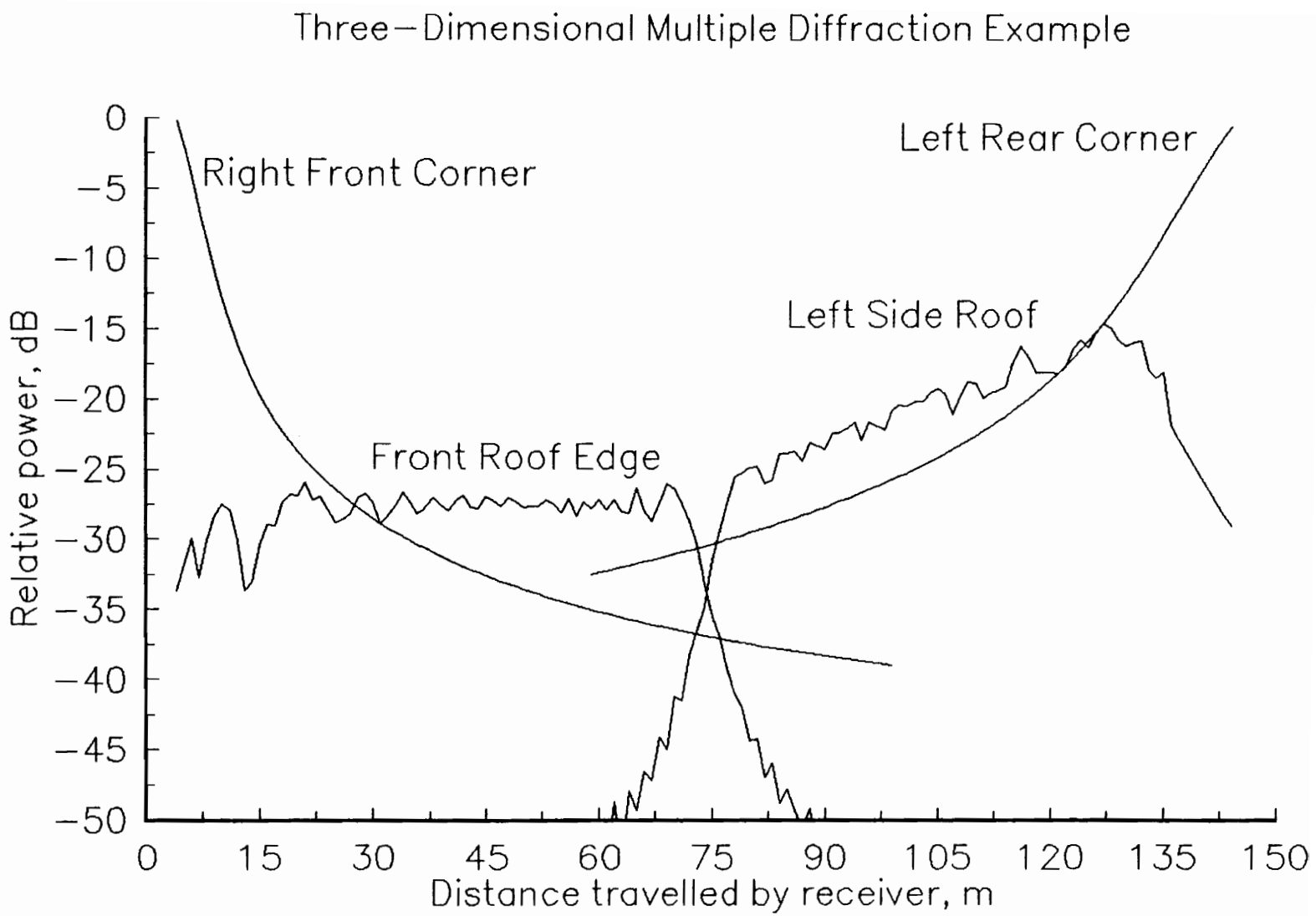


Figure 5.12. Individual Field Components, Multiple Diffraction

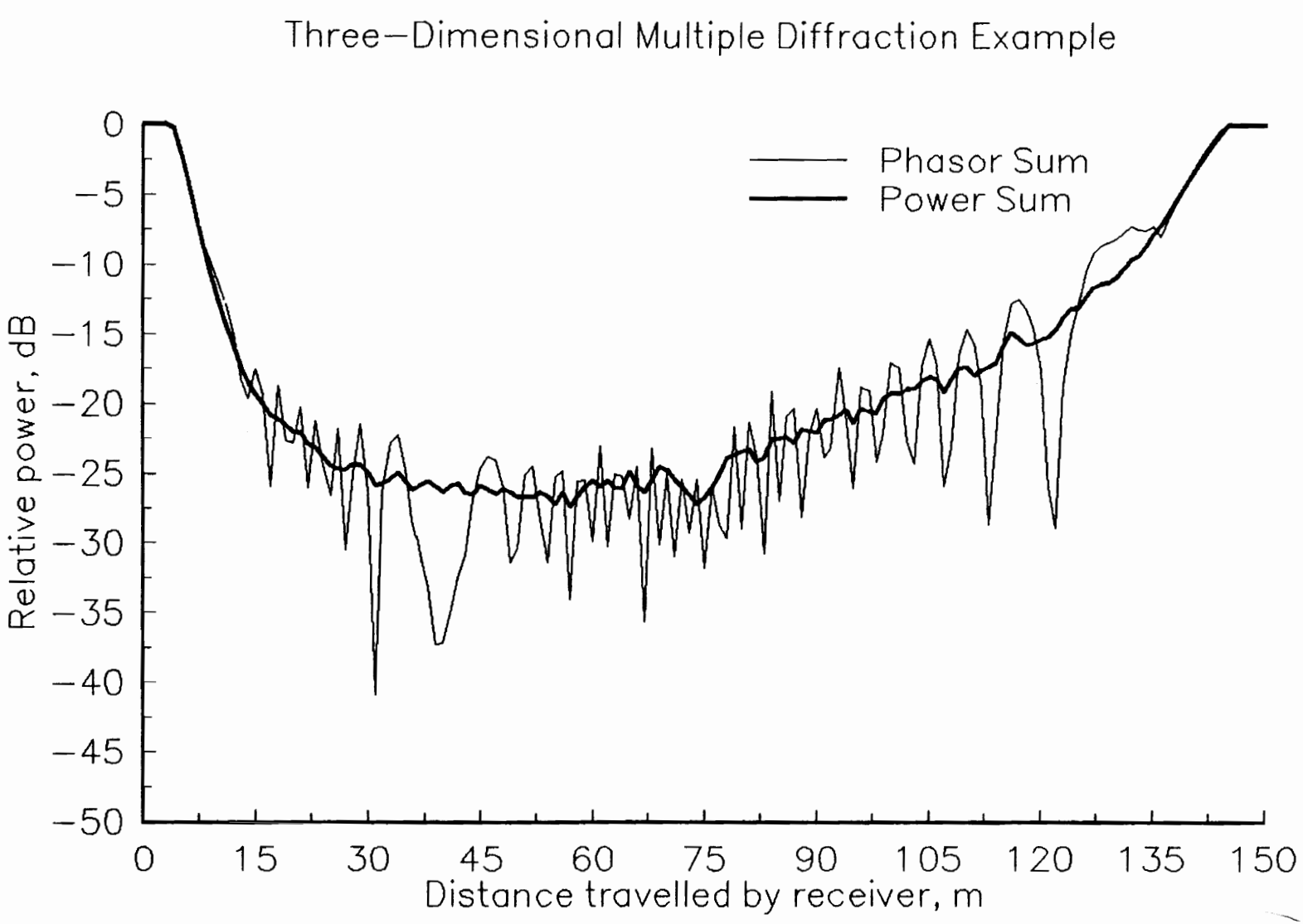


Figure 5.13. Total Multiple Diffraction Field

5.3 Comparisons with Measurements

5.3.1 Measurement Track SL32

Referring back to Figure 4.17, it is evident that as the receiver travels along measurement track SL32, the radio path past the intervening buildings varies considerably, with many areas experiencing multiple field contributions from front and side roof edges, and from corner edges. From the side view it is also evident that the received field has been diffracted by multiple successive buildings for much of the measurement track. Track SL32 thus provides a good test for the three dimensional multiple diffraction model. Comparisons with track SL32 were prepared with an early version of the software which, while implementing the same diffraction models as the final version, did not use general algorithms to automatically determine the relevant diffracting edges from the building database and to order them according to successive diffractions. As the radio paths are very complicated in some of these cases, some experimentation was possible in order to find a good fit to the data while still using reasonable applications of the models. The comparisons with track SL32 therefore provide an interesting demonstration of the application of Fresnel-Kirchoff theory to buildings, but not a real test of all the methods described in this thesis.

Figure 5.14 presents the signal strengths of the predicted diffraction field components. The complex sum of these components yields the total signal

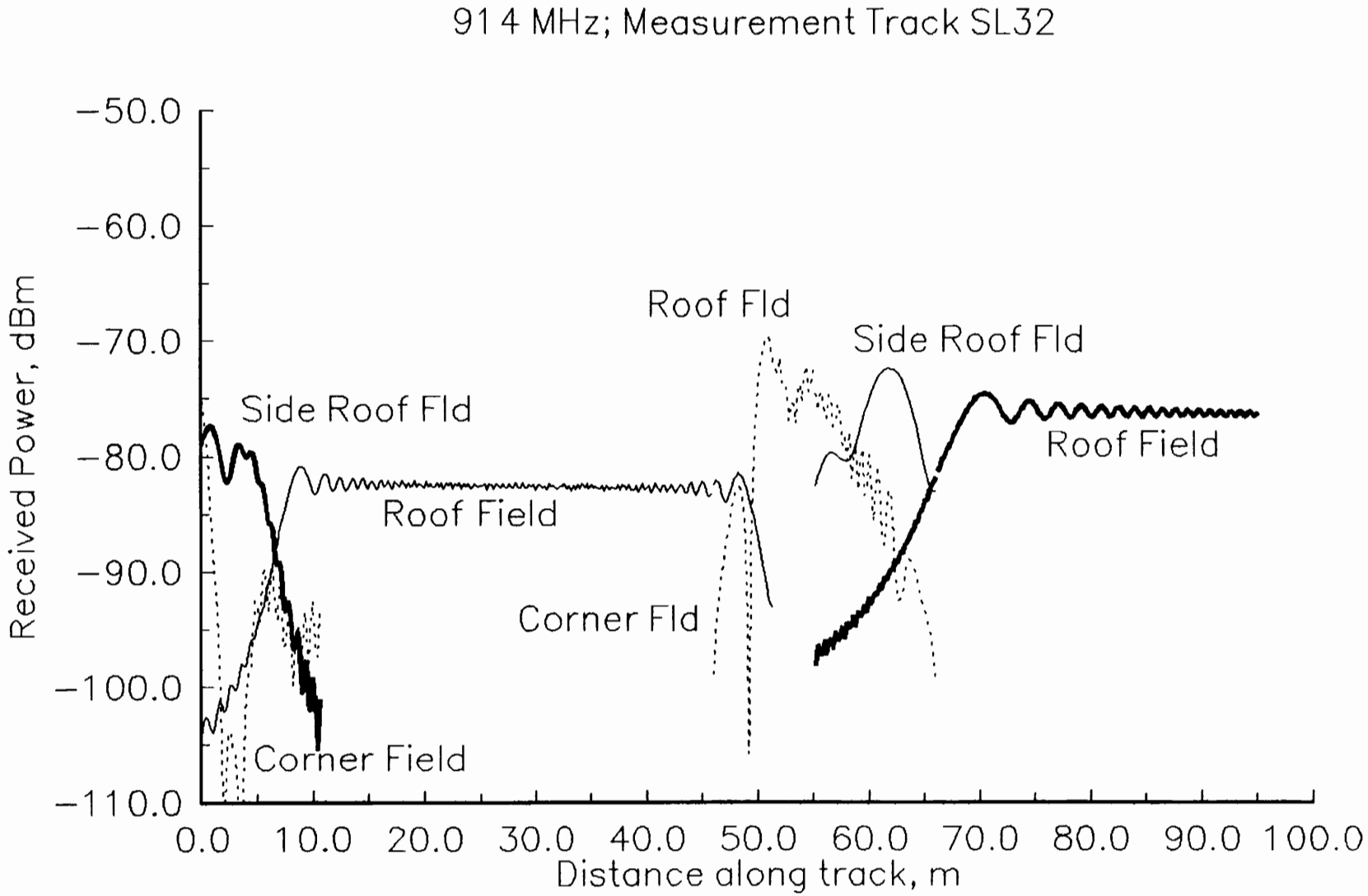


Figure 5.14. Predictions of Individual Field Components

strength predictions, which are presented with the measured data in Figure 5.15. As in the comparison with track SL22, few of the peaks and valleys of the predictions line up with those in the measured data. However, particularly in the regions 5m-8m and 48m-62m, the scale of signal variations is comparable, suggesting that in those regions much of the multipath interference may be due to multiple diffraction paths.

The measured data from track SL32 were filtered with a Hamming window of length 5λ to obtain an estimate of the mean received signal strength. This result is compared with the model predictions in Figure 5.16. Neglecting some constructive and destructive interference in the predictions, the two curves match to within 5 dB for almost all of the track length. Each of the major characteristics of the filtered measurements is predicted well: the downward slope as the receiver travels from 0m to 8m and passes the corner of the building; the two fairly stable regions: 15m-45m and 70m-90m; and the 12 dB jump in power as the receiver passes between buildings at 50m-55m. These results lend confidence in the application of Fresnel-Kirchoff diffraction theory to the microcellular environment, and to a greater degree they illustrate the possibility of gaining knowledge of characteristics such as abrupt changes in average signal levels, based solely on three-dimensional diffraction prediction models applied to a building database.

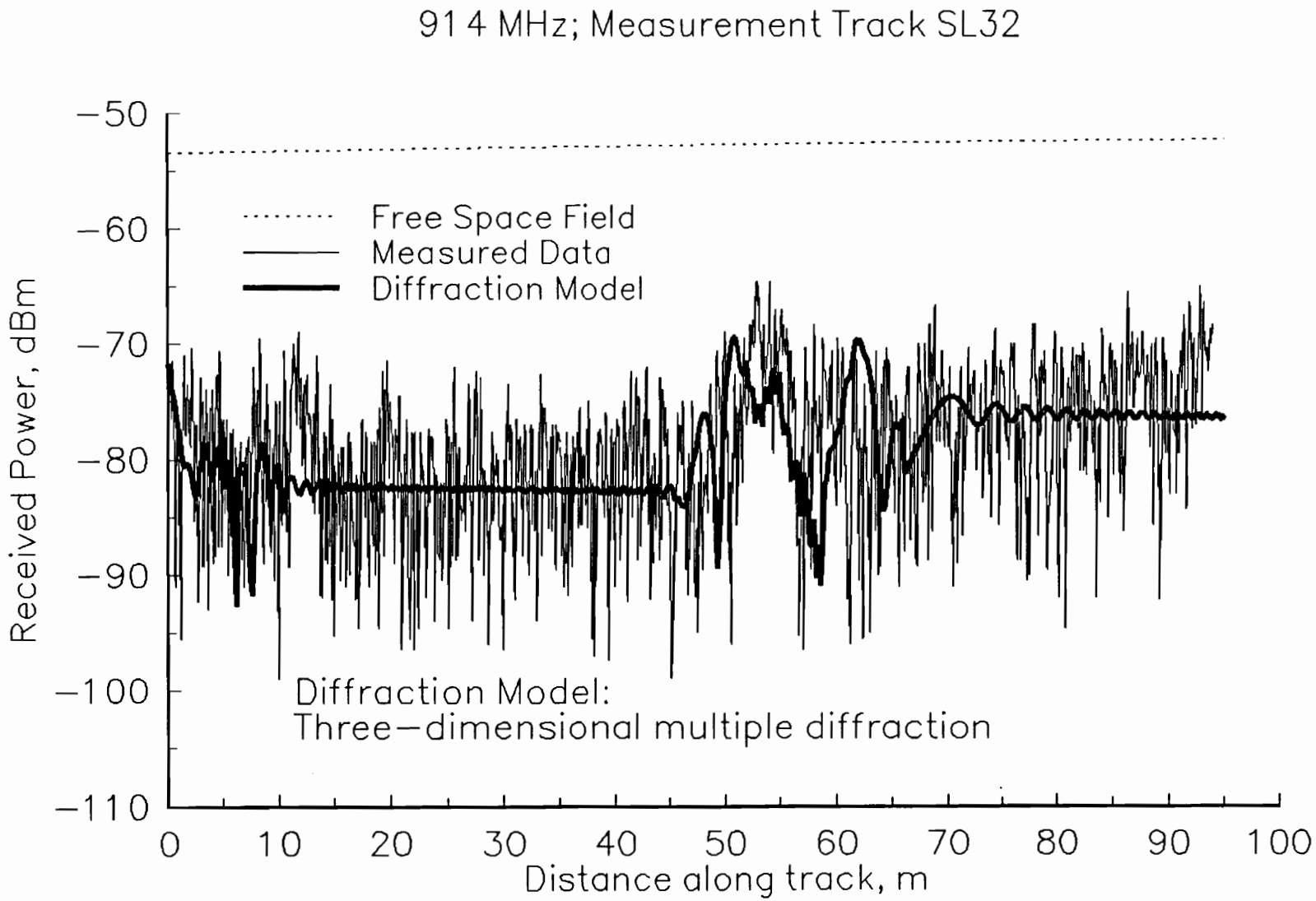


Figure 5.15. Predictions Compared with Unfiltered Measurements

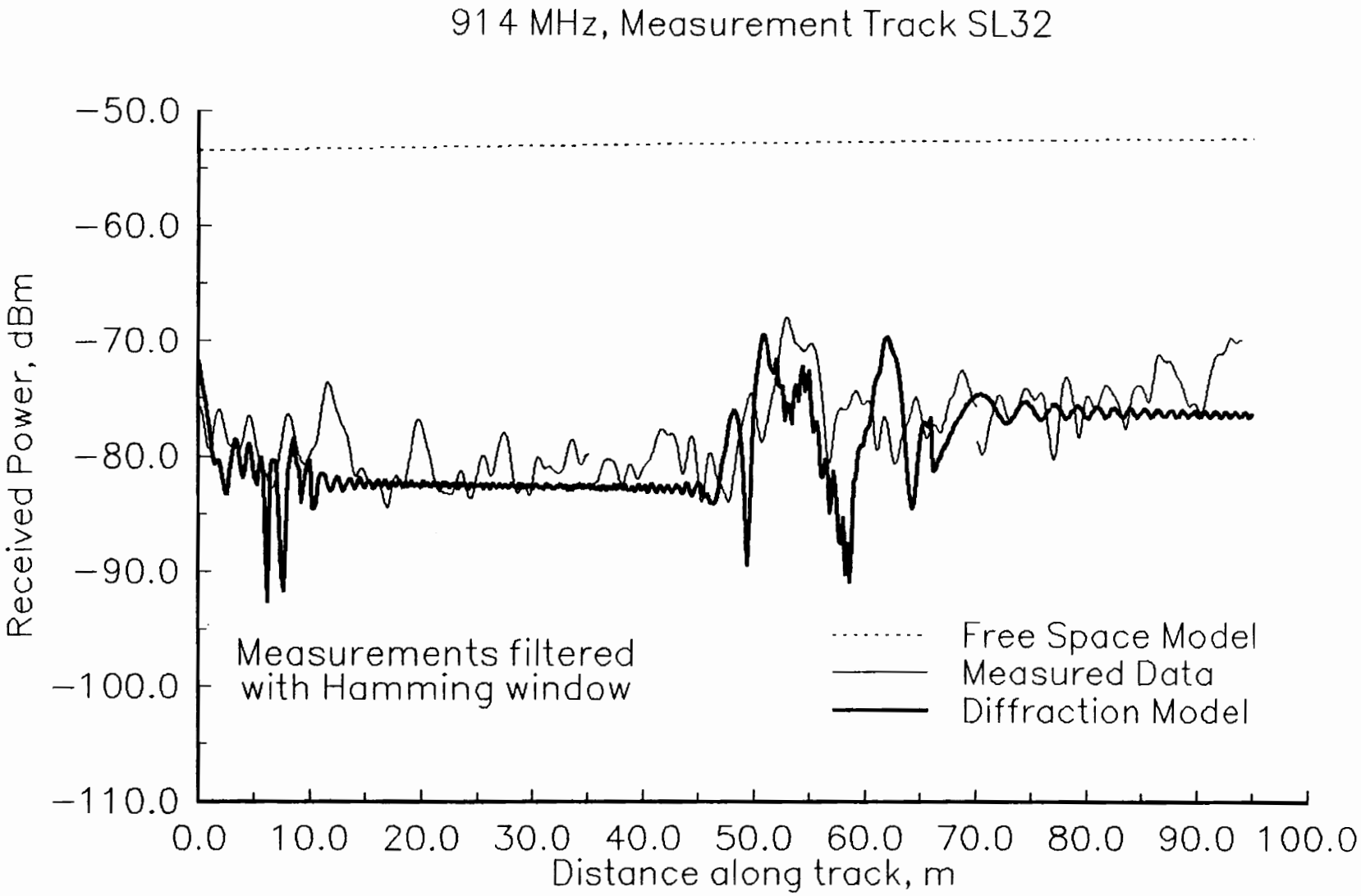


Figure 5.16. Predictions Compared with Filtered Measurements

5.3.2 Patton Hall

The full version of the software was tested against measurements taken near Patton Hall and Davidson Hall. Only the building coordinates and the locations of the transmitter and receiver, along with the frequency, 914 MHz, were input to the program, and no adjustments of parameters were done, making these "blind" tests of the models. Both of the measurement runs were chosen to isolate a single building and the two sets of data clearly show the strong impact of the individual buildings on signal strength.

The geometry of the Patton Hall measurements is provided in Figure 5.17. The transmitter was placed on top of another building, providing the case of propagation from above the obstacle. This measurement area and the following one near Davidson Hall were specifically chosen for the absence of sources of strong specular reflections. The measured data is presented in Figure 5.18 in both its raw form and after filtering with a 20λ Hamming window, which provides an estimate of the local mean field strength [3]. The predicted diffracted field is plotted in Figure 5.19 as a phasor sum and as a power sum, where the latter gives an approximate prediction of the local mean field strength according to Ikegami et al. [4]. The phasor sum demonstrates multipath interference between the diffraction field components. A comparison of the estimates of the measured and predicted mean field strengths is presented in Figure 5.20. The limitations of a diffraction prediction tool are clear from this comparison. The onset of signal loss as the receiver passes behind the building is

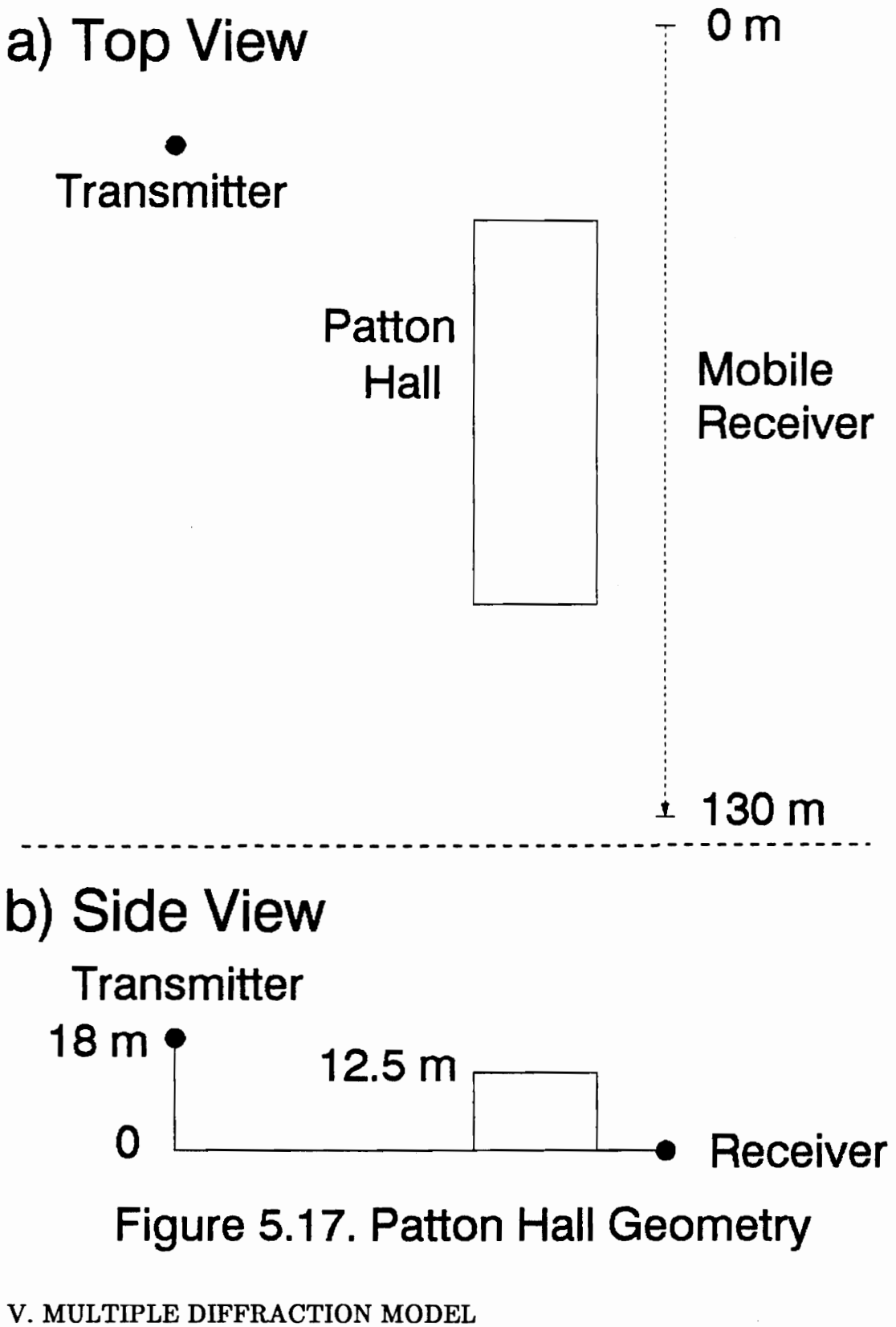


Figure 5.17. Patton Hall Geometry

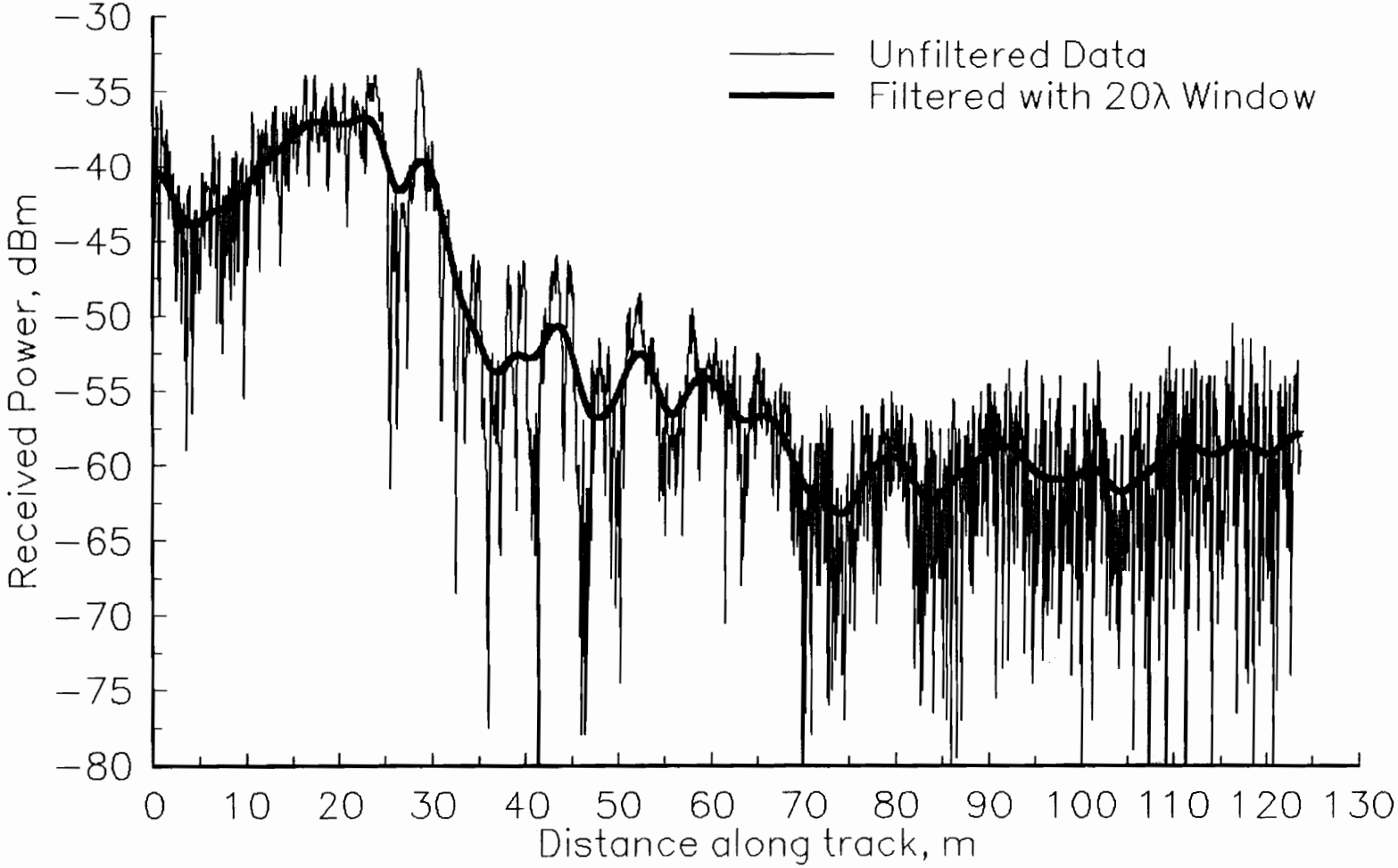


Figure 5.18. Patton Hall Measured Data

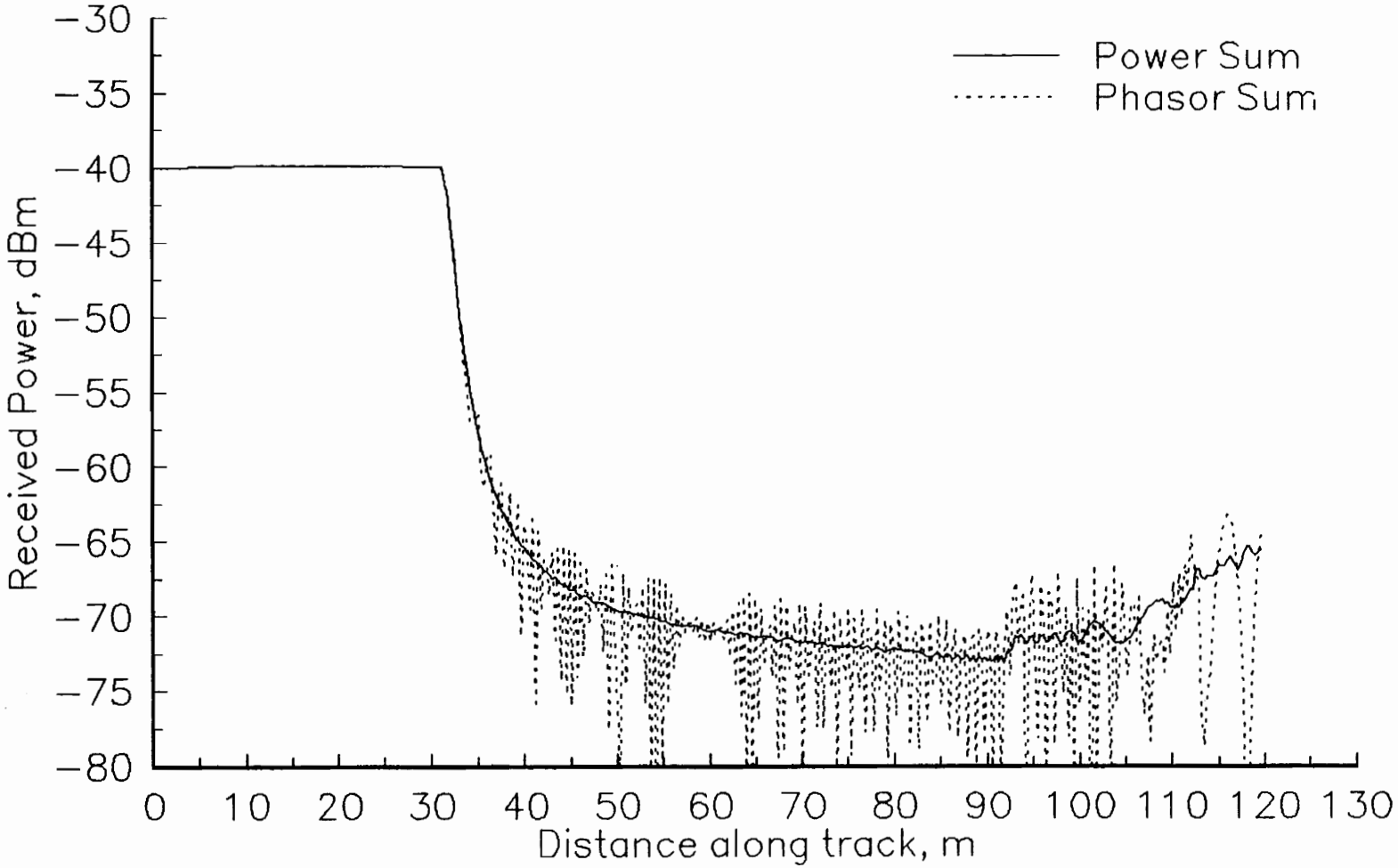


Figure 5.19. Patton Hall Predictions

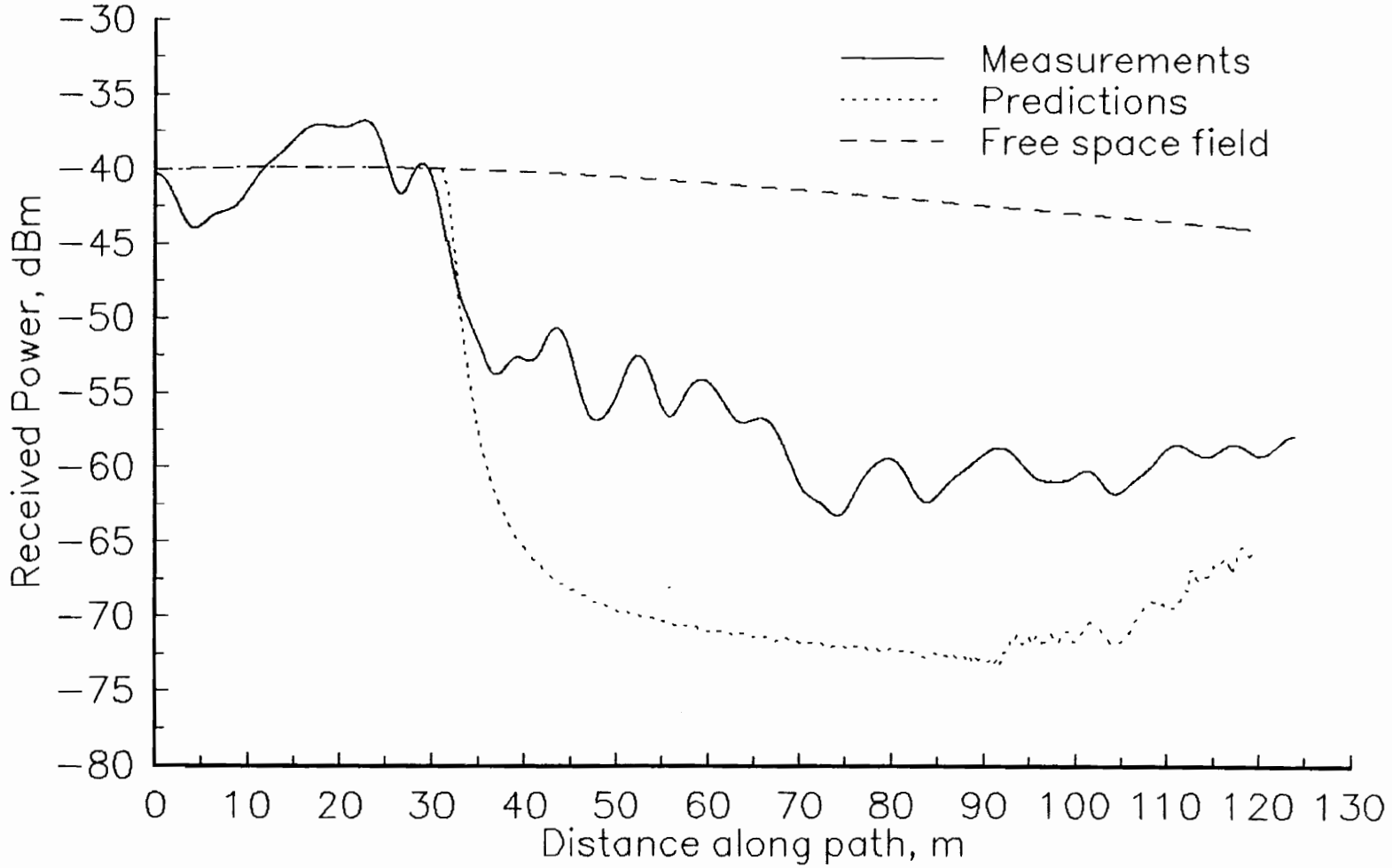


Figure 5.20. Patton Hall: Predictions Compared with Measured Data

well predicted, but when the receiver is in a deep shadow, the predictions are 10-15 dB pessimistic. A similar result was reported by Rappaport and McGillem [32] on tests in the indoor factory environment. The explanation in that case appears reasonable here too: that signal is arriving by alternate paths through scattering or reflections by other elements of the environment. In this case there are two large trees situated in an ideal spot directly off the top right corner of the building, as viewed in the top view of Figure 5.17, such that field diffracting around that corner will be enhanced by scattering from tree trunks. It should be noted that the trees do not have any leaves until a great height, near the corner of the building they are simply two vertical trunks about a foot and a half in diameter, separated by the same distance. Because the signal is vertically polarized, the trees appear as though they could function as excellent sources for redirection of the signal, albeit with significant loss. Note also that the predicted diffraction field strength increases towards the end of the path. This is due to the contribution from around the far side of the building becoming stronger as the receiver nears that end of the building.

Figure 5.20 shows that the mean of the measured signal never falls more than approximately 20 dB below the free space level, whereas the diffraction field falls up to 35 dB below free space. This can be explained satisfactorily if it can be assumed the trees scatter energy with a 20 dB attenuation. Unfortunately this does not allow confirmation of the roof edge diffraction models.

5.3.3 Davidson Hall

Davidson Hall has sections with three different heights, as seen in Figure 5.21. This set of measurements, presented in Figure 5.22, provides a test of the algorithms for modeling a complex building. Moreover, these measurements simulate a typical situation in the proposed microcellular system architecture. The transmitter was placed 15 feet high on the sidewalk, similar to the proposed lamp-post placement of low-power base stations, and the receiving antenna was 6 feet high on a mobile cart, simulating a pedestrian with a personal communication device. Figure 5.21 shows that the receiver begins with a line of sight of the transmitter, and as it travels down the sidewalk, it becomes increasingly shadowed by the building.

Figure 5.23 displays the phasor sum and power sum of the predicted diffraction field components. The small fluctuations in the phasor sum indicate a strongly dominant component, which in this case is the corner diffraction field. Only in the deep shadow near the end of the track does the multiply-diffracted rooftop field begin to approach the level of the singly-diffracted corner field. The comparisons of predicted versus measured local mean field strength are plotted in Figure 5.24, with a result similar to that of the Patton Hall measurements. The slow drop in the signal level at 10-25 m on the track matches very well, with only a 2-3 m offset that may be due to imprecision in locating the various elements. Again, though, as the predicted diffraction field falls below a certain threshold, the measured signal power levels off at a higher power than the

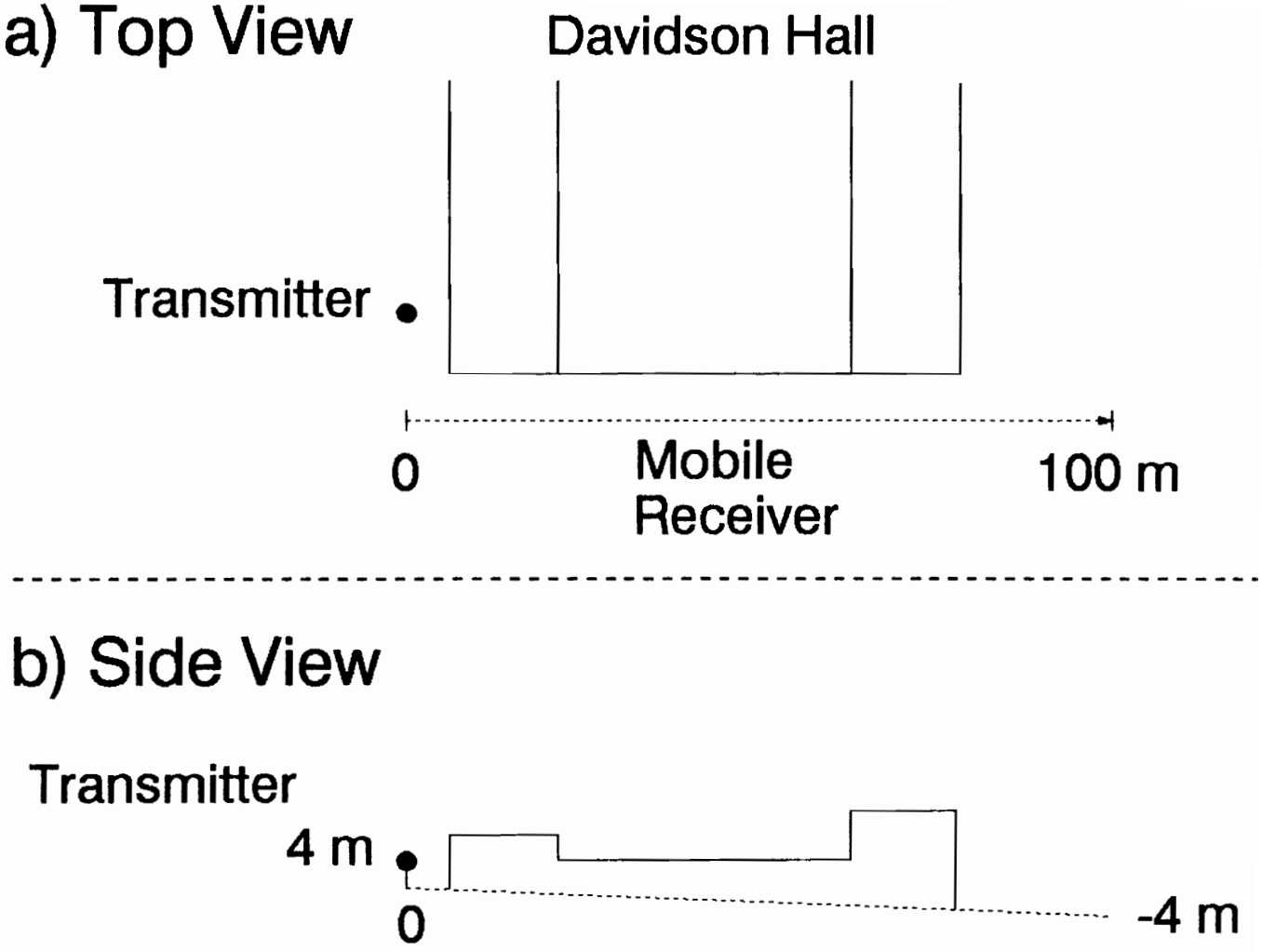


Figure 5.21. Davidson Hall Geometry

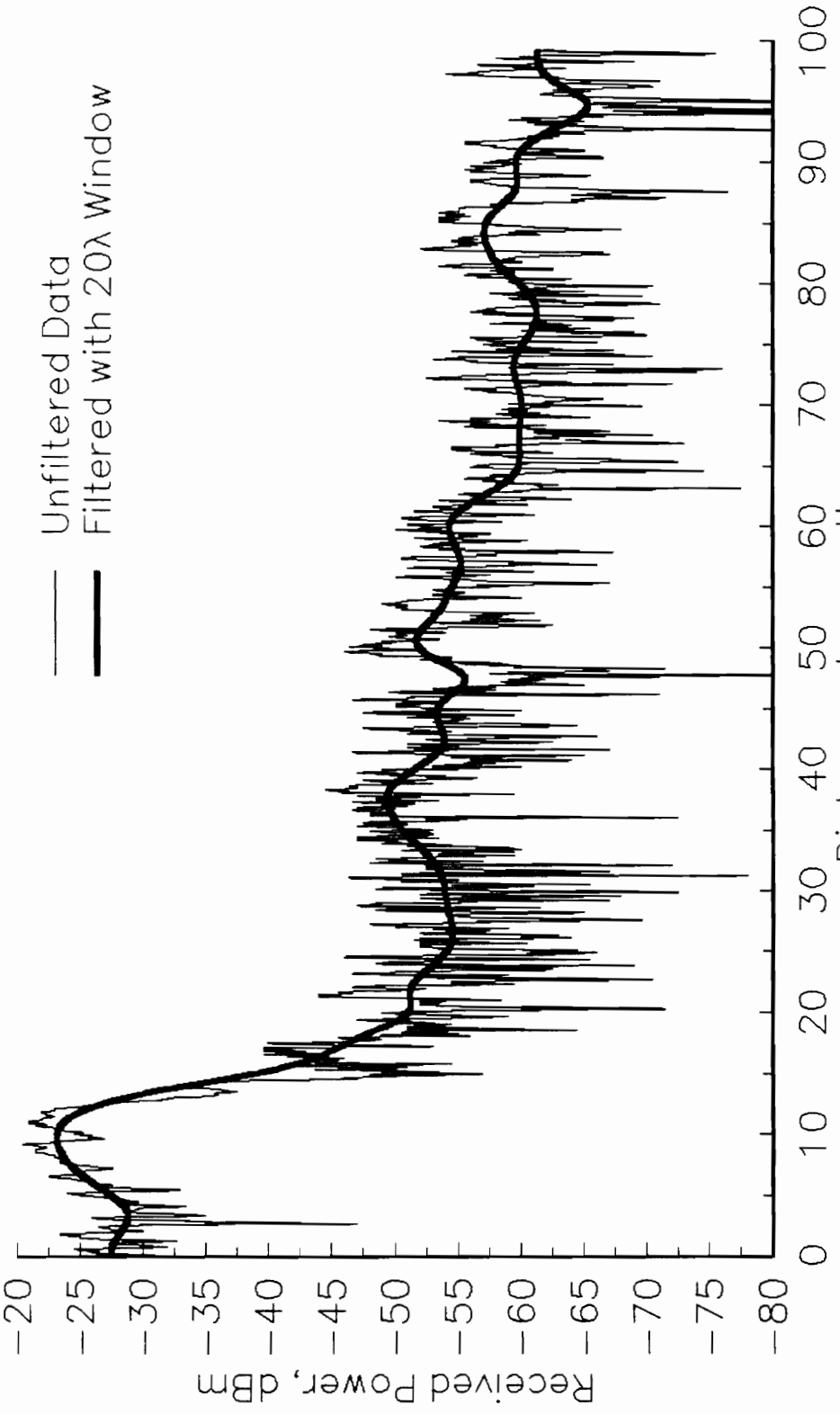


Figure 5.22. Davidson Hall Measured Data

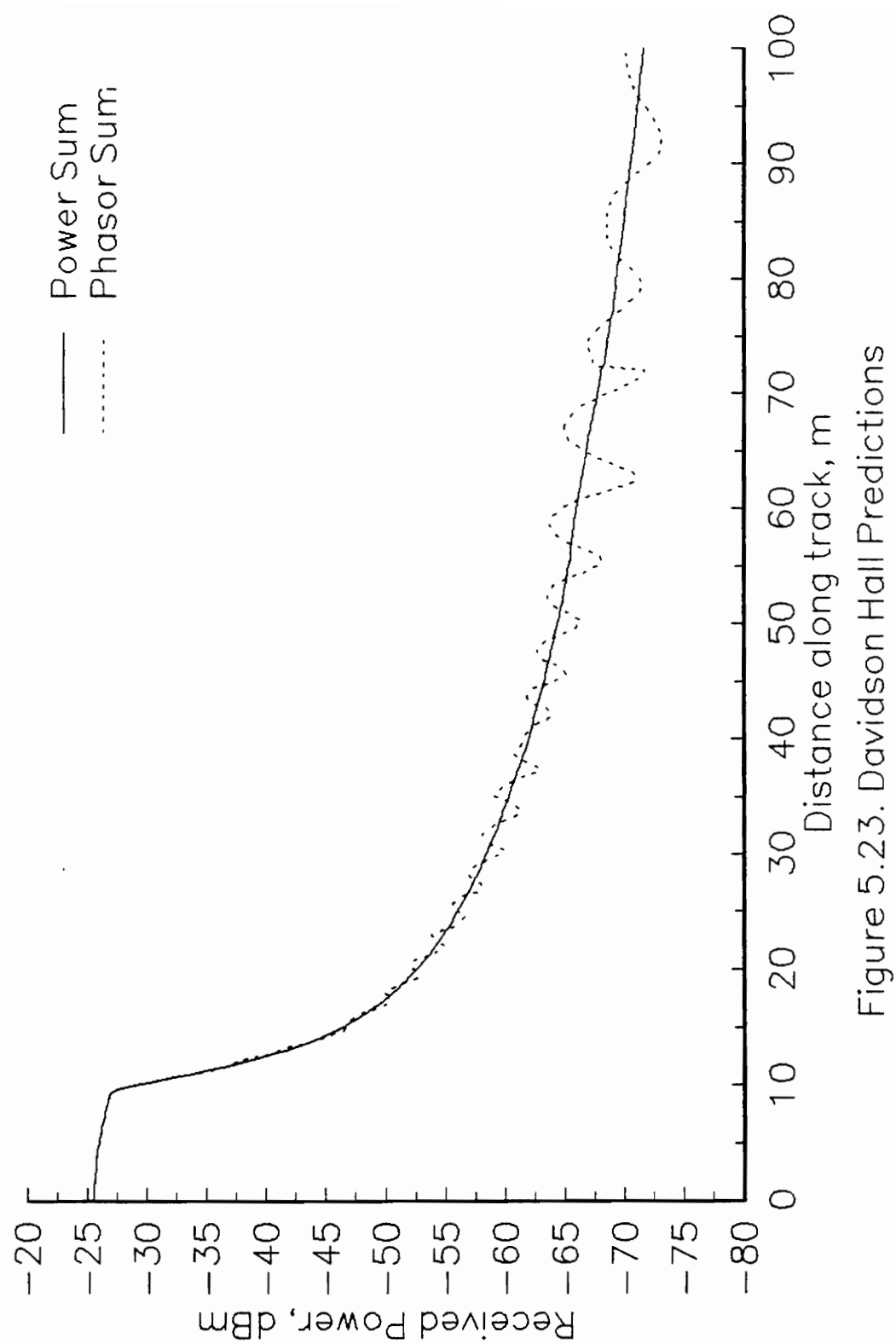


Figure 5.23. Davidson Hall Predictions

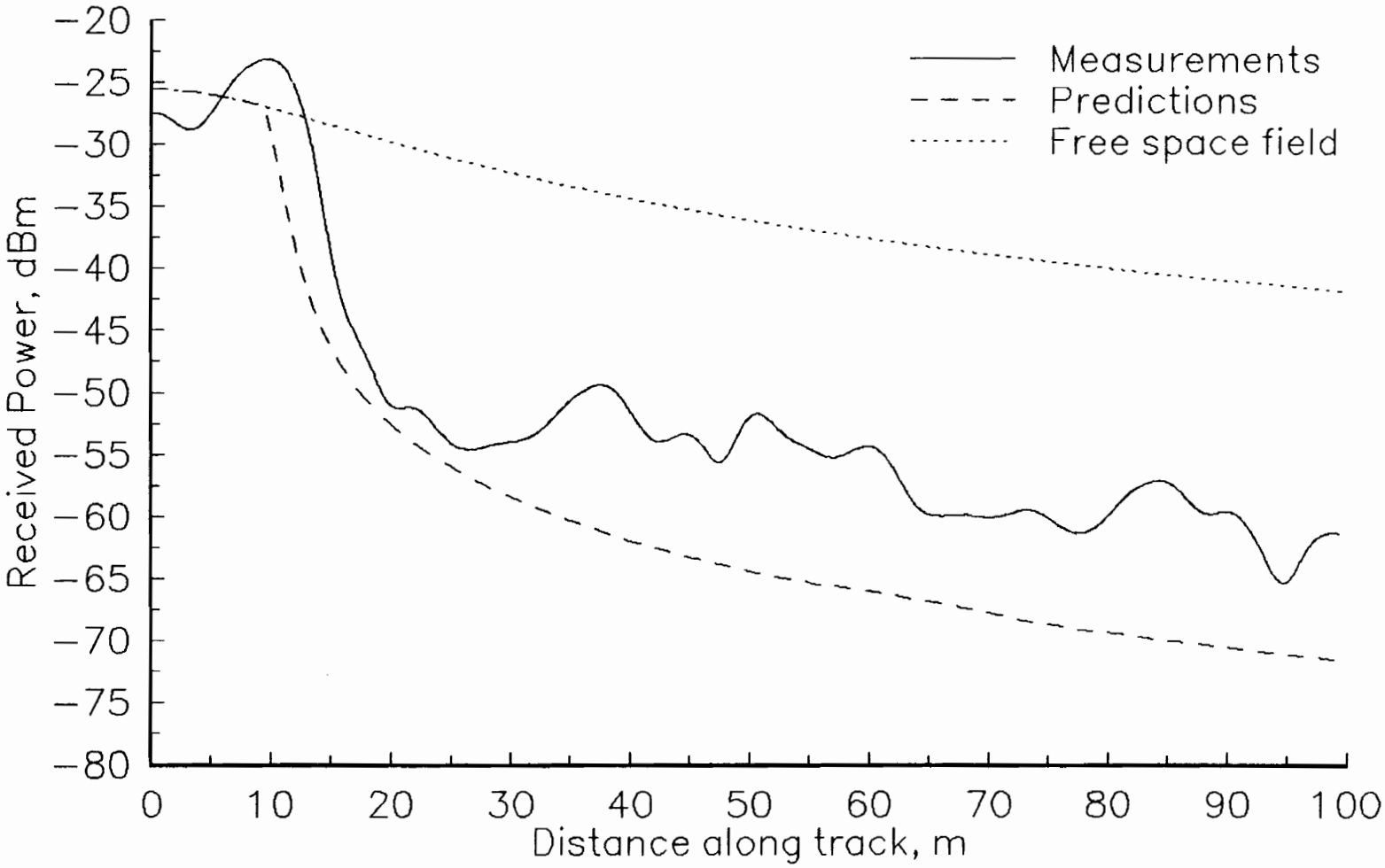


Figure 5.24. Davidson Hall: Predictions Compared with Measurements

prediction. For the range 35-100 m on the track, the measured signal is 17-20 dB below the free space level and the predicted diffraction field is 25-30 dB below free space, leaving an 8-10 dB difference. There are again trees in the vicinity that could be contributing scattered energy.

5.3.4 Conclusions

The results of the comparisons with the Patton Hall and Davidson Hall measurements define very clearly the capabilities as well as limitations of a diffraction prediction model. Diffraction is only one way in which the signal can change direction to penetrate the shadow zone of an obstacle; scattering and reflections are two other common ways. The level of scattered and reflected energy may not be expressly dependent on the depth of the shadow, as is the diffraction level. Thus, when there is fairly uniform illumination of a shadow zone with low-level scattered field, and no significant specular reflections, the diffraction field will be dominant until the building presents enough of an obstruction that diffraction field drops below the level of scattered radiation.

Two points can be inferred:

1. A diffraction model, used alone as a prediction tool, will predict worst case signal levels. The measurement comparisons, though limited, appear to confirm this. Pessimistic predictions are useful in some aspects of system design, but not for interference analysis.

2. A model for scattered field is necessary for accurate prediction of signal level

in well-shadowed areas, at least in the absence of strong specular reflections.

VI. NUMERICAL METHODS

6.1 Fresnel Integrals

Definitions of the Fresnel integral vary widely. We will use the form given by Hecht [33], among others,

$$F(\nu) = \int_0^{\nu} e^{i\pi t^2/2}\, dt, \tag{6.1}$$

where

$$F(\nu) = C(\nu) + i\, S(\nu) \tag{6.2}$$

and

$$C(\nu) = \int_0^{\nu} \cos(\pi t^2/2)\, dt, \tag{6.3}$$

$$S(\nu) = \int_0^{\nu} \sin(\pi t^2/2)\, dt. \tag{6.4}$$

The general case involving two variable integration limits can be written

$$\int_{\nu_1}^{\nu_2} e^{i\pi t^2/2}\, dt = F(\nu_2) - F(\nu_1). \tag{6.5}$$

The Fresnel integrals (6.3) and (6.4) have the following important properties,

$$C(\infty) = 1/2, \tag{6.6}$$

$$S(\infty) = 1/2, \tag{6.7}$$

$$C(\nu) = - C(-\nu), \tag{6.8}$$

$$S(\nu) = - S(-\nu). \tag{6.9}$$

We compute the Fresnel integrals (6.3) and (6.4) with a standard Romberg adaptive numerical integration for $|\nu|$ less than 5.0. The Romberg technique makes successive approximations to the value of the integral by subdividing the integration region into smaller intervals for each evaluation, until three consecutive approximations to the value differ by less than a specified error tolerance. In order to guard against a premature assumption of convergence, a danger when evaluating the oscillatory Fresnel integrals, we use three different tests: a minimum depth of four is required so that at least four cycles are made through the process of subdivision and evaluation, a tolerance of 0.01 must be satisfied on the difference between the current evaluation and the evaluation made two cycles prior, and a tolerance of 0.002 is required on the difference between the current evaluation and the immediately preceding evaluation.

An alternate technique would be to make use of a look-up table containing pre-computed values of $C(\nu)$ and $S(\nu)$ for ν between 0 and 5. We have not implemented this method because the memory required to store the entire table would be prohibitive, while the time required to read the correct entry in the table from a data file may be greater than that required to integrate

numerically. The numerical integration, which allows specification of the desired error tolerance, provides the flexibility required in the investigation of the other parts of the model; the look-up table is an option that should be considered for future use of the model.

When the absolute value of ν is greater than 5.0, the integrals (6.3) and (6.4) tend toward asymptotes, so the approximate asymptotic functions published by Abramowitz and Segun [34] are used:

$$C(\nu) = 0.5 + (0.3183099 - \frac{0.0968}{\nu^4}) \frac{\sin(\pi\nu^2/2)}{\nu} -$$

$$(0.10132 - \frac{0.154}{\nu^4}) \frac{\cos(\pi\nu^2/2)}{\nu^3} + \epsilon(\nu), \tag{6.10}$$

$$S(\nu) = 0.5 - (0.3183099 - \frac{0.0968}{\nu^4}) \frac{\cos(\pi\nu^2/2)}{\nu} -$$

$$(0.10132 - \frac{0.154}{\nu^4}) \frac{\sin(\pi\nu^2/2)}{\nu^3} + \epsilon(\nu), \tag{6.11}$$

where the maximum error is given by

$$|\epsilon(\nu)| < 3 \text{ X } 10^{-7}. \tag{6.12}$$

6.2 Integration Method for Multiple Diffraction

Multiple diffraction involves many numerical integrations, so to minimize computation time an efficient integration method was implemented. This method follows Whitteker [28], who extended the work of Stamnes et al. [35] to include integration to infinity. The computational speed of the multiple diffraction procedure depends on the number of evaluations of the field, E(Q), required to obtain an accurate result. This number can be minimized by performing integrations over interpolating polynomials. The method employed here to integrate over y in (5.9) (the same technique applies to the integral over x in (5.11)) divides the integration region into sub-intervals. The number of sub-intervals and the spacing is discussed in Section 6.2.4. (The integration region is itself a portion of the total integration range, as explained in Section 5.1.3. The method described here applies to both the finite and infinite portions, where a final step to infinity is taken in the latter case.) On each sub-interval we compute the coefficients of interpolation functions for both the amplitude and phase. The amplitude interpolator is quadratic, and the phase function can be treated as linear or quadratic, depending on the size of the computed quadratic coefficient.

The integration over the n^{th} sub-interval is of the form (assuming the general case of quadratic phase)

$$I_n(t_1,t_2) = \Delta \int_{t_1}^{t_2} (A_n + B_n t + C_n t^2) \exp[i(\alpha_n + \beta_n t + \gamma_n t^2)]\, dt, \quad (6.14)$$

where Δ is the sub-interval size, the coefficients A_n, B_n, C_n, α_n, β_n, and γ_n are computed from the amplitude and phase at three local evaluation points, according to [35], and the variable of integration is

$$t = \frac{y\text{-}y_n}{\Delta} . \tag{6.15}$$

The limits of integration, (t_1,t_2), for the sub-intervals derive from the following limits on y,

$$(y_0,\ y_0+\tfrac{\Delta}{2}), \qquad n = 0; \tag{6.16}$$

$$(y_n\text{-}\tfrac{\Delta}{2},\ y_n+\tfrac{\Delta}{2}), \qquad 1 \le n \le (N\text{-}1); \tag{6.17}$$

$$(y_N\text{-}\tfrac{\Delta}{2},\ y_N), \qquad n = N; \tag{6.18}$$

and the evaluation points used in the determination of the interpolating polynomials are

$$y_0, y_1, y_2 \qquad n = 0; \tag{6.19}$$

$$y_{n-1}, y_n, y_{n+1} \qquad 1 \le n \le (N\text{-}1); \tag{6.20}$$

$$y_{N-2}, y_{N-1}, y_N, \qquad n = N. \tag{6.21}$$

The total integral over the region is the sum of the sub-interval integrations,

$$I = \sum_{n=0}^{N} I_n. \tag{6.22}$$

Now we will consider integrations over individual sub-intervals, and drop the subscripts on I and on the polynomial coefficients.

6.2.1 Quadratic Phase

Stamnes et al. [35] suggest a test on the quadratic phase coefficient, γ, such that if $|\gamma| \geq 0.6$, a quadratic approximation to the phase, as in (6.14), should be made, otherwise the phase should be treated as linear. When the phase is modeled as quadratic, a change of variable,

$$u = \sqrt{|\gamma|}\,(t + \beta/2\gamma), \tag{6.23}$$

is made in (6.14), leaving

$$I(u_1,u_2) = \Delta\sqrt{\frac{1}{|\gamma|}}\, e^{i\psi} \int_{u_1}^{u_2} (a + bu + cu^2)\, e^{iu^2} du, \tag{6.24}$$

where ψ, a, b, and c are functions of the polynomial coefficients A, B, C, α, β, and γ, (see [35] for details), and the integration limits u_1 and u_2 are found from t_1 and t_2 with (6.23). We now write (6.24) as the sum of three integrals,

$$I(u_1,u_2) = \Delta\sqrt{\frac{\pi}{2|\gamma|}}\, e^{i\psi} \left[aG_0(u_1,u_2) + bG_1(u_1,u_2) + cG_2(u_1,u_2)\right], \tag{6.25}$$

where

$$G_n(u_1,u_2) = \sqrt{\frac{2}{\pi}} \int_{u_1}^{u_2} t^n \, e^{it^2} \, dt, \quad n=0,1,2. \tag{6.26}$$

Through integration by parts we find

$$G_0(u_1,u_2) = F(\sqrt{\frac{2}{\pi}}\, u_2) - F(\sqrt{\frac{2}{\pi}}\, u_1), \tag{6.27}$$

$$G_1(u_1,u_2) = \sqrt{\frac{2}{\pi}}\, \frac{i}{2}\, (e^{iu_1^2} - e^{iu_2^2}), \tag{6.28}$$

$$G_2(u_1,u_2) = \frac{i}{2} \left\{ \sqrt{\frac{2}{\pi}} \left[u_1 e^{iu_1^2} - u_2 e^{iu_2^2} \right] + F(\sqrt{\frac{2}{\pi}}\, u_2) - F(\sqrt{\frac{2}{\pi}}\, u_1) \right\}, \tag{6.29}$$

where $F(\nu)$ is the Fresnel integral defined in (6.1) and computed according to Section 6.1. The above expressions are only valid when $\gamma > 0$ (which is always the case, as discussed below), otherwise, the negative must be taken of the imaginary part of each term in (6.27), (6.28), and (6.29).

6.2.2 Linear Phase

When γ is small and the phase variation over the interval can be modeled as linear, then, by neglecting the quadratic phase term, and making a change of variable,

$$u = \beta t, \tag{6.30}$$

equation (6.14) can be written

$$I(u_1,u_2) = \Delta \frac{e^{i\alpha}}{\beta} \int_{u_1}^{u_2} (a + bu + cu^2)\, e^{iu}\, du, \tag{6.31}$$

where

$$a = A, \tag{6.32}$$

$$b = B/\beta, \tag{6.33}$$

$$c = C/\beta^2. \tag{6.34}$$

Following as in the quadratic phase case, we write the integral (6.31) as the sum of three integrals,

$$I(u_1,u_2) = \Delta \frac{e^{i\alpha}}{\beta} \left[aL_0(u_1,u_2) + bL_1(u_1,u_2) + cL_2(u_1,u_2)\right], \tag{6.35}$$

where

$$L_n(u_1,u_2) = \int_{u_1}^{u_2} t^n\, e^{it}\, dt, \qquad n=0,1,2. \tag{6.36}$$

The integrals (6.36) can all be evaluated analytically, producing

$$L_0(u_1,u_2) = i\,(e^{iu_1} - e^{iu_2}), \tag{6.37}$$

$$L_1(u_1,u_2) = e^{iu_2}(1 - iu_2) - e^{iu_1}(1 - iu_1), \tag{6.38}$$

$$L_2(u_1,u_2) = i\left\{e^{iu_1}\left[u_1^2 - 2(1 - iu_1)\right] - e^{iu_2}\left[u_2^2 - 2(1 - iu_2)\right]\right\}. \tag{6.39}$$

6.2.3 Integration to Infinity

The integration to infinity is evaluated according to the method of Whitteker, [36], [28], [37]. Three sample points are taken as the minimum needed to characterize the amplitude and phase with quadratic interpolation functions. Whitteker's method provides for integration to infinity by replacing the upper integration limit of the last interval with infinity, while integrating over amplitude and phase interpolation functions found from the three sample points. This is an infinite extrapolation of the interpolation functions; accordingly, the form of these functions is important. The quadratic, rather than the linear, phase representation should always be used as it is much more representative of the phase variation over large ranges. In Figure 6.1, the curvature of the phase of the integrand can be seen for the single building situation of Figure 5.3, and a 10 m transmitter height.

When infinity is substituted for u_2 in (6.27), (6.28), and (6.29), we see that Fresnel integrals and exponentials must be evaluated at infinity. The value of Fresnel integrals at infinity is given by (6.2) together with (6.6) and (6.7). The exponential terms can be evaluated at infinity by substituting i-ϵ for i, where ϵ is an arbitrarily small, positive real number, following Whitteker. We must first assume a value of |u| which will not be exceeded on any of the subintervals (u_1,u_2) on which the integral (6.24) will be evaluated, except of course for the last, infinite, sub-interval. Then we must take an appropriately

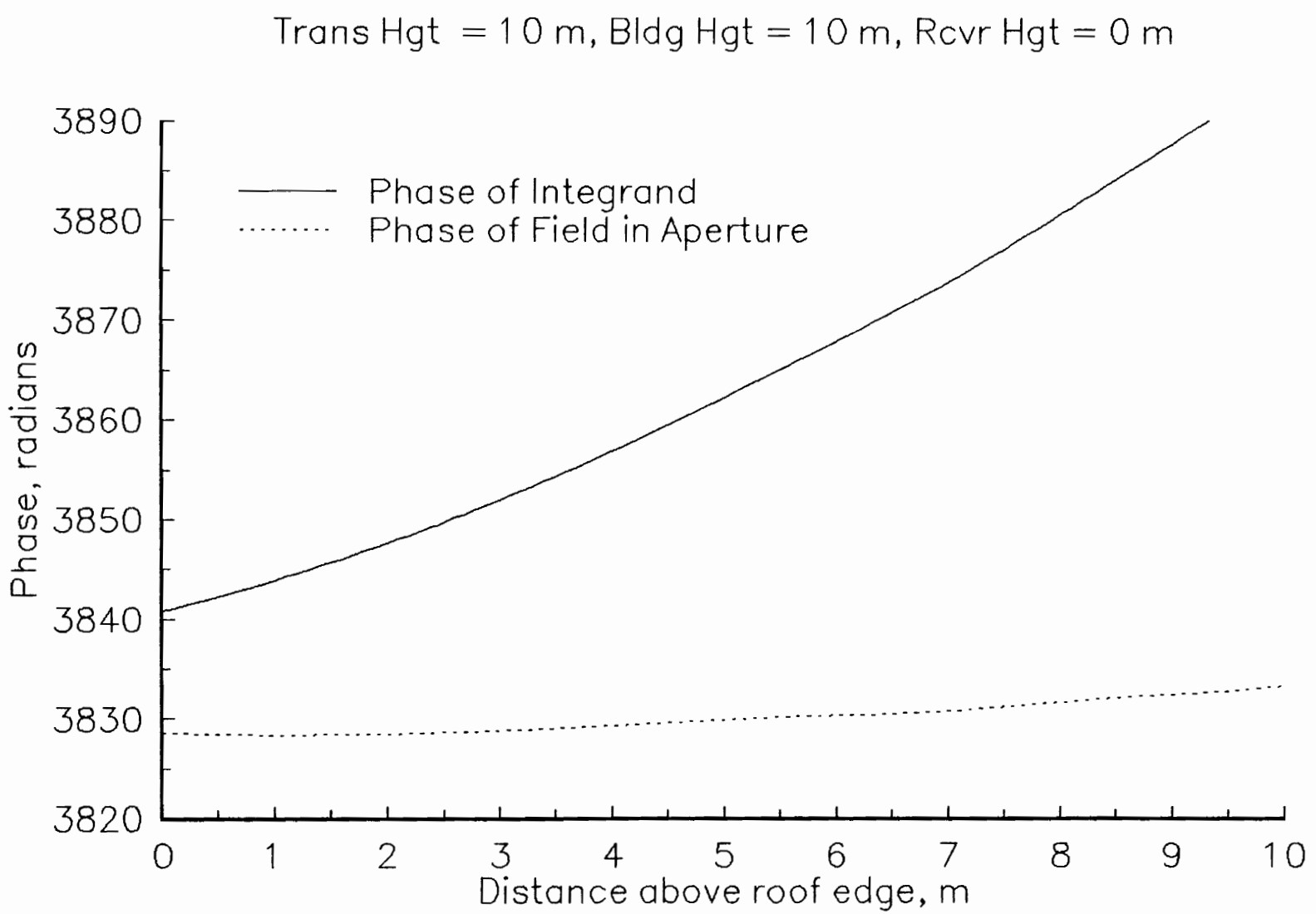


Figure 6.1 . Phase of Integrand, Multiple Diffraction

small value for ϵ, such that $\epsilon << 1/u^2$, for any u within our bounds. For example, we may take the maximum finite |u| to be $1X10^4$, a reasonable bound based on testing with various values of sample number and spacing, and then assign ϵ the value $1X10^{-20}$. The relevant term of (6.28) is of the form

$$e^{(i-\epsilon)u^2} = e^{iu^2}e^{-\epsilon u^2}. \tag{6.40}$$

The first term on the right side of (6.40) oscillates in phase with constant amplitude 1. The exponential part of the second term, $-\epsilon u^2$, has a maximum absolute value of $(1X10^{-20})\cdot(1X10^4)^2 = 1X10^{-12}$, and we can write

$$e^{-\epsilon u^2} \simeq 1, \tag{6.41}$$

with a maximum error of approximately $1X10^{-12}$. Now (6.40) can be written

$$e^{(i-\epsilon)u^2} \simeq e^{iu^2}, \tag{6.42}$$

with the same error, for any u within our bounds.

When u_2 is extended to infinity on the final integration step, and ϵ is any non-zero, positive, real constant, we can find the following limits,

$$\lim_{u_2\to\infty}(-\epsilon u_2^2) = -\infty, \tag{6.43}$$

$$\lim_{u_2 \to \infty} e^{-\epsilon u_2^2} = 0, \tag{6.44}$$

and

$$\lim_{u_2 \to \infty} e^{(i-\epsilon)u_2^2} = 0. \tag{6.45}$$

We can then make the approximation (6.42) in (6.28) and incur only a small error on the finite sub-intervals, and find that the exponential term drops out of the equation when u_2 is extended to infinity.

Following the same approach, the exponential term in (6.29) that involves u_2 can be written

$$u_2 e^{(i-\epsilon)u_2^2} = e^{iu_2^2} u_2 e^{-\epsilon u_2^2}. \tag{6.46}$$

We can write the second and third terms on the right side as follows,

$$u_2 e^{-\epsilon u_2^2} = \frac{u_2}{e^{\epsilon u_2^2}}. \tag{6.47}$$

This is an indeterminate form, as both the numerator and denominator go to infinity in the limit as $u_2 \to \infty$. We can apply L'Hopital's rule as follows,

$$\lim_{u_2 \to \infty} \left(\frac{u_2}{e^{\epsilon u_2^2}} \right) = \lim_{u_2 \to \infty} \left(\frac{1}{(2\epsilon u_2) e^{\epsilon u_2^2}} \right) = 0, \tag{6.48}$$

and we can consider the term $u_2 e^{(i-\epsilon)u_2^2}$ to go to zero when evaluated at $u_2 = \infty$. With the error in approximation (6.42) neglected, equations (6.28), (6.29), and (6.30), evaluated at $u_2 = \infty$, respectively become,

$$G_0(u_1,\infty) = F(\infty) - F(\sqrt{\tfrac{2}{\pi}}\, u_1), \tag{6.49}$$

$$G_1(u_1,\infty) = \sqrt{\tfrac{2}{\pi}}\, \tfrac{i}{2}\, e^{iu_1^2}, \tag{6.50}$$

$$G_2(u_1,\infty) = \tfrac{i}{2} \left\{ \sqrt{\tfrac{2}{\pi}}\, u_1 e^{iu_1^2} + F(\infty) - F(\sqrt{\tfrac{2}{\pi}}\, u_1) \right\}. \tag{6.51}$$

6.2.4 Implementation

The sample spacing for the first integration region is set at 0.2 m, or about 2/3 wavelength at 900 MHz. Walfisch and Bertoni [11], using only linear representations of the amplitude and phase and some different assumptions in the integral formulation, found that point spacing should be less than a wavelength for an error in the integrand of less than 0.8 percent. Our error in the integrand, using quadratic polynomials, should be less. When our sample spacing was reduced to 0.1 m and 0.165 m, and the number of points increased in order to hold approximately constant the sampling range, the error in the computed power was less than 0.2 dB for all test cases, including up to 4 diffracting edges.

The sampling interval for the second region, Δ_2, should be comparable with the change in height for which the clearance changes by one Fresnel zone, according to reference [28]. In the oscillatory portion of each curve in Figure 5.4 we see amplitude variations with a spatial period that decreases with increasing clearance. This period relates to Fresnel zone clearance in that each newly unobstructed Fresnel zone interferes in an alternately constructive or destructive manner with the central, dominant zone (the size of Fresnel zones decreases with increasing zone number, see Section 4.1, thus causing the decreasing period). Thus a change in clearance of one Fresnel zone corresponds to one half period of the amplitude oscillations, a sampling frequency that is well known as the minimum required, Nyquist rate.

The examples in Figure 5.4 show a half-period of amplitude oscillations that varies from 1.0 to 2.0 m, depending on the portion of the curve sampled. As we have already chosen 0.2 m (at 900 MHz) as the value of Δ_1, the sample spacing for region 1, the choice of N_1, the number of samples in the first region, determines the size of the first region and the starting point of the second region. The choice of Δ_2, since we always use 3 samples in the second region, determines the size of this region, which is $2\Delta_2$. The geometry of Figure 5.5 is used to test the sensitivity of the integration to the choice of N_1 and Δ_2 at 900 MHz. Building heights of 15 and 10 m were used for the first and second buildings, respectively, and a receiver height of 0 m, to produce the curves in Figure 6.2.

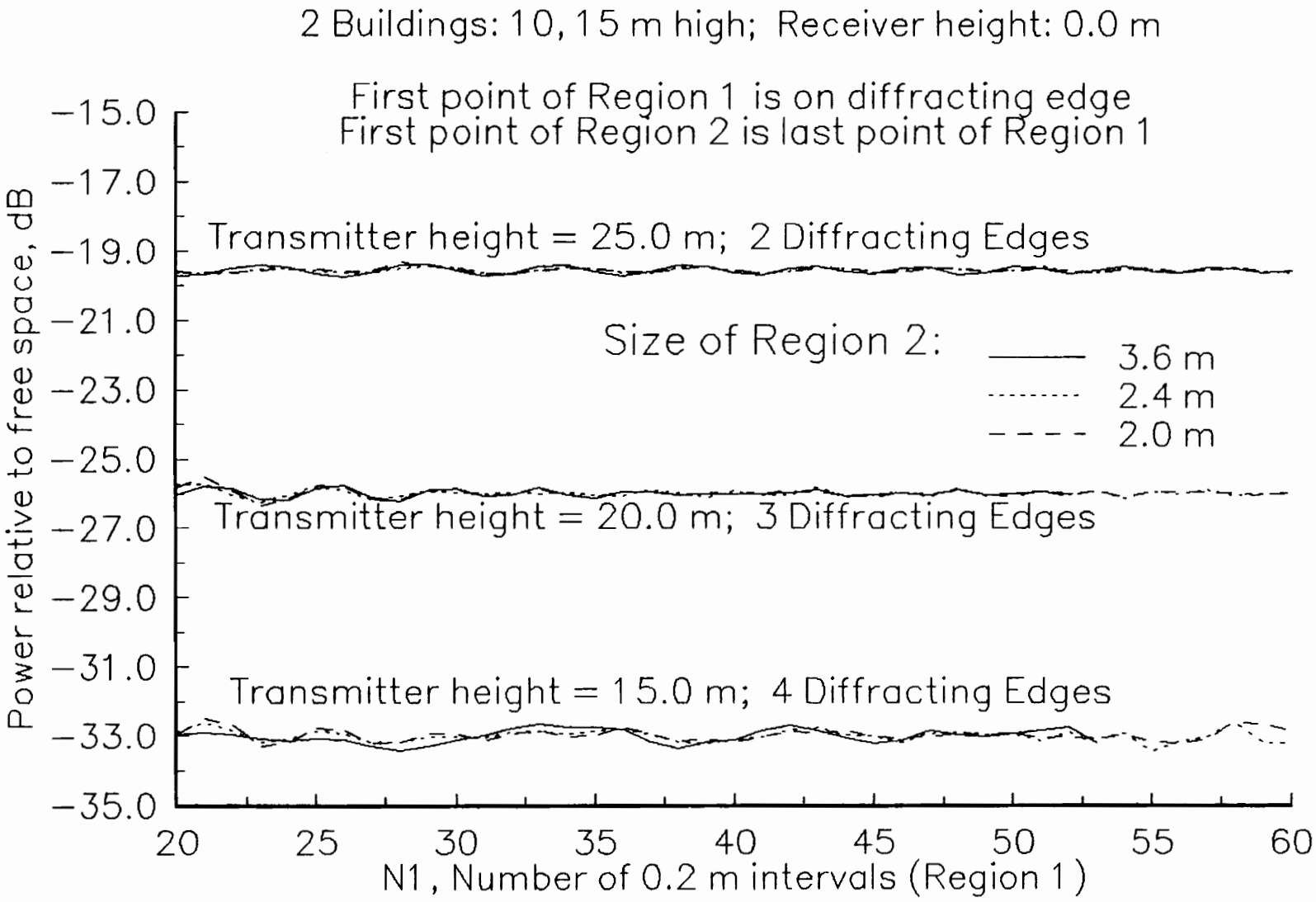


Figure 6.2. Sensitivity to Integration Range and Sample Placement

There are three sets of three curves in Figure 6.2, one set for each of three transmitter heights. The three curves making up each set are the results of using three different values of Δ_2: 1.8 m, 1.2 m, and 1.0 m, or approximately 5.5λ, 3.6λ, and 3.0λ. The three transmitter heights were chosen such that the the total number of diffracting edges considered was different for each case. (If an edge leaves 55% of the first Fresnel zone unobstructed, it is neglected.) The results turned out to be very robust to the choice of N_1 and Δ_2 for this building geometry, with variations of less than 2 dB peak-to-peak for all cases. The smallest variations were seen with $\Delta_2 = 1.0$ m, so this value was chosen.

These results seem to demonstrate a certain insensitivity to two different issues: the size of the total sampled region, and the portion of the amplitude and phase curves that is used as the basis of the extrapolation to infinity. The sampled portions of these curves for two of the cases included in the above test, ($N_1 = 20$, $\Delta_2 = 1.0$m) and ($N_1 = 27$, $\Delta_2 = 1.8$m), are explicitly marked in Figures 6.3 (amplitude) and 6.4 (phase). The two sets of samples of the amplitude variations in Figure 6.3 will be fit by completely different parabolas: one negative and one positive in curvature. Since there is only a small difference between the results for these two cases, we can infer a lack of dependence of the computed field on the behavior of the amplitude function as it is extrapolated to infinity.

On the other hand, the phase of the integrand, shown as the solid line in

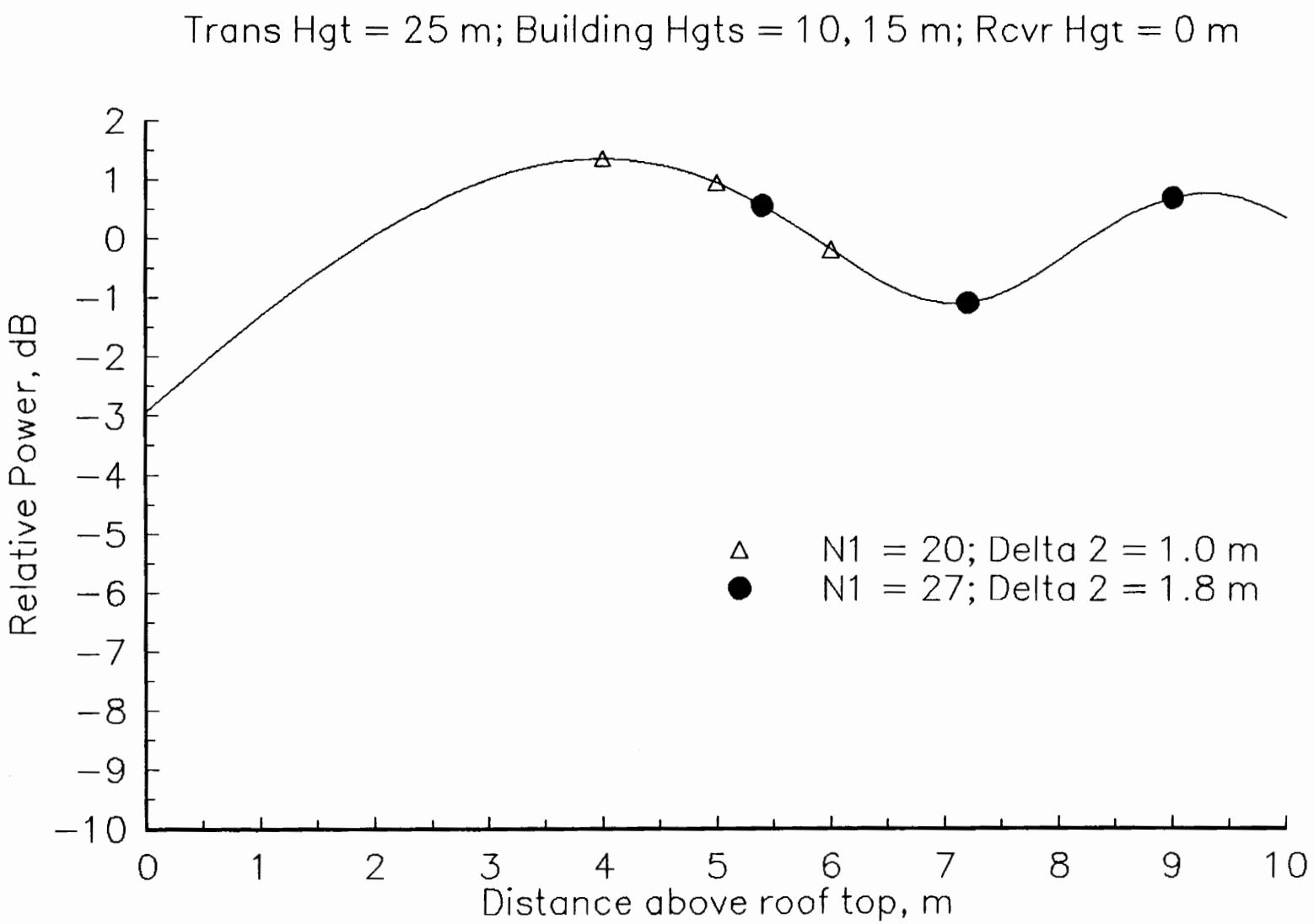


Figure 6.3. Amplitude Samples; Integration Region 2

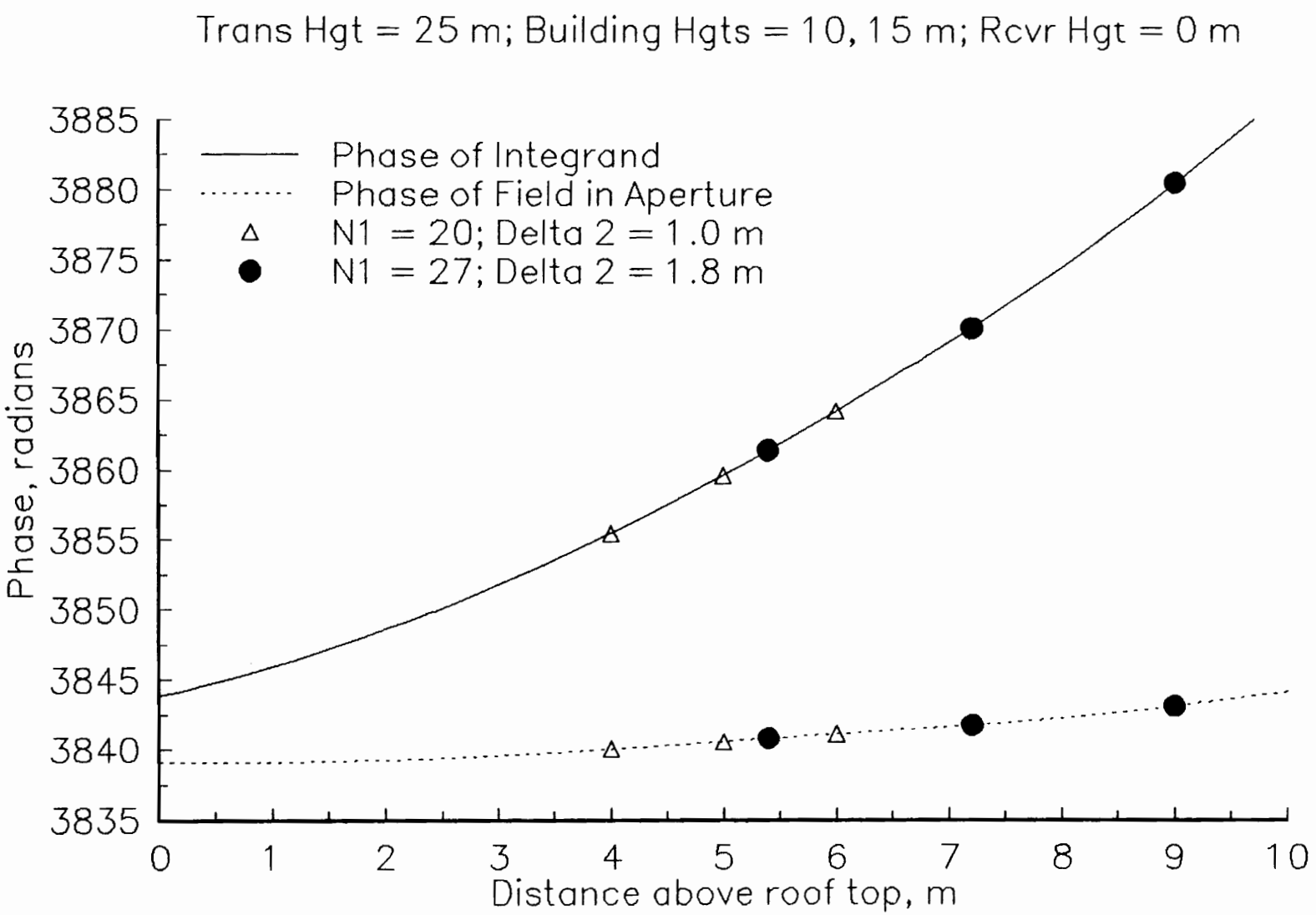


Figure 6.4. Phase Samples; Integration Region 2

Figure 6.4, includes a $\pi y^2/\lambda p$ term which is added to the phase of the field in the aperture to describe propagation from the obstacle to the observation point, P, and which in this case appears to dominate over the phase of the aperture field, providing a smooth curve that appears as if it could be well approximated by a single second degree polynomial. However, when the phase of the field in the aperture is examined on an expanded scale, as in Figure 6.5, some oscillation of the phase is distinctly visible. This oscillation does affect the form of the interpolating polynomial, but the above results show a robustness to this effect for the particular building geometry tested. In tests of other building geometries, often the oscillations affect the phase of the integrand sufficiently to cause the interpolating parabola to have a negative curvature when certain portions of the curve are sampled, and the computed results are then obviously wrong, showing a 40 dB difference with a slightly perturbed situation having a positive phase curvature. Hence, while the result is not sensitive to small variations in the coefficients of the phase parabola, it is necessary that the function have a positive curvature so that in the extrapolation to infinity, the function will go to positive infinity, as indeed the phase always does (as the path length from S to Q to P increases so will the phase rotation along the path, after a certain clearance of the edge is obtained).

There are two other issues in the choice of N_1 and Δ_2. First, the finely sampled portion of the integration range (which we call Region 1) must extend far enough that the obstructed portions of the amplitude and phase curves are

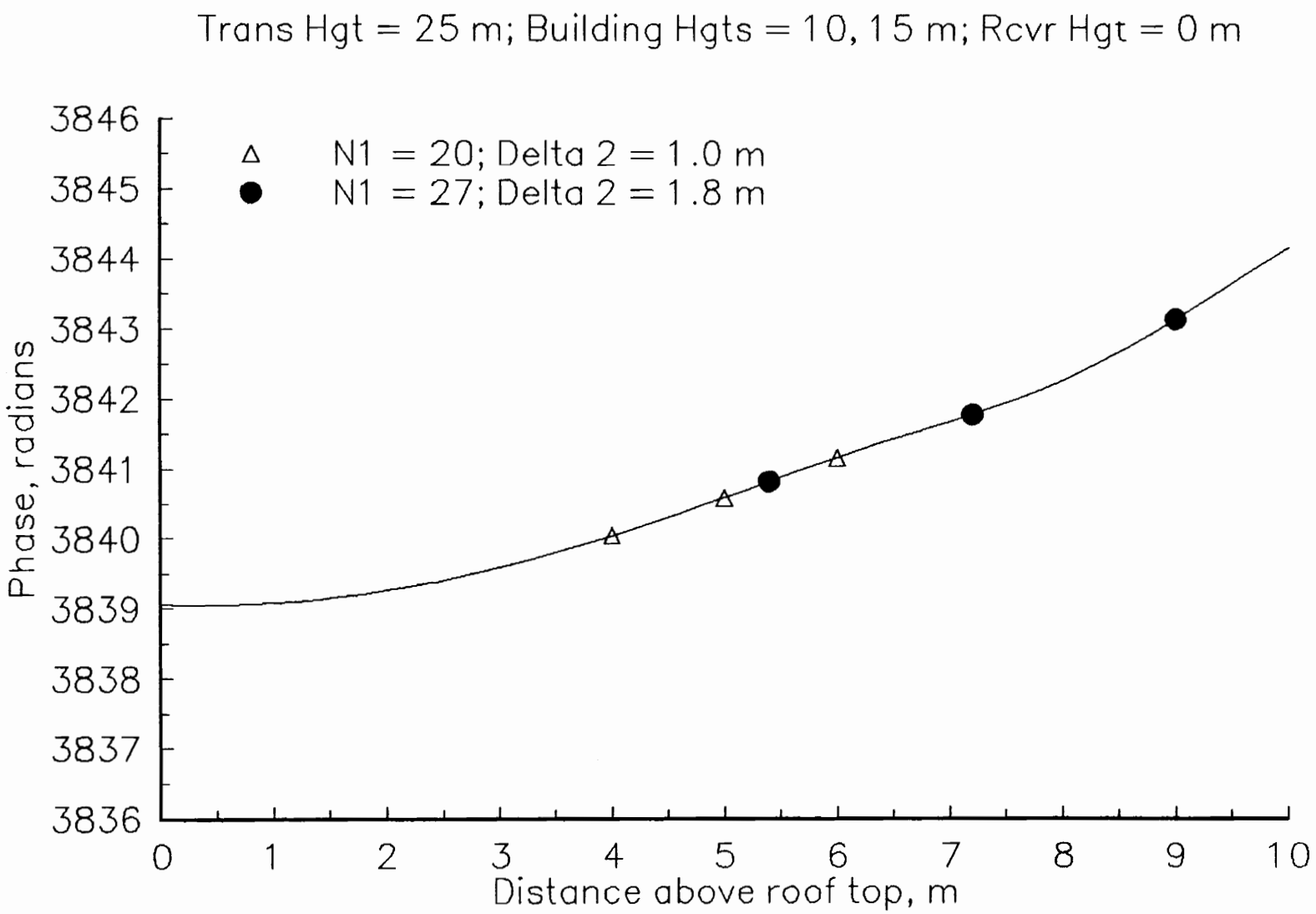


Figure 6.5. Phase of Field in Aperture; Integration Region 2

well characterized. The widely sampled portion (Region 2) is meant to characterize the unobstructed interference fringes converging at infinity. Second, we must assume some minimum size for the total sampled region, in order that even in the case of sub-path obstacles at Q′ (those that obstruct the field without blocking the line of sight), the field at Q is characterized well enough to allow solution for the field at points P. A conservative estimate would be to take a minimum N_1 of 25, providing a minimum of 5 m (at $\Delta_1 = 0.2$ m) for the first region.

In the roof diffraction examples displayed in Figures 5.4, the minimum 5 m region also satisfies the first issue above in that the first integration region extends to the oscillatory part of the curve. This is not true for a fairly typical case of diffraction around the left side of a building, caused by two successive corners, where the receiver is only in a slight shadow of the first corner, but points Q at that corner are in a deep shadow of the prior corner (but not so deep that we can consider the field blocked). The amplitude and phase curves for such a case are presented in Figures 6.6 and 6.7. The samples taken of the amplitude curve are also shown (with $\Delta_2 = 1.0$ m), where the last point taken in Region 1 is the point which has grazing clearance (where the relative power is always 6 dB down). In Region 2 we can see the beginning of the amplitude peak. For this situation, Region 1 extends more than 16 m, requiring 83 samples. Since many situations require a much smaller region, we implemented an adaptive method of determining N_1, ending the first region when clearance of the

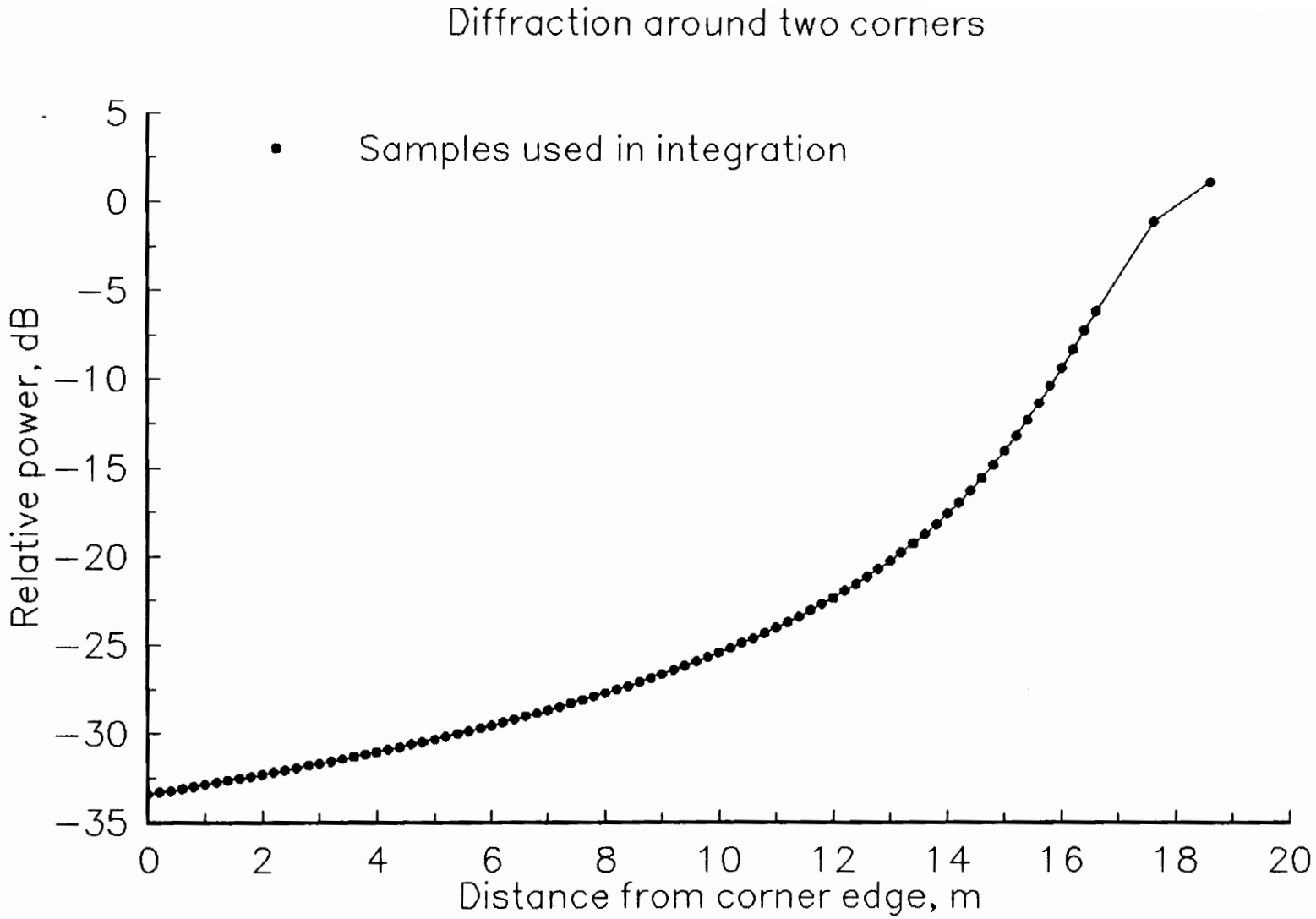


Figure 6.6. Amplitude for Case of Second Edge in Deep Shadow

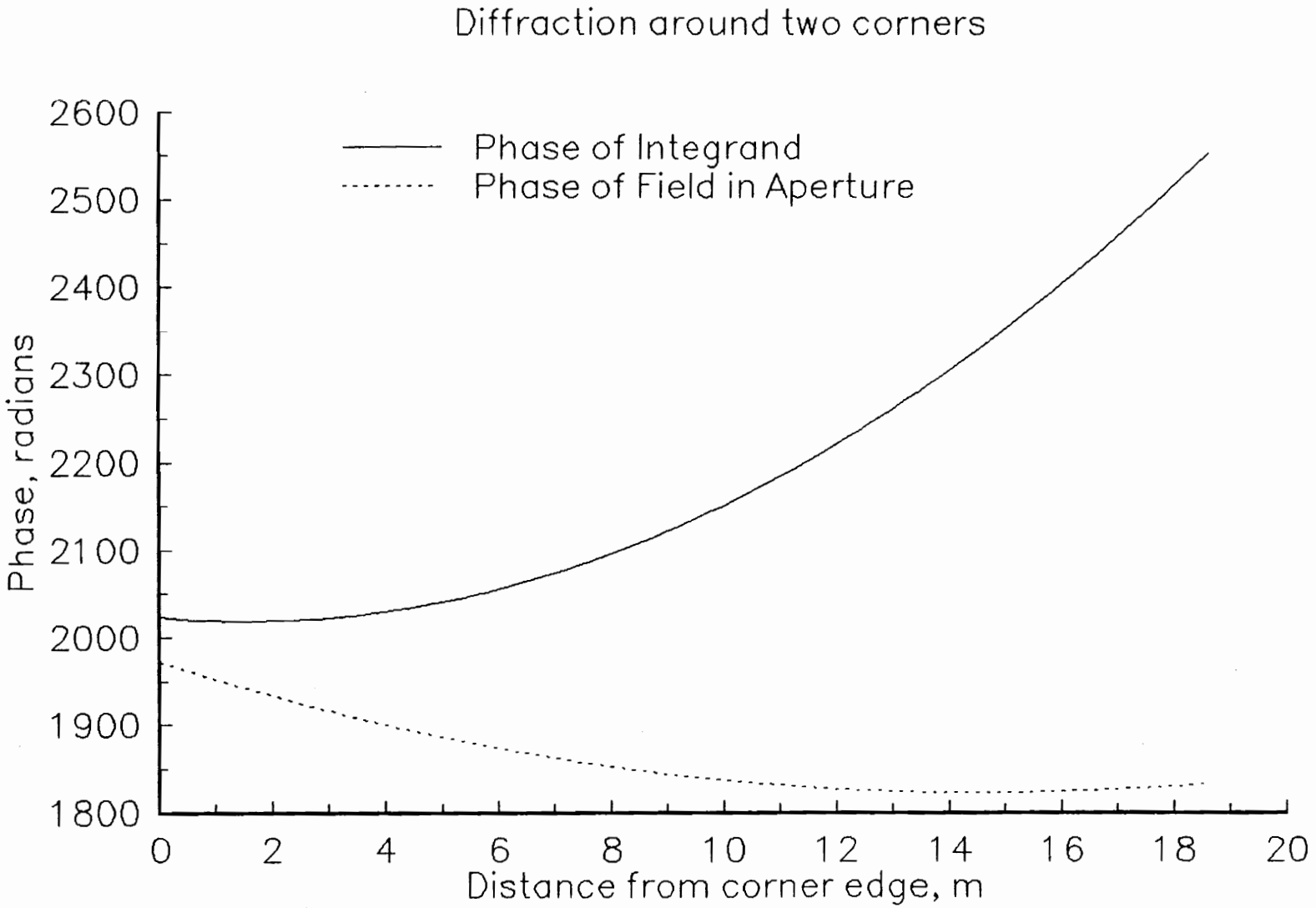


Figure 6.7. Phase for Case of Second Edge in Deep Shadow

line of sight is first obtained.

Returning to the issue of the phase curvature in Region 2, we see in Figure 6.7 that the phase of the field in the aperture (the dashed line) has positive curvature at the very beginning of the unobstructed region, so in the cases where sample N_1 is the first point in the unobstructed region (because the minimum has been exceeded), the phase curvature is not a problem. In the cases where the second edge is in a shallow shadow, or obstructed by a sub-path obstacle, and the minimum range for Region 1 extends into the oscillatory region, we ensure positive phase curvature on Region 2 by ending Region 1 not at the minimum range, but extending it to the next point of minimum phase slope. At this point the phase curvature is zero and going positive. Figure 6.8 shows the phase of the field in the aperture for the situation of Figure 5.3 and transmitter heights of 15 and 10 m, with the points of minimum phase slope marked. These points occur at values of Fresnel zone clearance of $n = 2k + 1.7$, where $k = 0,1,2,....$ In Figure 6.9 we see that these are also the points of minimum amplitude.

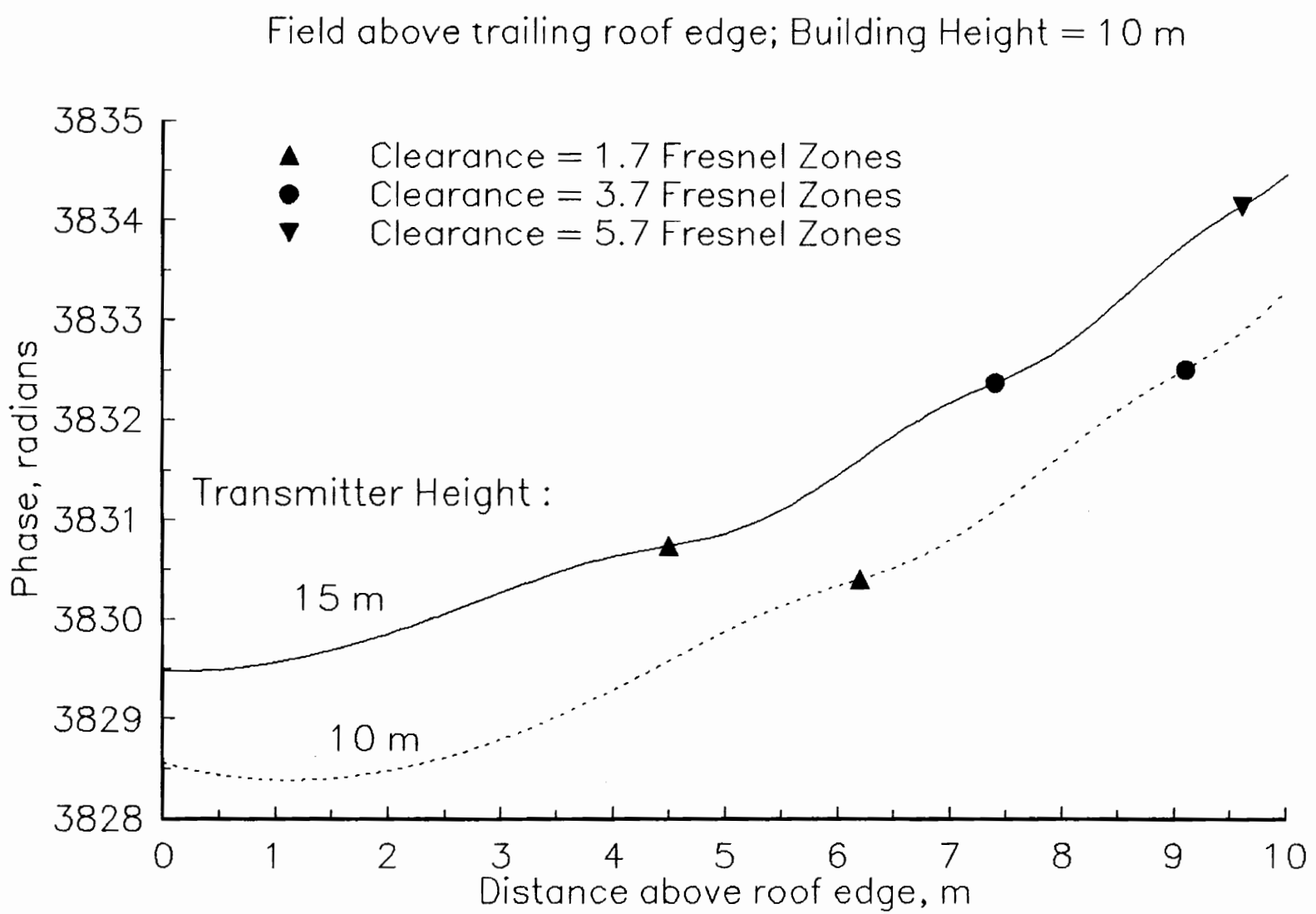


Figure 6.8. Phase Slope Minima

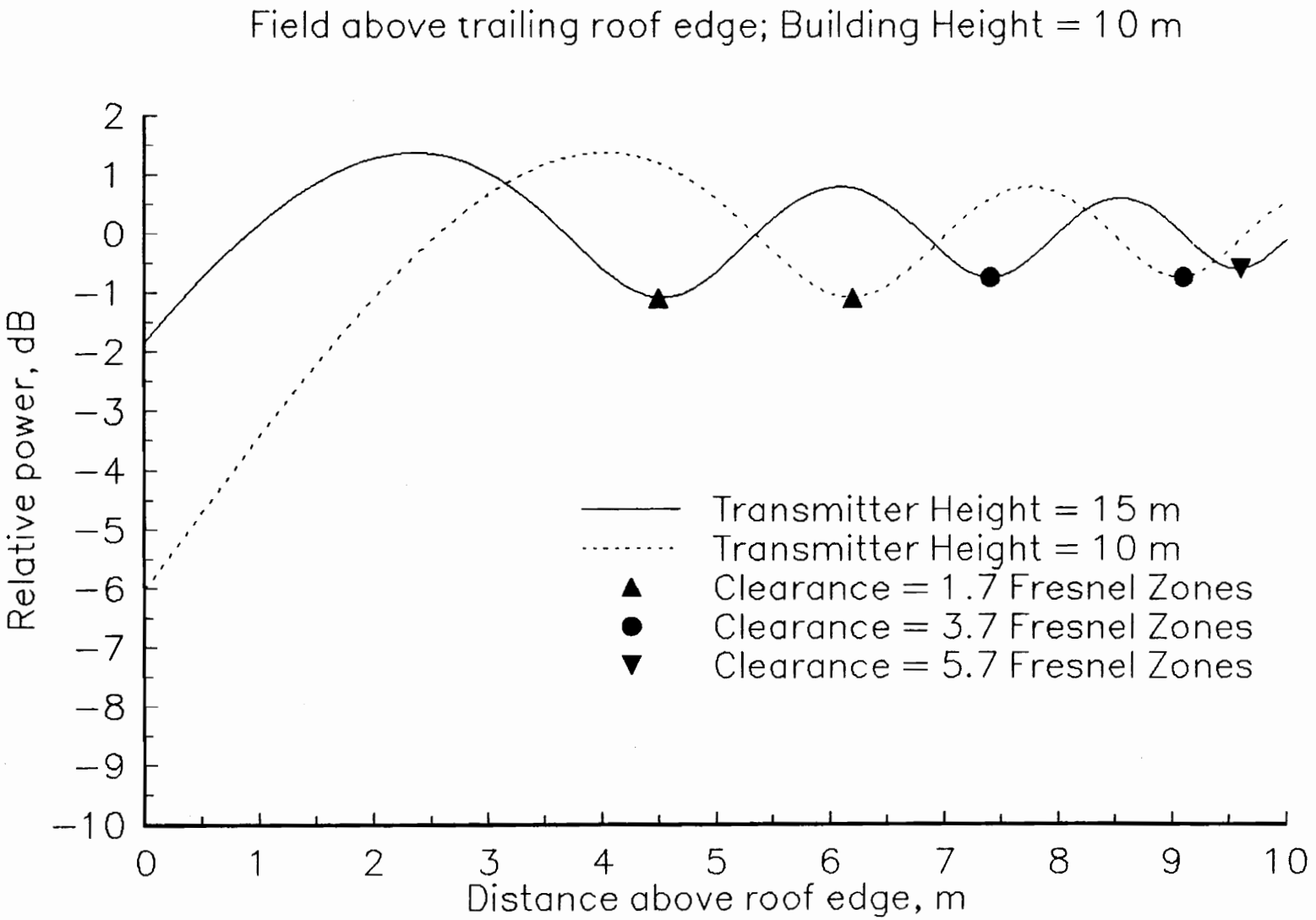


Figure 6.9. Amplitude Minima

VII. SOFTWARE

The methods and models described in this thesis are implemented in a C-language program. The program consists of self-contained subroutines, all well commented, including headers for each routine and variable definitions. The subroutines are split into several files: MDSRCH.C, MDB1.C, MDB2.C, EVALFLD.C, and INPTDATS.C. There are also two header files which are included in each program file: MDSRCH.H and MDTYPES.H, which include some global constants used to dimension arrays, and some user-defined data types. The entire program listing is included in the appendix.

Capabilities

The program will accept the coordinates of an ensemble of buildings, a transmitter point, and a receiver point, and perform the following general operations:

1. Identify the diffracting building nearest to the receiver.
2. Sort the edges of the diffracting building.
4. Compute the diffraction field at the receiver according to the methods of Chapters 3 through 6, recursively identifying each diffracting building and building edge between the transmitter and the receiver.
5. Output to a file the amplitudes of the field components at the receiver, the phasor sum of those components, and the power sum.

Parameters

1. Fresnel zone clearance beyond which a diffracting edge is neglected : $0.55R_1$

2. Maximum value of the diffraction parameter (a unit-less parameter defined in Equation (4.11b), for example) for consideration of the diffraction field: 22 (corresponding to approximately 40 dB of diffraction loss). Edges with larger values are considered to block all diffraction and none is computed.

3. Minimum separation between edges for computation of multiple diffraction: 5λ. Two edges with less separation will be considered a single edge. A larger value for this parameter may be safer, as close edges cause large phase slopes in the aperture, and may introduce problems in phase un-wrapping (discussed later in this chapter).

Memory Issues

The recursive nature of the algorithms requires functions to call themselves and others in a long chain. The memory space that holds the variable information for the list of function calls is known as the stack, and when the allocated stack space fills up, the program terminates with a stack overflow error. The primary consumers of stack space are the arrays that hold the amplitude and phase samples of the field distribution in the apertures. The program currently uses a global constant called SZ to dimension these arrays. This constant must be set to the maximum number of samples expected to be needed in order to characterize the field. When this number is reached without

the criteria for terminating the integration being satisfied, an error message is printed out. If desired, a "BLOCKED" flag can be set so that the field will not be computed in such a case. The constant, SZ, was set to values ranging from 50 to 86 to perform the runs in this thesis. These runs were performed on a 386SX computer with the Turbo C compiler, where the maximum stack space we could allocate was 35 kilobytes. On a Sun workstation, where it is expected this program will ultimately be used, there should be sufficient stack space to set SZ to a larger value. Dynamic allocation of memory may also be advantageous.

Phase Un-wrapping

The phase samples of the field in the aperture contain 2π discontinuities and must be "un-wrapped" to produce a continuous function which can be fit with an interpolating polynomial. If the phase change between sample points is too large, the un-wrapping may fail. Usually when this occurs, the computed quadratic phase coefficient will be negative, which triggers an error message. In this case, often a tweaking of the routine responsible for this function, SmoothPhs2, will fix the problem. A solution that will always work is a reduction of the sample spacing, although this has the disadvantages of slowing the computations and requiring more memory, as it requires more samples.

Special Cases

1. Connected Buildings.

Buildings that are made up of sections of different heights must be

specified in the database as individual buildings. There is a routine in the program called TCHNGBLDGS that checks the database for buildings that share an edge. These are marked with a variety of flags so that, for example, the diffraction is not computed around a corner that is connected to a taller building and so does not really exist; rather, the true corner of the connected building is considered as contributing diffraction field in the same manner as the other corners of the original building.

2. False Diffractors.

There is an inherent problem with the procedure wherein the relationship between the source point, the nearest building edge to the receiver, and the receiver point, is examined to determine whether an edge contributes diffraction. There may be an earlier edge in the radio path that is so high or wide that the path from that edge to the receiver has plenty of clearance over the later edge, even when the later edge is a diffracting edge by the usual criteria. This is a particularly common situation with successive corners of a single building, as in Figure 5.10. Here the first corner on the right of SP, if judged alone, certainly blocks the direct signal path, forcing it to go around the corner. However, as is obvious from the figure, the prior corner on the right renders it insignificant as a diffractor. Not only do we wish to avoid computing diffraction by an edge when not necessary, but in a situation such as that in Figure 5.10, the field must be sampled out quite a distance in order to satisfy the requirement of line of sight clearance past the prior edge, thus consuming a great deal of unnecessary

computing time.

A check for this situation has been implemented in the software, called checkintermcorn. For each corner that is judged to be a diffractor, the next corner is found and treated as a new source point. The necessary coordinate transformations are performed and the original corner is considered based on the new source point and the original observation point. This test eliminates false diffractors.

This situation is not commonly seen for roof edges of separated buildings. It would require a great differential in building heights, with the shorter building nearest the receiver, so the test is not performed in these cases, although it may be a useful addition to the program. A more common occurence of false diffractors among roof edges is with connected buildings, where roof edges of different heights are right up against each other, so the test is performed in this case, with a routine called checkintermroof.

VIII. REFLECTIONS

Reflected energy is an important source of signal strength in the urban mobile radio environment, and a model that includes accurate predictions of reflected and diffracted energy, based on individual building placements, should provide a good approximation to the total signal strength in many situations. The compatibility between the diffraction prediction approach described in this thesis and the image theory approach to reflection prediction is discussed here, but a thorough consideration of the prediction of reflections is beyond the scope of this work.

In the urban environment, diffractions and reflections may interact in extremely complex ways, but some approximate modeling techniques may allow prediction of the dominant mechanisms.

8.1 Diffracted/Reflected Field

The image theory of reflections, which involves the construction of the source or observation point on the opposite side of a reflecting surface, has been combined with knife-edge diffraction by Ikegami et al. [4]. The assumption is that a diffraction field can have a specular reflection in the same way as a direct field. Figure 8.1 illustrates the construction of the image point for a typical

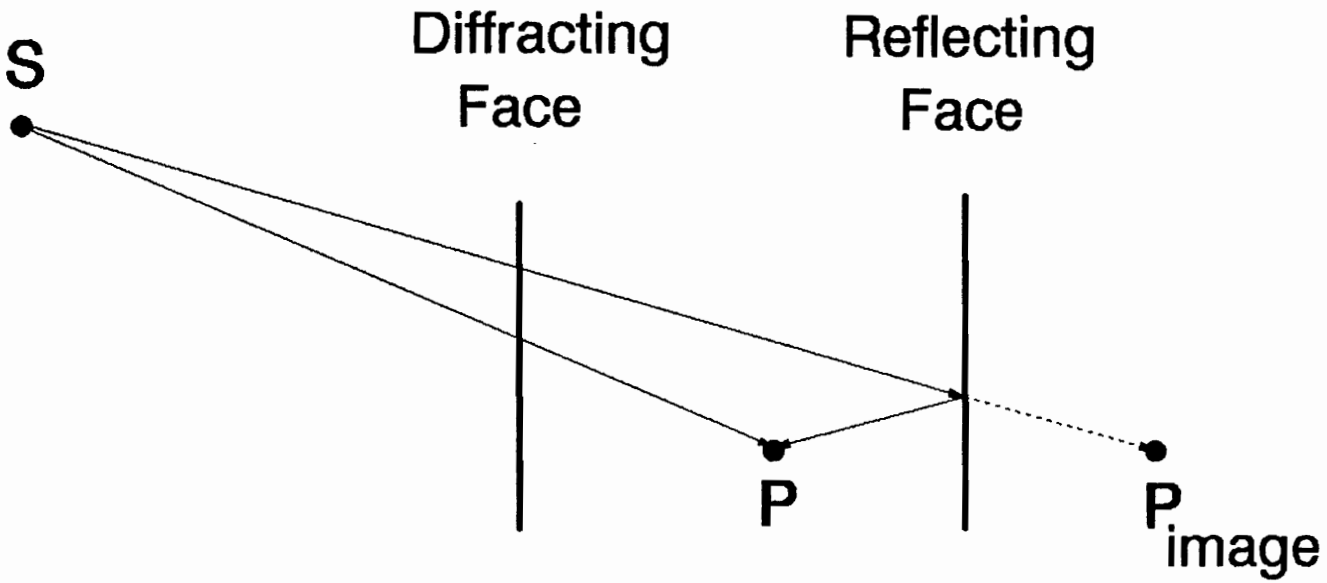


Figure 8.1. Image Theory applied to
Diffracted/Reflected Fields

urban situation where the receiver lies between two buildings and lower than both, and the source is behind one of the buildings. It is apparent that in this type of situation, the diffraction field at the image point will suffer less diffraction loss than that at the true point. The field at P_{image} is found by multiplying the diffraction field by a complex reflection coefficient. This coefficient varies according to building material and is difficult to characterize, as windows, doors and imperfections make for a complicated situation. Accordingly, approximate average values are used commonly, such as a simple 6 dB loss factor with no phase shift [4]. The total field at P is the sum of the computed fields at P and P_{image}.

In the campus measurement area pictured in Figure 4.17 a possible source of diffracted/reflected field is apparent to the right of measurement track SL22. The image of the receiver was constructed in order to compute this field, which is plotted alongside the direct diffracted field in Figure 8.2. Even with a 6 dB reflection loss and the longer propagation path, the diffracted/reflected signal is a significant source of field relative to the direct path. The phasor sum of the two fields is plotted in Figure 8.3, showing strong multipath nulls particularly towards the end of the track where the fields are closer in amplitude.

8.2 Reflections from Rooftops

Whitteker [21] and Haslett [22] have published models for solid objects that include a diffraction by the front edge, a reflection by the connecting face,

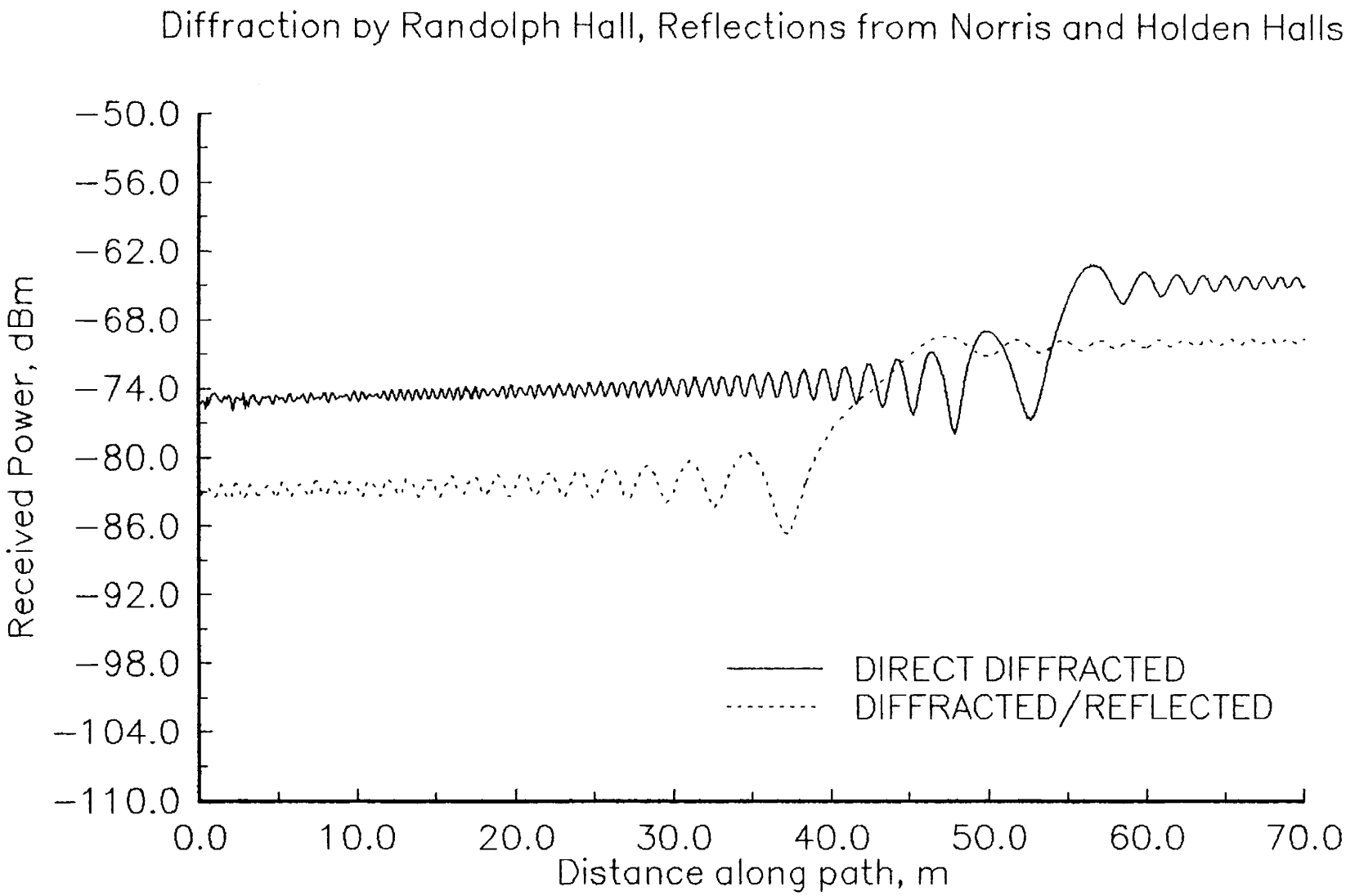


Figure 8.2. Diffracted and Diffracted/Reflected Fields

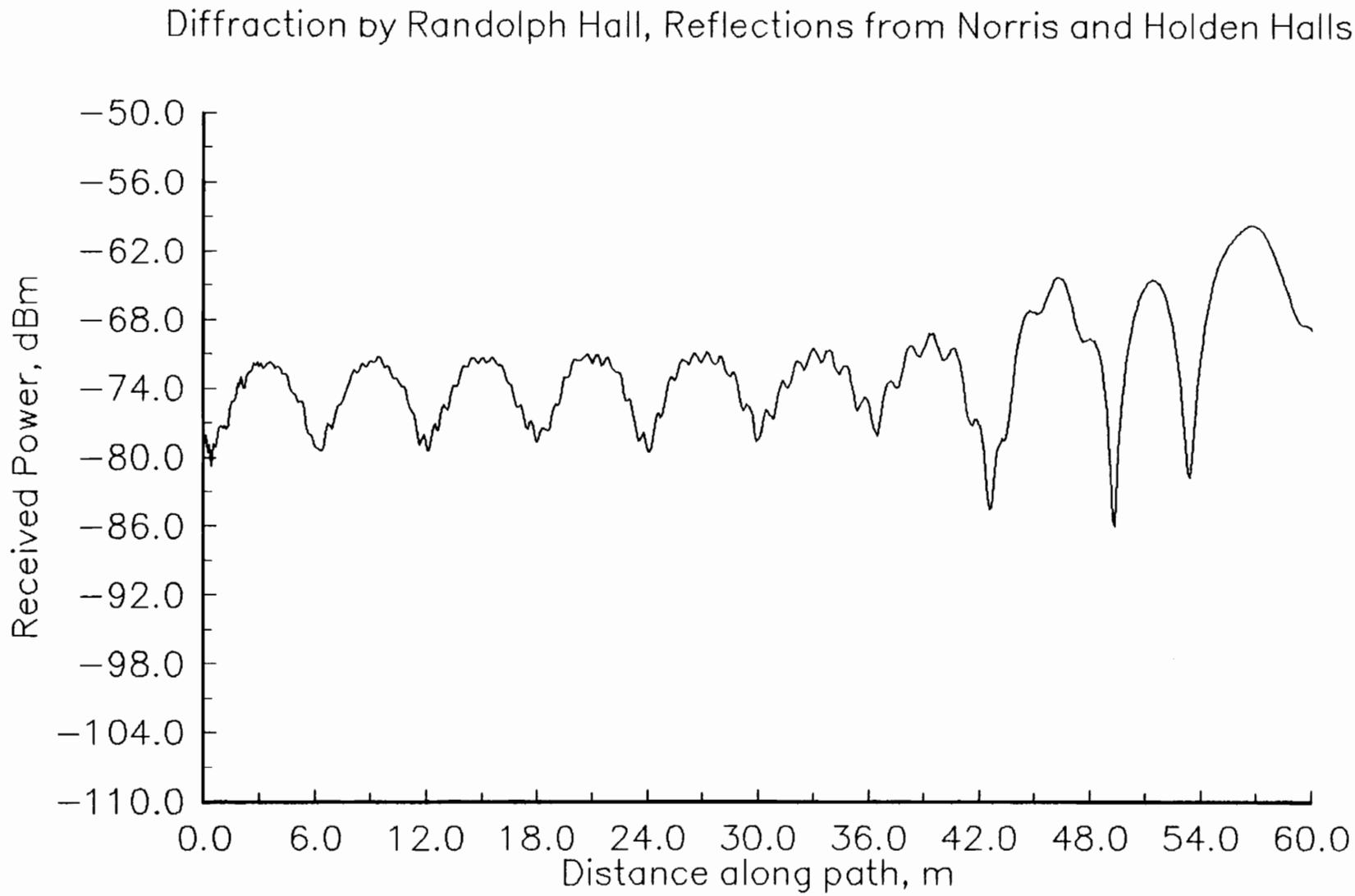
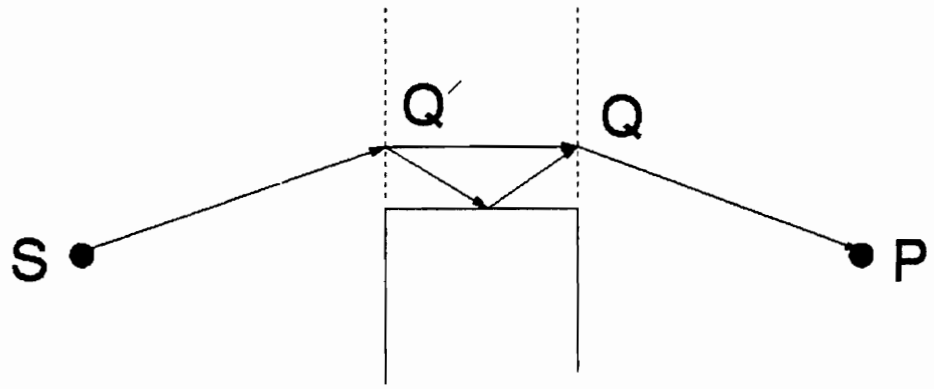
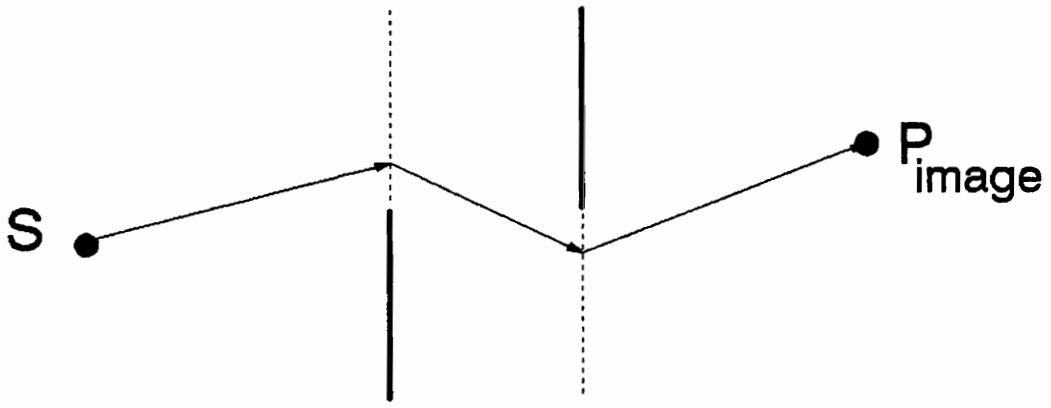


Figure 8.3. Phasor Sum of Diffracted and Diffracted/Reflected Fields

and a diffraction by the trailing edge. The geometry of the situation is given in Figure 8.4a. Two radio paths are used to model propagation from Q to Q′: a direct path and a reflection from the connecting face. Whitteker's development was intended for flat-topped terrain obstacles, while Haslett applied the idea to building rooftops. It is a natural model for the sides of buildings as well, where the edges now represent the corners of the building. In Figure 8.4b is presented the image construction for the solution of the field component at P that has undergone reflection from the connecting face. It is apparent in comparing this path with the direct path in Figure 8.4a that the diffracted/reflected path in this type of situation will always be forced to bend through larger angles, and thus undergo deeper diffraction than the direct path. Nevertheless, it may be an important field component and the diffraction solution methods in this thesis are amenable to the addition.



a) Ray paths



b) Image construction

Figure 8.4. Reflection from Rooftop

IX. CONCLUSIONS

Based on the work in this thesis, we can make the following conclusions:

1. Geographic Information Systems (GIS) software packages can provide a useful tool for site-specific propagation prediction. As far as the models in this thesis are concerned, GIS packages are necessary in deriving the approximate height of buildings, which may be specified in municipal databases only by number of floors. They also provide a platform for display and storage of the buildings. The capability of GIS systems to produce terrain profiles will be a benefit when extensions including ground reflections or diffractions are made to the present prediction models.

2. Using simplified geometries for buildings, the three-dimensional profile of the radio path is surprisingly easy to access and to use as a basis of diffractive shadowing prediction. The search algorithms implemented in the prediction program can very quickly (in an imperceptable amount of time for the small databases we tested) identify and sort both vertical and horizontal edges that diffract the signal.

3. Consideration of the diffraction by all of the building edges in the radio path can be accomplished in acceptable time, contingent on the system designer's

needs. The computation time for characterizing the field along a one-dimensional track is on the order of a half hour, using a 386SX computer with a math co-processor, on a 100 m track, with prediction points spaced every 1/3 m, and only one or two buildings obstructing the path at any given time. In the prediction of local mean field strengths these samples can be spaced much more widely. However, a two-dimensional prediction grid may be time-consuming, particularly in the case of diffraction by many buildings at a time. The speed-up gained in using a high-performance workstation as a computing platform should allow these models to be time-efficient enough to be useful in system design, provided the service area is not very large and complex.

4. Predictions based solely on diffractive shadowing have limitations. An accurate diffraction prediction model will match well with measurements only when the diffraction field is the dominant signal source. Prediction of diffraction alone may be adequate, however, when a pessimistic, or worst case, signal strength prediction is desirable.

Future work is needed in the following areas:

1. The diffraction models need more extensive experimental confirmation. Conclusive validation, which requires isolating diffraction effects, may be nearly impossible without building scale models. Any outdoor environment is complex

enough to provide numerous scattering and reflection sources which can obscure the diffraction field. Nevertheless, more experimental confirmation would give more confidence in these models.

2. Models for reflections and scattering should be combined with these diffraction models for a more complete characterization of urban propagation.

3. Interfaces between these modeling routines, the GRASS database, and the user need to be developed.

X. REFERENCES

[1] J. H. Whitteker, "Measurements of path loss at 910 MHz for proposed microcell urban mobile systems," *IEEE Trans. Veh. Technol.*, Vol. 37, No. 3, pg. 125-129, Aug. 1988.

[2] F. H. Palmer, "VHF/UHF path-loss calculations using terrain profiles deduced from a digital topographic data base," *AGARD Conf. Proc.*, No. 269, 1979, pg. 26-1 - 26-11.

[3] W. C. Y. Lee, "Estimate of local average power of a mobile radio signal," *IEEE Trans. Veh. Technol.*, Vol. VT-34, No. 1, pg. 22-27, Feb. 1985.

[4] F. Ikegami et al., "Propagation factors controlling mean field strength on urban streets", *IEEE Trans. Antennas Propagat.*, Vol. AP-32, No. 8, pg. 822-829, Aug. 1984.

[5] Y. Okumura et al., "Field strength and its variability in VHF and UHF land-mobile radio service," *Rev. Elec. Comm. Lab.*, Vol. 16, Nos. 9 and 10, pg. 825-873, Sept.-Oct., 1968.

[6] M. Hata, "Empirical formula for propagation loss in land mobile radio services," *IEEE Trans. Veh. Technol.*, Vol. VT-29, No. 3, pg. 317-325, 1980.

[7] J. Durkin, "Computer prediction of service areas for VHF and UHF land mobile radio services," *IEEE Trans. Veh. Technol.*, Vol. VT-26, pg. 323-327, Nov. 1977.

[8] A. G. Longley and P. L. Rice, "Prediction of tropospheric radio transmission loss over irregular terrain, a computer method," ESSA Tech. Rep., ERL-79-ITS 67, Institute for Telecommunication Sciences, Boulder, CO.

[9] J. H. Whitteker, "Propagation prediction from a topographic data base," *Proc. IEEE Int. Conf. Commun.*, 1983, pg. A2.1.1 - A2.1.5.

[10] J. F. Aurand and R.E. Post, "A comparison of prediction methods for 800 MHz mobile radio propagation," *IEEE Trans. Veh. Technol.*, Vol. VT-34, No. 4, Nov. 1985.

[11] J. Walfisch and H.L. Bertoni, "A theoretical model of UHF propagation in urban environments," *IEEE Trans. Antennas Propagat.*, Vol. 36, No. 12, pg. 1788-1796, Dec. 1988.

[12] S. R. Saunders and F. R. Bonar, "Building diffraction modeling for area coverage predictions in mobile radio propagation," *IEE Colloquium Digest*, No. 1990/157, pg. 4/1-4/3, Nov., 1990.

[13] A. Ranade, "Local access radio interference due to building reflections," *IEEE Trans. Commun.,* Vol. 37, No. 1, pg. 70-74, Jan. 1989.

[14] F. Ikegami et al., "Theoretical prediction of mean field strength for urban mobile radio", *IEEE Trans. Antennas Propagat.,* Vol. 39, No. 3, pg. 299-302, March 1990.

[15] M. Born and E. Wolf, *Principles of Optics*, 3rd ed. New York: Pergamon, 1965.

[16] J. W. Goodman, *Introduction to Fourier Optics.* New York: McGraw-Hill, 1968.

[17] J. C. Schelleng et al., "Ultra-short wave propagation," *Bell Syst. Tech. J.,* pg. 125-161, Apr. 1933.

[18] A. B. Carlson and A. T. Waterman, "Microwave propagation over mountain diffraction paths," *IEEE Trans. Antennas Propagat.,* Vol. AP-14, No. 4, pg. 489-496, July 1966.

[19] M. P. Bachynski and M. G. Kingsmill, "Effect of obstacle profile on knife-edge diffraction," *IRE Trans. Antennas Propagat.,* Vol. AP-10, pg. 201-205, March 1962.

[20] G. Millington et al., "Double knife-edge diffraction in field strength predictions", *Proc. IEE,* Vol. 109C, No. 16, pg. 419-429, March 1962.

[21] J. H. Whitteker, "Diffraction over a flat-topped terrain obstacle," *Proc. IEE,* Vol. 137, Pt. H, No. 2, pg. 113-116, April 1990.

[22] C. J. Haslett, "The diffraction of microwaves by buildings - A model for system planners," *IEE Colloquium Digest,* No. 1990/157, pg. 6/1-6/6, Nov., 1990.

[23] J. Deygout, "Multiple knife-edge diffraction of microwaves," *IEEE Trans. Antennas Propagat.,* Vol. AP-14, No. 4, pg. 480-489, July 1966.

[24] J. Epstein and D. W. Petersen, "An experimental study of wave propagation at 850 Mc.," *Proc. IRE,* Vol. 41, No. 5, pg. 595-611, 1953.

[25] R. J. Luebbers, "Propagation prediction for hilly terrain using GTD wedge diffraction," *IEEE Trans. Antennas Propagat.,* Vol. AP-32, No. 9, pg. 951-955, Sept. 1984.

[26] L. E. Vogler, "An attenuation function for multiple knife-edge diffraction," *Radio Sci.,* Vol. 17, No. 6, pg. 1541-1546, Nov.-Dec. 1982.

[27] P. A. Sharples and M. J. Mehler, "Cascaded cylinder model for predicting terrain diffraction loss at microwave frequencies," *Proc. IEE,* Vol. 136, Pt. H, No. 4, pg. 331-337, Aug. 1989.

[28] J. H. Whitteker, "Fresnel-Kirchoff theory applied to terrain diffraction problems," *Radio Sci.*, Vol. 25, No. 5, pg. 837-851, Sept.-Oct. 1990.

[29] R. J. Luebbers, "Finite conductivity uniform GTD versus knife edge diffraction in prediction of propagation path loss," *IEEE Trans. Antennas Propagat.*, Vol. AP-32, No. 1, pg. 70-76, Jan. 1984.

[30] H. E. J. Neugebauer and M. P. Bachynski, "Diffraction by smooth cylindrical mountains," *Proc. IRE,* Vol. 46, pg. 1619-1627, Sept. 1958.

[31] J. D. Parsons and J. G. Gardiner, *Mobile Communication Systems.* Glasgow: Blackie, 1989.

[32] T. S. Rappaport and C. D. McGillem, "UHF fading in factories", *IEEE Journal on Selected Areas in Commun.*, Vol. 7, No. 1, pg. 40-48, Jan. 1989.

[33] E. Hecht, *Optics.* Reading, Mass.: Addison-Wesley, 1987.

[34] M. Abramowitz and I. A. Segun, *Handbook of Mathematical Functions.* New York: Dover, 1965.

[35] J. J. Stamnes, B. Spjelkavik, and H. M. Pedersen. "Evaluation of diffraction integrals using local phase and amplitude approximations," *Optica Acta*, Vol. 30, No. 2, pg. 207-222, 1983.

[36] J. H. Whitteker, "Calculation by numerical integration of diffraction attenuation at VHF and UHF," *Proc. IEE Int. Conf. Antennas Propagat.*, 1987, pg. 31-34.

[37] J. H. Whitteker, "VHF/UHF propagation by diffraction - calculation by numerical integration," *AGARD Conf. Proc.*, No. 407, 1986, pg. 6-1 - 6-7.

APPENDIX. Computer Program

```
 /*************************************************************************
* FILE NAME: MDSRCH.H
* DESCRIPTION: Header file for all of the files associated with MDSRCH.C,
        containing standard header files and some constants which
        are used to dimension arrays
*************************************************************************/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define pi M_PI
#define Nb 7  /* Number of buildings */
#define SZ 80  /* Size of multiple diffraction data arrays */
#define Np 10  /* Number of Paths */
#define Nc 4   /* Number of field components (changing this causes problems) */
#define FTtoMTRS 0.3048;
```

```
/*************************************************************************
* FILE NAME: MDTYPES.H
* DESCRIPTION: Header file for all of the files associated with MDSRCH.C,
        containg user-defined data types.
*************************************************************************/
typedef double BAH[Nb];          /* Bldg Array: Heights  */
typedef double BAC1[2][4];       /* Bldg Array: 1 bldg's corners */
typedef double BAC[2][4][Nb];    /* Bldg Array: All bldgs' corners */
typedef double DA[2][SZ];        /* Data Array : APDAT */
typedef double DA2[2][2][SZ];    /* Data Array: APTMP */
typedef int BAF[Nb];             /* Bldg Array: Flags */
typedef int BACF[2][4][Nb];      /* Bldg Array: Corner, Roof Flags */
typedef double ZXUVY[5];         /* Pnt coord's */
typedef double LCL[4][Nb];       /* LOWCORNLIM */
```

```
/*************************************************************************
* FILE NAME: MDSRCH.C
* CONTENTS:
        FindFieldP
        BldgDiff1
        BldgDiff2
        CornDiff
        CornDiff1
        TCHNGBLDGS
        InitAMPDAT
        InitAMPTMP
        InitRItmp
        FindCornConn
        FindUpperEdgeConn
        FindLowerEdgeConn
* DESCRIPTION:
        This file contains the upper level routines and some
        intialization and connected-building functions
*************************************************************************/
#include <mdsrch.h>
#include <mdtypes.h>
int FindFieldP(int RoofCorn, ZXUVY P, BAC zxcorn, BAH ybldg,
                double lam, double D[2], double yt, BACF NOCORNROOF,
                LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
                double delp, double ampfac, DA APDAT, double RIA[3],
                int *RCout, int *BLOCKED, int *N1out);
void BldgDiff1(int j, int RoofCorn, ZXUVY P, BAC zxcorn, BAH ybldg,
                BAC1 uvcorn, double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK,
                BACF CONNIJ, int ConnB,
                double delp, double ampfac,
                DA APDAT, double RIA[3],
                int *RCout, int *BLOCKED, int *N1out);
void BldgDiff2(int j, int NOBlev2, int RoofCorn, ZXUVY P, BAC zxcorn,
                BAH ybldg, double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK,
                BACF CONNIJ, int ConnB,
                int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
                double delp, double ampfac,
                DA APDAT, double RIA[3],
                int *RCout, int *BLOCKED, int *N1out);
void CornDiff(int j, int NOBlev2, int RoofCorn, ZXUVY P, BAC zxcorn,
                BAH ybldg, double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
                int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
                double delp, double ampfac,
                DA APDAT, double RIA[3], int *kcnt,
                double apcomp[2][Nc], int *BLCKDCRN, int *N1corn);
void CornDiff1(int j, int NOBlev2, int RoofCorn, ZXUVY P,
                BAC zxcorn, BAH ybldg, int klr, int kc,
                double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
                int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
                int kcnt1, int FirstCorn, double delp, double ampfac,
                DA2 APTMP, double RItmp[2][2],
                int *BLOCKED, int *RCout, int *N1out);
void TCHNGBLDGS(BAC zxcorn, BAH ybldg,
                BACF NOCORNROOF, BACF CONNIJ, LCL LOWCORNLIM);
void InitAMPDAT(DA APDAT);
void InitAMPTMP(DA2 APTMP);
void InitRItmp(double RItmp[2][2]);
void FindCornConn(BAC zxcorn, int iQ, int jQ, int *jConn);
void FindUpperEdgeConn(BACF CONNIJ, int iQ, int jQ, int *iConn, int *jConn);
void FindLowerEdgeConn(BACF NOCORNROOF, int EDGE[3][2], int j, int *iConnQ);
/* MDB1.C */
```

```
extern void SumRoofDiff(int RoofCorn, DA2 APTMP,
        double RItmp[2][2], int cnt, int N1tmp[2], double RIA[3], int *N1roof);
extern void SumCornDiff(int RoofCorn, DA2 AMPTMP,
        double RItmp[2][2], int cnt, int N1tmp[2],
        DA AMPDAT, double RIA[3], int *N1corn);
extern void ChooseDiff(int rcnt, int kcnt, int RoofCorn,
        DA2 APTMP, DA APDAT,
        double RIAroof[3], double RIAcorn[3],
        int N1roof, int N1corn,
        double RIA[3], int *RCout, int *N1new);
extern void computediff(int RoofCorn, int RC1, int k,
        double yp, double rso, double rpo, double y1, double V1, double V2,
        double uq, /* These are the coord's of yorigin, NOT the field point */
        double lam, int N1, double D[2], double up, double ybldg1, double yt,
        DA APDAT, DA2 APTMP, double RItmp[2][2],
        int *cnt, int N1out[2], int *BLOCKEDout);
extern void FillAmptmp(DA APDAT, DA2 APTMP, int klr);
extern void FillAmpdat(DA2 APTMP, DA APDAT, int klr);
/*  MDB2.C   */
extern void PntCoordzx1(ZXUVY P, ZXUVY Q);
extern void BldgCoorduv1(ZXUVY P, int j, BAC zxcorn, BAC1 uvcorn);
extern void SortBldg1(BAC1 uvcorn,
        int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2]);
extern int checkintermRoof(ZXUVY P, ZXUVY Q, BAH ybldg,
        double lam, BAC zxcorn,
        int iConn, int jConn, int iQ, int jQ);
extern int checkintermCorn(ZXUVY P, ZXUVY Q, double yt,
        double lam, LCL LOWCORNLIM, BAC zxcorn,
        int CORN[2][4], int klr, int kc, int j);
extern int checkFZcorn(ZXUVY P, ZXUVY Q, double yt, double lam,
                double LOWCORNLIM1, int klr,
                double *rso, double *rpo, double *y1,
                double *v1, double *v2);
extern int checkNotBlock(double rso, double rpo, double y1,
        double v1, double v2, double lam);
extern int checkFZ2(int Blev, ZXUVY P,
                double yt, int EDGE[3][2], int nedge[3],
                BAC1 uvcorn, double ybldg1, double lam,
                ZXUVY Q1, ZXUVY Q2,
                double rso[2], double rpo[2], double y1[2],
                double v1[2], double v2[2]);
extern void NearestDiffBldg(ZXUVY P, double yt, BAH ybldg, double lam,
        BAC zxcorn, double umax[Nb], BAF OBSTRUCT, BAF TOOSHORT,
        BAC1 uvcorn, int *NOBLDGS, int *j);
extern void Sift(ZXUVY P, BAC zxcorn, BAF MASK,
        BAF OBSTRUCT, double umax[Nb], int *NOBLDGS);
extern int cyc4neg(int n);
extern int cyc4pos(int n);
/* EVALFLD */
extern int sign(double x);
/* INPTDATS.C */
extern void FindRcvr(int path, double delp, double zro[Np], double xro[Np],
        double yro[Np], ZXUVY R);
extern void inputdata1(int TRANS, double *yt, double zro[Np],
        double xro[Np], double yro[Np], BAH ybldg, BAC zxcorn);
/*           */
/* GLOBAL VARIABLES */
FILE *filef;    /* Output file: amplitudes of field components */
FILE *fileg;    /* Output file: Totals at receiver: amplitude and phase */
FILE *filep;    /* Output file: phases of field components */
FILE *file[4];  /* Output files: diffraction parameters of fld components*/
FILE *fileint;  /* Output file: intermediate results of integration */
FILE *filephs;  /* Output file: phase before and after smoothing */
FILE *filefld;  /* Output file: amp and phase, in aperture and in integrand */
int NoSecondEdge; /* Forces the neglect of leading edges of each building */
```

```
int NoCorners;  /* Forces the neglect of all corner fields */
int Printfld;   /* Causes output to *filefld */
int Printphs;   /* Causes output to *filephs */
int Printint;   /* Causes output to *fileint */
extern unsigned _stklen = 30000; /* Stack Length: 36000 works for SZ=86 */


/*************************************************************************
* FUNCTION: void main()
* DESCRIPTION: Assigns file names; acquires input data necessary: transmitter,
       receiver, and building locations, wavelength; computes diffraction
       field and outputs to a file the amplitude and phase information, based
       on the assumptions of isotropic antenna patterns and no system losses
       at either end.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: none
* OUTPUTS: none
* CALLED BY: none
* FUNCTION CALLS:
       InitAMPDAT: initializes APDAT array
       inputdata1: returns building, transmitter and receiver locations
       TCHNGBLDGS: finds connected buildings
       FindRcvr: locates a mobile receiver at a new point on a path
       FindFieldP: computes diffraction field
*************************************************************************/

void main()
{
       double RIA[3];          /* 0: Real part of field
                                  1: Imaginary part
                                  2: Amplitude squared, Mean field strength */
       double amp,ampdb;       /* Amplitude as magnitude squared, dB's */
       double phs,phsdeg;      /* Phase in radians, degrees */
       double ampmndb;         /* Amplitude sqrd, Mean field Strength */
       double ampfac;          /* Amplitude factor: Includes lam/4Pi,
                                  converts to dBm */
       double ampfsdb;         /* Free Space field, dB's */
       double lam;             /* Wavelength */
       double delp;            /* Delta p, distance along a rcvr path */
       double delht;           /* Delta ht, change in height */
       double r;               /* Distance from transmitter to receiver */
       double yt;              /* Transmitter height */
       double zro[Np],xro[Np],yro[Np]; /* Coordinates of receiver at first
                                  point of each of several possible paths */
       BAH ybldg;              /* Array holding building heights */
       BAC zxcorn;             /* Array holding horizontal locations of
                                     building corners: 0: z coordinate
                                                       1: x coordinate */
       ZXUVY R;                /* Receiver coordinates:
                                  0: z;  1: x;  2: u;  3: v;  4: y  */
       LCL LOWCORNLIM;         /* Lower corner limits for each building,
                                     assumed -10000 unless the building is
                                     connected to a lower one */
       DA APDUM;               /* Dummy amplitude and phase array */
       double D[2];            /* 0: Delta 1, sample spacing in Region 1
                                  1: Delta 2, sample spacing in Region 2 */
       int i,j,pk;             /* Loop Counters */
       int Resp;               /* Response: 1=Choose integration parameters */
       int N1dum;              /* Dummy number of samples in Region 1 */
       int TRANS,PATH;         /* Choice of transmitter and receiver loc's */
       int RC1dum;             /* Dummy indicator of obstacle type */
       int BLOCKED;            /* Flag indicating near-total blockage */
       BACF NOCORNROOF;        /* Flags for non-existent roofs or corners,
                                     due to connections to other bldgs */
```

```
        BACF CONNIJ;            /* Indicators of building connections */
        BAF MASK;               /* Masks for forced neglect of buildings */

/* Set file names */
        filephs = fopen("phsa.out","w");
        filefld = fopen("ph1.out","w");
        fileint = fopen("in1.out","w");
        filef = fopen("md1a.out","w");
        fileg = fopen("mdb.out","w");
        filep = fopen("mdp.out","w");
        file[0] = fopen("cparta6.out","w");
        file[1] = fopen("cpartb6.out","w");
        file[2] = fopen("cpartc6.out","w");
        file[3] = fopen("cpartd6.out","w");

        lam = 300.0/914.0; /* Frequency = 914 MHz */
        ampfac = 20.0 * log10(lam/(4.0*pi)) + 30.0;
        InitAMPDAT(APDUM);

/* Set Flags */
        /*  PATHS:
           0: Single
           1: Multiple (Bldg edge rises)
           2: Davidson
           3: Patton
           4: Trans. lowers
           5: Multiple edges of one building, Recvr moves in hgt
           6: Flexible: Multiple, Recvr moves in delp

           TRANS:
           0: 30 m
           1: (2050 + 15)*FTtoMTRS, Next to Davidson, on 15 ft pole
           2: (2118)*FTtoMTRS , Top of Holden
        */
        PATH = 4;
        TRANS = 0;
        NoSecondEdge = 0;  /* Meaning No Leading Edge */
        NoCorners = 0;
        Printfld = 0;
        Printphs = 0;
        Printint = 0;

/* Get Input Data */
        inputdata1(TRANS,&yt,zro,xro,yro,ybldg,zxcorn);
/* Check for connected buildings */
        TCHNGBLDGS(zxcorn,ybldg,NOCORNROOF,CONNIJ,LOWCORNLIM);

/*      printf("Change integration ranges?, 1=Yes, 0=No: ");
        scanf("%d",&Resp);
         */
        Resp = 0;

        if(Resp) {
            printf("Enter Delta1: ");
            scanf("%lf",&D[0]);
            printf("Enter Delta2: ");
            scanf("%lf",&D[1]);
            }
        else {
            D[0] = 0.2;
            D[1] = 1.0;
            }

        for(pk=0;pk<=30;pk++)  { /*TRAVELING PATH */
```

```
        delp = 0.0;
        delht = 0.0;

        if(PATH ==1)
            delht = (double)pk;

        if(PATH == 2)
            delp = (double)pk/3.0;

        if(PATH == 3)
            delp = pk*FTtoMTRS;  /* pk in ft, delp in mtrs */

        if(PATH == 4)
            delht = 30.0 - (double)pk;

        if(PATH == 5)
            delp =  pk/10.0;

        if( PATH == 6)
                delp = (double)pk/3.0;

/* Locate Mobile Receiver */
        FindRcvr(PATH,delp,zro,xro,yro,R);

        if( PATH == 1 )  {
            /* BLDG EDGE RISES */
            ybldg[0] = delht;
            yt = 2.0;
            }

        if( PATH == 4 )
            /* Trans. Lowers */
            yt = delht;

        if( PATH == 5 ) {
            /* Recvr Rises */
            R[4] = R[4] + delht;
            yt = 10.0;
            }

        if( PATH == 6 )
            yt = 10.0;

/* Set Building Masks */
        for(j=0;j<=(Nb-1);j++)
            MASK[j] = 1;
        MASK[0] = 0;  /* second building, 10 m high */
        MASK[1] = 0;  /* Bldg [1] is the first bldg encountered (15 m) */
        MASK[2] = 1;  /* Davidson Hall 1 */
        MASK[3] = 1;  /* Davidson Hall 2 */
        MASK[4] = 1;  /* Davidson Hall 3 */
        MASK[5] = 1;  /* PATTON HALL */
        MASK[6] = 1;  /* Block-long, 3-D mult diff bldg */

/* Find diffracting buildings and return diffraction field */
        if( FindFieldP(0,R,zxcorn,ybldg,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                        delp,ampfac,APDUM,RIA,
                        &RC1dum,&BLOCKED,&N1dum) ) {

                if(!BLOCKED) {
                amp =  RIA[0]*RIA[0] + RIA[1]*RIA[1] ;
                phs = atan2( RIA[1],RIA[0] );
                phsdeg = phs*180.0/pi;
                ampdb = 10.0 * log10(amp) + ampfac;
```

```
                ampmndb = 10.0 * log10(RIA[2]) + ampfac;
                r = (R[4]-yt)*(R[4]-yt) + R[0]*R[0] + R[1]*R[1];
                ampfsdb = 10.0*log10(1.0/r) + ampfac;
                ampdb = ampdb - ampfsdb;
                ampmndb = ampmndb - ampfsdb;
                }
                }
        else {
                r = (R[4]-yt)*(R[4]-yt) + R[0]*R[0] + R[1]*R[1];
                ampfsdb = 10.0*log10(1.0/r) + ampfac;
                ampdb = 0.0;
                ampmndb = 0.0;
                RIA[0] = 0.0;
                RIA[1] = 0.0;
                }

        if(BLOCKED) printf("BLOCKED\n");

                printf("pk = %d\n",pk);
                printf("DIFFRACTED TOTAL (Rel Pow): rl,im,pow,mean: %lf %lf %lf %lf\n\n"
                        RIA[0],RIA[1],ampdb,ampmndb);
                fprintf(fileg," %.3lf  %.3lf  %.3lf  %.3lf %.3lf\n",
                        delp,ampfsdb,ampdb,ampmndb,phsdeg);

        }    /* END LOOP THROUGH POINTS ALONG PATH */
        fclose(filef);
        fclose(fileg);
        fclose(filep);
        fclose(filephs);
        fclose(fileint);
        fclose(filefld);
        for(i=0;i<=3;i++) {
                fclose(file[i]); }
        return;
}


/*************************************************************************
* FUNCTION: int FindFieldP
* DESCRIPTION: Finds the diffraction field at point P. First searches for
        and identifies the nearest diffracting building, then calls routines
        to compute the diffraction caused by that building. P may be the
        receiver, or the first intermediate point on a diffracting edge.
        Returns 0 if no diffracting buildings are found, 1 otherwise.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RoofCorn: identifies point P as (0:Receiver),(1:Roof Edge),(2:Corner)
        P: observation point
        zxcorn: building coordinates
        ybldg: building heights
        lam: wavelength
        D: sampling intervals for aperture fields
        yt: transmitter height
        NOCORNROOF: flags for edges that cannot diffract (don't exist)
        LOWCORNLIM: lower limits of corners
        MASK: flags for disregard of buildings
        CONNIJ: identifies connections between buildings
        delp: distance along receiver path
        ampfac: amplitude path loss factor
* OUTPUTS:
  If P is the receiver point, then the field at P (RIA) is computed;
  If P is the first intermediate point in an aperture, then the
  field distribution in the aperture is sampled (APDAT).
        APDAT: amplitude and phase samples in the aperture at P
```

```
        RIA: field at P
        *RCout: type of prior diffracting edge
        *BLOCKED: flag indicating blockage (no diffraction field)
        *N1out: number of field samples in Region 1 of aperture
* CALLED BY:  main, BldgDiff2, CornDiff1
* FUNCTION CALLS:
        Sift: finds all the buildings obstructing the horizontal projection
                of the radio path
        NearestDiffBldg: identifies the nearest diffracting building
        BldgDiff1: considers the nearest edges of the building
*************************************************************************/

int FindFieldP(int RoofCorn, ZXUVY P, BAC zxcorn, BAH ybldg,
                double lam, double D[2], double yt, BACF NOCORNROOF,
                LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
                double delp, double ampfac, DA APDAT, double RIA[3],
                int *RCout, int *BLOCKED, int *N1out)
{
        double umax[Nb];        /* Maximum u coordinates of each building */
        BAC1 uvcorn;            /* (u,v) coord's of the corners of
                                        a single building; 0: u,  1: v  */
        BAF OBSTRUCT;           /* True for bldgs that obstruct in horiz plane*/
        BAF TOOSHORT;           /* True for bldgs too short to truly obstruct*/
        int jnear;              /* Index of nearest diffracting building */
        int k;                  /* Loop counter */
        int NOBLDGS;            /* Flag: no diffracting buildings */
        int ConnB;              /* Flag: connected building */
        int BLOCKED1;           /* Flag: diffraction field blocked */
        int RC1;                /* Prior diffracting edge, 0: none
                                                           1: roof edge
                                                           2: corner    */
        int N1new;              /* Number of samples in Region 1 of aperture */

/* Initialization */
        ConnB = 0;
        *BLOCKED = 0;
        for(k=0;k<=(Nb-1);k++) {
                TOOSHORT[k] = 0;
                }
        P[2] = sqrt( P[0]*P[0] + P[1]*P[1] ); /* u coord of P */
        P[3] = 0.0; /* v coord of P */

/* Find diffracting building */
        Sift(P,zxcorn,MASK,OBSTRUCT,umax,&NOBLDGS);
        if( NOBLDGS )   {
                *RCout = 0;
                return(0);
                }
        NearestDiffBldg(P,yt,ybldg,lam,zxcorn,umax,
                OBSTRUCT,TOOSHORT,uvcorn,&NOBLDGS,&jnear);
        if( NOBLDGS ) {
                *RCout = 0;
                return(0);
                }
/* Find diffraction field */
        BldgDiff1(jnear,RoofCorn,P,zxcorn,ybldg,uvcorn,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,ConnB,
                        delp,ampfac,APDAT,RIA,&RC1,&BLOCKED1,&N1new);
        *RCout = RC1;
        *BLOCKED = BLOCKED1;
        *N1out = N1new;
        if( RC1 )  /* RC1 is zero if no diffracting edges are found */
            return(1);
        else
            return(0);
```

```
}


/*************************************************************************
* FUNCTION:  void BldgDiff1
* DESCRIPTION: Considers the diffraction past each of the two roof edges
        nearest the receiver and all of the corners, calling BldgDiff2
        to compute the diffraction by earlier edges of the same building.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        j: index of diffracting building
        RoofCorn: identifies point P as (0:Receiver),(1:Roof Edge),(2:Corner)
        P: observation point
        zxcorn: (z,x) building coordinates
        ybldg: building heights
        uvcorn: (u,v) building coordinates
        lam: wavelength
        D: sampling intervals for aperture fields
        yt: transmitter height
        NOCORNROOF: flags for edges that cannot diffract (don't exist)
        LOWCORNLIM: lower limits of corners
        MASK: flags for disregard of buildings
        CONNIJ: identifies connections between buildings
        ConnB: flag indicating connected building
        delp: distance along receiver path
        ampfac: amplitude path loss factor
* OUTPUTS:
  If P is the receiver point, then the field at P (RIA) is computed;
  If P is the first intermediate point in an aperture, then the
  field distribution in the aperture is sampled (APDAT).
        APDAT: amplitude and phase samples in the aperture at P
        RIA: field at P
        *RCout: type of prior diffracting edge
        *BLOCKED: flag indicating blockage (no diffraction field)
        *N1out: number of field samples in Region 1 of aperture
* CALLED BY: FindFieldP, BldgDiff1, BldgDiff2
* FUNCTION CALLS:
        InitAMPTMP: initializes APTMP
        SortBldg1: sorts building edges according to diffraction order
        checkFZ2: checks for explicit Fresnel zone blockage by either of the
                  two nearest building faces; computes diffraction parameters
        checkNotBlock: checks for blockage of diffraction field
        FindLowerEdgeConn: find upper edge of a connected pair
        FindUpperEdgeConn: find lower edge of a connected pair
        checkintermRoof: checks for false diffracting edge
        BldgDiff2: considers prior edges of same building
        computediff: computes diffraction
        BldgCoorduv1: computes (u,v) coord's of connected building
        BldgDiff1: re-calls present fcn for connected building
        FillAmptmp: fills APTMP with contents of APDAT
        SumRoofDiff: combines roof edge diffraction fields
        CornDiff: finds corner diffraction fields
        ChooseDiff: chooses largest diffraction field component
*************************************************************************/

void BldgDiff1(int j, int RoofCorn, ZXUVY P, BAC zxcorn, BAH ybldg,
               BAC1 uvcorn, double lam, double D[2], double yt,
               BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK,
               BACF CONNIJ, int ConnB,
               double delp, double ampfac,
               DA APDAT, double RIA[3],
               int *RCout, int *BLOCKED, int *N1out)
{
        DA2 APTMP;              /* sampled aperture fields for both edges */
```

```
        BAC1 uvcConn;           /* connected building coordinates */
        double RItmp[2][2];     /* field contributions from both edges */
        double RIAroof[3];      /* combined field due to roof edges */
        double RIAcorn[3];      /* combined field due to corners */
        ZXUVY Q1,Q2,Q;          /* intermediate points on prior edges */
        double rso[2],rpo[2],y1[2],v1[2],v2[2]; /* diffraction parameters
                                                   for both edges */
        double amp,phs;         /* amplitude and phase */
        double apcomp[2][Nc];   /* amp and phs for each field component */
        int rcnt,kcnt;          /* counters for roof and corner components */
        int BLCKDEDGE[2],BLCKDCRN,BLOCKED1; /* flags for field blockage */
        int BLSZ;               /* disregard field because SZ too small */
        int Blev;               /* level of diffraction field, 1: leading edge
                                                               2: trailing edge*/
        int i,n;                /* loop counters */
        int RC1;                /* type of prior diffracting edge */
        int ConnBL;             /* local flag for connected building */
        int EDGE[3][2];         /* ordered indices of roof edges of each level*/
        int CORN[2][4];         /* ordered indices of corner edges */
        int nedge[3];           /* number of roof edges of each level */
        int ncorn[2];           /* number of corners each side of SP */
        int N1tmp[2],N1roof,N1corn,N1new,N1; /* Region 1 samples */
        int iConn,jConn,iConnQ; /* connected edge identifiers,
                                      i: Edge index, j: Bldg index */

/* Initialization */
        InitAMPTMP(APTMP);
        MASK[j] = 1; /* mask the present building, avoid double jeopardy */
        rcnt = 0;
        kcnt = 0;
        for(n=0;n<=3;n++) {
            apcomp[0][n] = -150.0;
            apcomp[1][n] = 0.0;
            }
        BLCKDEDGE[0] = 0;
        BLCKDEDGE[1] = 0;
        BLCKDCRN = 0;
        BLOCKED1 = 0;
        *BLOCKED = 0;
        ConnBL = 0;
        Blev = 2;

        SortBldg1(uvcorn,EDGE,CORN,nedge,ncorn);
        if( checkFZ2(Blev,P,yt,EDGE,nedge,uvcorn,ybldg[j],lam,
                        Q1,Q2,rso,rpo,y1,v1,v2) ) {
            for(i=0;i<=nedge[Blev];i++) {
                for(n=0;n<=4;n++) {
                    if(!i)
                        Q[n] = Q1[n];
                    else
                        Q[n] = Q2[n];
                    }
                if( ( checkNotBlock(rso[i],rpo[i],y1[i],
                                v1[i],v2[i],lam) ) &&
                     ( !NOCORNROOF[1][ EDGE[2][i] ][j] ) ) {

                  if( (NOCORNROOF[1][ EDGE[1][0] ][j]) ||
                      (NOCORNROOF[1][ EDGE[1][1] ][j]) ) {

/* If,at the NEXT level, the edge is connected to a higher edge,
        the present edge may be rendered inconsequential */
                    FindLowerEdgeConn(NOCORNROOF,EDGE,j,&iConnQ);
                    FindUpperEdgeConn(CONNIJ,iConnQ,j,&iConn,&jConn);
                    if( checkintermRoof(P,Q,ybldg,lam,zxcorn,
                                iConn,jConn,EDGE[Blev][i],j) ) {
```

```
                        /* Present edge is not inconsequential */
/*OUTPUT*/
        printf("RoofEdge: RoofCorn= %d  Blev= %d  i,j= %d %d\n",
                RoofCorn,Blev,EDGE[Blev][i],j);
        printf("    rso= %.3lf rpo= %.3lf y1= %.3lf  v1= %.3lf  v2= %.3lf\n",
                rso[i],rpo[i],y1[i],v1[i],v2[i]);
        fprintf(file[rcnt],"%.3lf  %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                delp,rso[i],rpo[i],y1[i],v1[i],v2[i]);
/* END OUTPUT*/

                        BldgDiff2(j,0,1,Q,zxcorn,ybldg,lam,D,yt,
                              NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,ConnB,
                              EDGE,CORN,nedge,ncorn,
                              delp,ampfac,APDAT,RIA,
                              &RC1,&BLOCKED1,&N1);
                        BLCKDEDGE[i] = BLOCKED1;
                        if(!BLOCKED1)  /* THEN: fill APTMP and rcnt++ */
                          computediff(RoofCorn,RC1,0,P[4],
                              rso[i],rpo[i],y1[i],v1[i],v2[i],
                              Q[2],lam,N1,D,P[2],ybldg[j],yt,APDAT,
                              /*OUTPUTS*/ APTMP,RItmp,&rcnt,N1tmp,&BLSZ);
                        if( BLSZ )
                          BLCKDEDGE[i] = 1;
                        }

                       else {  /* !checkinterm, Ignore Blev1, go to jConn*/
                          ConnBL = 2;
                          BldgCoorduv1(P,jConn,zxcorn,uvcConn);
                          BldgDiff1(jConn,RoofCorn,P,zxcorn,ybldg,
                              uvcConn,lam,D,yt,
                              NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                              ConnBL,delp,ampfac,
                              APDAT,RIA,&RC1,&BLOCKED1,&N1new);
                          FillAmptmp(APDAT,APTMP,rcnt);
                          RItmp[0][rcnt] = RIA[0];
                          RItmp[1][rcnt] = RIA[1];
                          N1tmp[rcnt] = N1new;
                          BLCKDEDGE[i] = BLOCKED1;
                          rcnt++;
                          }  /*end else not checkinterm */
                       } /* end if NOCORNROOF */

                  else {   /* !NOCORNROOF: Begin usual case */
/*OUTPUT*/
        printf("RoofEdge: RoofCorn= %d  Blev= %d  i,j= %d %d\n",
                RoofCorn,Blev,EDGE[Blev][i],j);
        printf("    rso= %.3lf rpo= %.3lf y1= %.3lf  v1= %.3lf  v2= %.3lf\n",
                rso[i],rpo[i],y1[i],v1[i],v2[i]);
        fprintf(file[rcnt],"%.3lf  %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                delp,rso[i],rpo[i],y1[i],v1[i],v2[i]);
/* END OUTPUT*/

                        /* Find prior diffractions */
                        BldgDiff2(j,0,1,Q,zxcorn,ybldg,lam,D,yt,
                              NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,ConnB,
                              EDGE,CORN,nedge,ncorn,
                              delp,ampfac,APDAT,RIA,
                              &RC1,&BLOCKED1,&N1);
                        BLCKDEDGE[i] = BLOCKED1;
                        if(!BLOCKED1) /* THEN compute diffraction field at P,
                                         AMPTMP filled and rcnt++ */
                          computediff(RoofCorn,RC1,0,P[4],
                              rso[i],rpo[i],y1[i],v1[i],v2[i],
                              Q[2],lam,N1,D,P[2],ybldg[j],yt,APDAT,
                              /*OUTPUTS*/ APTMP,RItmp,&rcnt,N1tmp,&BLSZ);
```

```
                        if( BLSZ )
                          BLCKDEDGE[i] = 1;
                        } /* end else !NOCORNROOF */

                    } /* end if not blocked */

                else /* Blocked */
                        BLCKDEDGE[i] = 1;
                } /* end for i=0,nedge */

/* Combine roof diffraction fields */
            SumRoofDiff(RoofCorn,APTMP,RItmp,rcnt,N1tmp,RIAroof,&N1roof);

            if(!NoCorners && !ConnB)
/* Find corner diffraction fields */
                CornDiff(j,0,RoofCorn,P,zxcorn,ybldg,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                        EDGE,CORN,nedge,ncorn,
                        delp,ampfac,APDAT,RIAcorn,
                        &kcnt,apcomp,&BLCKDCRN,&N1corn);
            else
                BLCKDCRN = 1;

/* Select dominant field contribution */
            ChooseDiff(rcnt,kcnt,RoofCorn,
                APTMP,APDAT,RIAroof,RIAcorn,
                N1roof,N1corn,RIA,&RC1,&N1new);
            *RCout = RC1;
            *N1out = N1new;
            if( ConnB == 2 ) /* If this is a connected building, don't
                                output field components */
                return;

            if(!RoofCorn) {
/* If at the receiver point (RoofCorn=0), output field components */
              for(n=0;n<=(rcnt-1);n++) {
                if( (RItmp[0][n] != 0.0) || (RItmp[1][n] != 0.0) )  {
                  amp =  RItmp[0][n]*RItmp[0][n] + RItmp[1][n]*RItmp[1][n];
                  phs = atan2( RItmp[1][n],RItmp[0][n] );
                  apcomp[1][n] = phs*180.0/pi;
                  apcomp[0][n] = 10.0 * log10(amp) + ampfac;
                  printf("ROOF EDGE SUB-TOTAL: rl,im,pow,phs: %lf %lf %lf %lf\n",
                        RItmp[0][n],RItmp[1][n],apcomp[0][n],apcomp[1][n]);
                  }
                } /* end for n */
                fprintf(filef," %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                    delp,apcomp[0][0],apcomp[0][1],apcomp[0][2],apcomp[0][3]);
                fprintf(filep," %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                    delp,apcomp[1][0],apcomp[1][1],apcomp[1][2],apcomp[1][3]);
              } /* end if !RoofCorn */
            if( BLCKDCRN && BLCKDEDGE[0] )
                if( nedge[Blev] > 0 ) {
                    if( BLCKDEDGE[1] )
                        *BLOCKED = 1;
                    }
                else
                    *BLOCKED = 1;
            } /* end if checkFZ2 */

        else /* No Diffraction at this level, look at earlier edges */ {
            BldgDiff2(j,1,RoofCorn,P,zxcorn,ybldg,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,ConnB,
                        EDGE,CORN,nedge,ncorn,
                        delp,ampfac,APDAT,RIA,
                        &RC1,&BLOCKED1,&N1new);
```

```
            *BLOCKED = BLOCKED1;
            *RCout = RC1;
            *N1out = N1new;
            }

        return;
}


/*************************************************************************
* FUNCTION:  void BldgDiff2
* DESCRIPTION: Considers the diffraction past each of the two roof edges
        farthest from the receiver, considers the corners if not already
        considered. Goes back to FindFieldP to find prior diffracting
        buildings.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        j: index of diffracting building
        NOBlev2: no diffraction by trailing edges
        RoofCorn: identifies point P as (0:Receiver),(1:Roof Edge),(2:Corner)
        P: observation point
        zxcorn: (z,x) building coordinates
        ybldg: building heights
        lam: wavelength
        D: sampling intervals for aperture fields
        yt: transmitter height
        NOCORNROOF: flags for edges that cannot diffract (don't exist)
        LOWCORNLIM: lower limits of corners
        MASK: flags for disregard of buildings
        CONNIJ: identifies connections between buildings
        ConnB: flag indicating connected building
        EDGE: indices of ordered roof edges
        CORN: indices of ordered corners
        nedge: number of roof edges of each level
        ncorn: number of corners, each side of Source-P line
        delp: distance along receiver path
        ampfac: amplitude path loss factor
* OUTPUTS:
  If P is the receiver point, then the field at P (RIA) is computed;
  If P is the first intermediate point in an aperture, then the
  field distribution in the aperture is sampled (APDAT).
        APDAT: amplitude and phase samples in the aperture at P
        RIA: field at P
        *RCout: type of prior diffracting edge
        *BLOCKED: flag indicating blockage (no diffraction field)
        *N1out: number of field samples in Region 1 of aperture
* CALLED BY: BldgDiff1
* FUNCTION CALLS:
        InitAMPTMP: initializes APTMP
        BldgCoorduv1: computes (u,v) coord's of connected building
        checkFZ2: checks for explicit Fresnel zone blockage by either of the
                  two nearest building faces; computes diffraction parameters
        FindUpperEdgeConn: find lower edge of a connected pair
        BldgDiff1: considers trailing edges of connected building
        FillAmptmp: fills APTMP with contents of APDAT
        checkNotBlock: checks for blockage of diffraction field
        FindFieldP: looks for prior diffracting buildings
        computediff: computes diffraction
        SumRoofDiff: combines roof edge diffraction fields
        CornDiff: finds corner diffraction fields
        ChooseDiff: chooses largest diffraction field component
*************************************************************************/

void BldgDiff2(int j, int NOBlev2, int RoofCorn, ZXUVY P, BAC zxcorn,
```

```
                BAH ybldg, double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK,
                BACF CONNIJ, int ConnB,
                int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
                double delp, double ampfac,
                DA APDAT, double RIA[3], int *RCout, int *BLOCKED, int *N1out)
{
        DA2 APTMP;              /* sampled aperture fields for both edges */
        double RItmp[2][2];     /* field contributions from both edges */
        double RIAroof[3];      /* combined field due to roof edges */
        double RIAcorn[3];      /* combined field due to corners */
        ZXUVY Q1,Q2,Q;          /* intermediate points on prior edges */
        double rso[2],rpo[2],y1[2],v1[2],v2[2]; /* diffraction parameters
                                                   for both edges */
        BAC1 uvcorn;            /* (u,v) coord's of present building corners */
        BAC1 uvcConn;           /* (u,v) coord's of connected building */
        double amp,phs;         /* amplitude and phase */
        double apcomp[2][Nc];   /* amp and phs for each field component */
        int rcnt,kcnt;          /* counters for roof and corner components */
        int BLCKDEDGE[2],BLCKDCRN,BLOCKED1; /* flags for field blockage */
        int BLSZ;               /* disregard field because SZ too small */
        int Blev;               /* level of diffraction field, 1: leading edges
                                                               2: trailing edges*/
        int i,n;                /* loop counters */
        int RC1;                /* type of prior diffracting edge */
        int ConnBL;             /* local flag for connected building */
        int N1tmp[2],N1roof,N1corn,N1new,N1; /* Region 1 samples */
        int iConn,jConn;        /* connected edge identifiers,
                                        i: Edge index, j: Bldg index */

/* Initialization */
        InitAMPTMP(APTMP);
        BLCKDEDGE[0] = 0;
        BLCKDEDGE[1] = 0;
        BLCKDCRN = 0;
        BLOCKED1 = 0;
        *BLOCKED = 0;
        rcnt = 0;
        kcnt = 0;
        Blev = 1;
        P[2] = sqrt( P[0]*P[0] + P[1]*P[1] );
        P[3] = 0.0;

        BldgCoorduv1(P,j,zxcorn,uvcorn);
        if(!NoSecondEdge && checkFZ2(Blev,P,yt,EDGE,nedge,uvcorn,ybldg[j],lam,
                        Q1,Q2,rso,rpo,y1,v1,v2) ) {
            for(i=0;i<=nedge[Blev];i++) {
                for(n=0;n<=4;n++) {
                    if(!i)
                        Q[n] = Q1[n];
                    else
                        Q[n] = Q2[n];
                    }
                if( NOCORNROOF[1][ EDGE[Blev][i] ][j] ) {

/* Find connected building, consider diffraction by that bldg (index jConn) */
                    FindUpperEdgeConn(CONNIJ,EDGE[Blev][i],j,&iConn,&jConn);
                    BldgCoorduv1(P,jConn,zxcorn,uvcConn);
                    ConnBL = 1;
                    BldgDiff1(jConn,RoofCorn,P,zxcorn,ybldg,uvcConn,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,ConnBL,delp,ampfac,
                        APDAT,RIA,&RC1,&BLOCKED1,&N1new);
                    FillAmptmp(APDAT,APTMP,rcnt);
                    RItmp[0][rcnt] = RIA[0];
                    RItmp[1][rcnt] = RIA[1];
```

```
                    N1tmp[rcnt] = N1new;
                    BLCKDEDGE[i] = BLOCKED1;
                    rcnt++;
                    }

                else {   /* !NOCORNROOF */
                    if( checkNotBlock(rso[i],rpo[i],y1[i],
                                v1[i],v2[i],lam) ) {
/*OUTPUT*/
        printf("RoofEdge: RoofCorn= %d  Blev= %d  i,j= %d %d\n",
                RoofCorn,Blev,EDGE[Blev][i],j);
        printf("    rso= %.3lf rpo= %.3lf y1= %.3lf  v1= %.3lf  v2= %.3lf\n",
                rso[i],rpo[i],y1[i],v1[i],v2[i]);
        fprintf(file[rcnt],"%.3lf  %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                delp,rso[i],rpo[i],y1[i],v1[i],v2[i]);
/* END OUTPUT*/
                        if( FindFieldP(1,Q,zxcorn,ybldg,lam,D,yt,
                                NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                                delp,ampfac,APDAT,RIA,
                                &RC1,&BLOCKED1,&N1) ) ;
                        BLCKDEDGE[i] = BLOCKED1;
                        if(!BLOCKED1)
                            computediff(RoofCorn,RC1,0,P[4],
                                rso[i],rpo[i],y1[i],v1[i],v2[i],
                                Q[2],lam,N1,D,P[2],ybldg[j],yt,APDAT,
                                /*OUTPUTS*/ APTMP,RItmp,&rcnt,N1tmp,&BLSZ);
                        if( BLSZ )
                            BLCKDEDGE[i] = 1;
                        } /* if not blocked */
                    else
                        BLCKDEDGE[i] = 1;
                    }  /* end else !NOCORNROOF */
                }  /* end for i=0,nedge */

/* Combine roof diffraction fields */
            SumRoofDiff(RoofCorn,APTMP,RItmp,rcnt,N1tmp,RIAroof,&N1roof);

            if( NOBlev2 && !NoCorners && !ConnB )
/* Find corner diffraction field if there was no diffraction at the
    trailing edges in BldgDiff1 (then corners haven't been considered yet) */
                CornDiff(j,1,RoofCorn,P,zxcorn,ybldg,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                        EDGE,CORN,nedge,ncorn,
                        delp,ampfac,APDAT,RIAcorn,
                        &kcnt,apcomp,&BLCKDCRN,&N1corn);

/* Select dominant field component */
            ChooseDiff(rcnt,kcnt,RoofCorn,
                APTMP,APDAT,RIAroof,RIAcorn,
                N1roof,N1corn,RIA,&RC1,&N1new);
            *RCout = RC1;
            *N1out = N1new;
            if( NOBlev2 && !RoofCorn) {
/* If P is receiver and no diffraction at trailing edges, print outputs */
              for(n=0;n<=(rcnt-1);n++) {
                if( (RItmp[0][n] != 0.0) || (RItmp[1][n] != 0.0) )  {
                  amp =  RItmp[0][n]*RItmp[0][n] + RItmp[1][n]*RItmp[1][n];
                  phs = atan2( RItmp[1][n],RItmp[0][n] );
                  apcomp[1][n] = phs*180.0/pi;
                  apcomp[0][n] = 10.0 * log10(amp) + ampfac;
                  printf("ROOF EDGE SUB-TOTAL: rl,im,pow,phs: %lf %lf %lf %lf\n",
                        RItmp[0][n],RItmp[1][n],apcomp[0][n],apcomp[1][n]);
                  }
                } /* end for n */
                fprintf(filef," %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
```

```
                delp,apcomp[0][0],apcomp[0][1],apcomp[0][2],apcomp[0][3]);
            fprintf(filep," %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                delp,apcomp[1][0],apcomp[1][1],apcomp[1][2],apcomp[1][3]);
          } /* end if NOBlev2 and !RoofCorn */

        if( BLCKDCRN && BLCKDEDGE[0] )
            if( nedge[Blev] > 0 ) {
                if( BLCKDEDGE[1] )
                    *BLOCKED = 1;
                }
            else
                *BLOCKED = 1;
        }

      else /* No Diffraction at this level, check for more buildings */ {
            if( FindFieldP(RoofCorn,P,zxcorn,ybldg,lam,D,yt,
                      NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                      delp,ampfac,APDAT,RIA,
                      &RC1,&BLOCKED1,&N1new)  )
              ;
            *RCout = RC1;
            *BLOCKED = BLOCKED1;
            *N1out = N1new;
          }
      return;
}


/*************************************************************************
* FUNCTION:  void CornDiff
* DESCRIPTION: Considers two corner diffraction fields: one around the left
      of the building and one around the right. For each side of the
      building, looks for diffraction at nearest corner with CornDiff1.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
      j: index of diffracting building
      NOBlev2: no diffraction trailing edges
      RoofCorn: identifies point P as (0:Receiver),(1:Roof Edge),(2:Corner)
      P: observation point
      zxcorn: (z,x) building coordinates
      ybldg: building heights
      lam: wavelength
      D: sampling intervals for aperture fields
      yt: transmitter height
      NOCORNROOF: flags for edges that cannot diffract (don't exist)
      LOWCORNLIM: lower limits of corners
      MASK: flags for disregard of buildings
      CONNIJ: identifies connections between buildings
      EDGE: indices of ordered roof edges
      CORN: indices of ordered corners
      nedge: number of roof edges of each level
      ncorn: number of corners, each side of Source-P line
      delp: distance along receiver path
      ampfac: amplitude path loss factor
* OUTPUTS:
  If P is the receiver point, then the field at P (RIA) is computed;
  If P is the first intermediate point in an aperture, then the
  field distribution in the aperture is sampled (APDAT).
      APDAT: amplitude and phase samples in the aperture at P
      RIA: field at P
      *kcnt: number of corner diffracting edges
      apcomp: amplitude and phase of field components
      *BLCKDCRN: flag indicating blockage (no diffraction field)
      *N1corn: number of field samples in Region 1 of aperture
* CALLED BY: BldgDiff1, BldgDiff2
```

```
* FUNCTION CALLS:
        InitAMPTMP: initializes APTMP
        InitRItmp: initializes RIA
        CornDiff1: finds diffraction by nearest corner on one side
        SumCornDiff: combines two corner diffraction fields
*************************************************************************/

void CornDiff(int j, int NOBlev2, int RoofCorn, ZXUVY P, BAC zxcorn,
                BAH ybldg, double lam, double D[2], double yt,
                BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
                int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
                double delp, double ampfac,
                DA APDAT, double RIA[3], int *kcnt,
                double apcomp[2][Nc], int *BLCKDCRN, int *N1corn)
{
        DA2 APTMP;              /* sampled aperture fields for both edges */
        double RItmp[2][2];     /* field contributions from both sides */
        double amp,phs;         /* amplitude and phase */
        int klr;                /* loop counter, left and right sides */
        int kc;                 /* number of corners on a particular side */
        int BLOCKED[2],BLCKDCRN1; /* flags for field blockage */
        int RC1;                /* type of prior diffracting edge */
        int kcnt1;              /* number of sides not blocked */
        int n,n2;               /* loop counter, component index */
        int N1tmp[2],N1new;     /* Region 1 samples */

/* Initialization */
        InitAMPTMP(APTMP);
        InitRItmp(RItmp);
        BLOCKED[0] = 0;
        BLOCKED[1] = 0;
        BLCKDCRN1 = 0;
        *BLCKDCRN = 0;
        kcnt1 = 0;
        if( ncorn[0] < 0 )
            BLOCKED[0] = 1;

        for(klr=0;klr<=1;klr++) {
            kc = ncorn[klr];
            if( kc >= 0 ) {
                if( NOBlev2 && (kc==2) )
                    kc = 1;
                CornDiff1(j,NOBlev2,RoofCorn,P,zxcorn,ybldg,
                        klr,kc,lam,D,yt,
                        NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                        EDGE,CORN,nedge,ncorn,kcnt1,1,
                        delp,ampfac,APTMP,RItmp,
                        &BLCKDCRN1,&RC1,&N1new);
                BLOCKED[klr] = BLCKDCRN1;
                N1tmp[kcnt1] = N1new;
                if( !BLCKDCRN1 )
                    kcnt1++;
                } /* end if kc >= 0 */

            } /* end for klr=0,1 */

        if( BLOCKED[0] && BLOCKED[1] ) {
            *BLCKDCRN = 1;
            *kcnt = 0;
            }

        else {

/* Combine corner diffraction fields */
            SumCornDiff(RoofCorn,APTMP,RItmp,kcnt1,N1tmp,APDAT,RIA,&N1new);
```

```
            *N1corn = N1new;
            *kcnt = kcnt1;
            }

        if(!RoofCorn) {  /* Print Outputs */
              for(n=0;n<=1;n++) {
                  apcomp[0][n+2] = -150.0;
                  apcomp[1][n+2] = 0.0;
                  }
              for(n=0;n<=(kcnt1-1);n++) {
                if( (RItmp[0][n] != 0.0) || (RItmp[1][n] != 0.0) )  {
                  amp =  RItmp[0][n]*RItmp[0][n] + RItmp[1][n]*RItmp[1][n];
                  phs = atan2( RItmp[1][n],RItmp[0][n] );
                  n2 = n+2;
                  if( BLOCKED[0] ) n2++;
                  apcomp[1][n2] = phs*180.0/pi;
                  apcomp[0][n2] = 10.0 * log10(amp) + ampfac;
                  printf("CORNER SUB-TOTAL: rl,im,pow,phs: %lf %lf %lf %lf\n",
                        RItmp[0][n],RItmp[1][n],apcomp[0][n2],apcomp[1][n2]);
                  }
                } /* ENDFOR */
              } /*ENDIF*/
        return;
}


/*************************************************************************
* FUNCTION:  void CornDiff1
* DESCRIPTION: Finds the diffraction by a single corner, calls itself
        to consider prior corners and calls FindFieldP to find
        prior diffracting buildings
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        j: index of diffracting building
        NOBlev2: no diffraction trailing edges
        RoofCorn: identifies point P as (0:Receiver),(1:Roof Edge),(2:Corner)
        P: observation point
        zxcorn: (z,x) building coordinates
        ybldg: building heights
        klr: 0: left-side diffraction field; 1: right-side
        kc: number of corners on the klr side of the building
        lam: wavelength
        D: sampling intervals for aperture fields
        yt: transmitter height
        NOCORNROOF: flags for edges that cannot diffract (don't exist)
        LOWCORNLIM: lower limits of corners
        MASK: flags for disregard of buildings
        CONNIJ: identifies connections between buildings
        EDGE: indices of ordered roof edges
        CORN: indices of ordered corners
        nedge: number of roof edges of each level
        ncorn: number of corners, each side of Source-P line
        kcnt1: counter of corner field components
        FirstCorn: true for first corner on each side
        delp: distance along receiver path
        ampfac: amplitude path loss factor
* OUTPUTS:
  If P is the receiver point, then the field at P (RIA) is computed;
  If P is the first intermediate point in an aperture, then the
  field distribution in the aperture is sampled (APDAT).
        APTMP: amplitude and phase samples in the aperture at P
        RItmp: field at P
        *BLOCKED: flag indicating blockage (no diffraction field)
        *RCout: type of prior diffracting edge
        *N1out: number of field samples in Region 1 of aperture
```

```
* CALLED BY: CornDiff, CornDiff1
* FUNCTION CALLS:
        InitAMPDAT: initializes APDAT
        BldgCoorduv1: finds (u,v) building coord's
        PntCoordzx1: finds (z,x) coord's of point Q
        checkFZcorn: checks Fresnel zone clearance of the corner
        checkNotBlock: checks for field blockage
        FindCornConn: finds building with connected corner
        SortBldg1: sorts building edges
        CornDiff1: considers prior corners or corners of connected buildings
        checkintermCorn: checks for false diffracting corner
        FillAmpdat: fills APDAT with contents of APTMP
        computediff: computes diffraction field at P
        FillAmptmp: fills APTMP with contents of APDAT
        FindFieldP: looks for prior diffracting buildings
*************************************************************************/

void CornDiff1(int j, int NOBlev2, int RoofCorn, ZXUVY P,
               BAC zxcorn, BAH ybldg, int klr, int kc,
               double lam, double D[2], double yt,
               BACF NOCORNROOF, LCL LOWCORNLIM, BAF MASK, BACF CONNIJ,
               int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2],
               int kcnt1, int FirstCorn, double delp, double ampfac,
               DA2 APTMP, double RItmp[2][2],
               int *BLOCKED, int *RCout, int *N1out)
{
        int iCorn;              /* index of present corner */
        int kcnt;               /* counter of field components */
        int BLOCKED1;           /* flag for field blockage */
        int RC1;                /* type of prior diffracting edge */
        int BLSZ;               /* field not considered because SZ too small */
        int N1,N1new,N1tmp[2];  /* Region 1 samples */
        int EConn[3][2],CConn[2][4],neConn[3],ncConn[2];
                                /* ordered edge indices for connected bldg */
        int jConn;              /* index of connected building */
        DA APDAT;               /* sampled aperture field */
        ZXUVY Q;                /* intermediate point on prior edge */
        BAC1 uvcorn;            /* (u,v) corner coordinates */
        double rso,rpo,y1,v1,v2;/* diffraction parameters */
        double RIAdum[3];       /* dummy field array */

/* Initialization */
        InitAMPDAT(APDAT);
        P[2] = sqrt( P[0]*P[0] + P[1]*P[1] );
        P[3] = 0.0;

        BldgCoorduv1(P,j,zxcorn,uvcorn);
        if(kc>=0) { /* If another corner exists on this side */
            iCorn = CORN[klr][kc];
            Q[2] = uvcorn[0][iCorn];
            Q[3] = uvcorn[1][iCorn];
            PntCoordzx1(P,Q);
            if( checkFZcorn(P,Q,yt,lam,LOWCORNLIM[iCorn][j],klr,
                        &rso,&rpo,&y1,&v1,&v2) || FirstCorn ) {
/* If the corner introduces Fresnel zone blockage, OR is the first corner
   on a side (we can't ignore the corner diffraction field contribution
   if the building is being considered a diffractor, just because the corner
   provides a lot of clearance) */

                    if( checkNotBlock(rso,rpo,y1,v1,v2,lam) ) {
/* If the field is not severely blocked */

                        if( NOCORNROOF[0][iCorn][j] ) {
/* If the corner is connected to a higher building */
                            FindCornConn(zxcorn,iCorn,j,&jConn);
```

```
                        BldgCoorduv1(P,jConn,zxcorn,uvcorn);
                        SortBldg1(uvcorn,EConn,CConn,neConn,ncConn);
                        CornDiff1(jConn,NOBlev2,RoofCorn,P,zxcorn,ybldg,
                            klr,kc,lam,D,yt,
                            NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                            EConn,CConn,neConn,ncConn,kcnt1,1,
                            delp,ampfac,APTMP,RItmp,
                            &BLOCKED1,&RC1,&N1new);
                        *BLOCKED = BLOCKED1;
                        *RCout = RC1;
                        *N1out = N1new;
                        return;
                        }

                    else {   /* !NOCORNFLAG: usual case */
                        if( checkintermCorn(P,Q,yt,lam,LOWCORNLIM,
                            zxcorn,CORN,klr,kc,j) ) {
                            /* if present corner still introduces
                              diffraction, even when a prior corner is
                              considered */
/*OUTPUT*/
        printf("CornEdge: RoofCorn= %d  i,j= %d %d\n",
                RoofCorn,iCorn,j);
        printf("    rso= %.3lf rpo= %.3lf y1= %.3lf  v1= %.3lf  v2= %.3lf\n",
                rso,rpo,y1,v1,v2);
        fprintf(file[kcnt1+2],"%.3lf  %.3lf  %.3lf  %.3lf  %.3lf  %.3lf\n",
                delp,rso,rpo,y1,v1,v2);
/* END OUTPUT*/
                            CornDiff1(j,NOBlev2,2,Q,zxcorn,ybldg,
                                klr,kc-1,lam,D,yt,
                                NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                                EDGE,CORN,nedge,ncorn,kcnt1,0,
                                delp,ampfac,APTMP,RItmp,
                                &BLOCKED1,&RC1,&N1);
                            if(!BLOCKED1) {
                                if(RC1)
                                    FillAmpdat(APTMP,APDAT,kcnt1);
                                kcnt = kcnt1; /* Don't let computediff
                                    increment kcnt1; can only be done
                                    in CornDiff ! */
                                computediff(RoofCorn,RC1,1,P[4],
                                    rso,rpo,y1,v1,v2,
                                    Q[2],lam,N1,D,P[2],ybldg[j],yt,APDAT,
                                    /*OUTPUTS*/ APTMP,RItmp,
                                    &kcnt,N1tmp,&BLSZ);
                                *N1out = N1tmp[kcnt1];
                                BLOCKED1 = BLSZ;
                                }
                            *BLOCKED = BLOCKED1;
                            *RCout = 2;
                            return;
                            }
                        else  /* !checkinterm: Ignore Q,
                                    go to next corner */ {
                            CornDiff1(j,NOBlev2,RoofCorn,P,zxcorn,ybldg,
                                    klr,kc-1,lam,D,yt,
                                    NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                                    EDGE,CORN,nedge,ncorn,kcnt1,0,
                                    delp,ampfac,APTMP,RItmp,
                                    &BLOCKED1,&RC1,&N1new);
                            *BLOCKED = BLOCKED1;
                            *RCout = RC1;
                            *N1out = N1new;
                            return;
                            } /* end else !checkinterm */
```

```
                        } /* end else !NOCORNROOF  */
                    }

                else /* Blocked */ {
                    *BLOCKED = 1;
                    return;
                    }

                } /* end if checkFZcorn: Free Space (no more diff edges) */

          } /* end if kc >= 0 : No more edges*/

        if( FindFieldP(RoofCorn,P,zxcorn,ybldg,lam,D,yt,
                    NOCORNROOF,LOWCORNLIM,MASK,CONNIJ,
                    delp,ampfac,APDAT,RIAdum,
                    &RC1,&BLOCKED1,&N1new)  ) {
            FillAmptmp(APDAT,APTMP,kcnt1);
            RItmp[0][kcnt1] = RIAdum[0];
            RItmp[1][kcnt1] = RIAdum[1];
            }
        *RCout = RC1;
        *BLOCKED = BLOCKED1;
        *N1out = N1new;
      return;
}


/*************************************************************************
* FUNCTION: void TCHNGBLDGS
* DESCRIPTION: Finds buildings that share an edge and marks various flags
      to identify the connected edges and point from one building edge
      to the connected building and edge. This routine need be
      performed only once for a data set.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
      zxcorn: (z,x) building coord's
      ybldg: building heights
* OUTPUTS:
      NOCORNROOF: flags for edges that cannot diffract (don't exist)
      CONNIJ: identifies connections between buildings
      LOWCORNLIM: lower limits of corners
* CALLED BY: main
* FUNCTION CALLS: none
*************************************************************************/
void TCHNGBLDGS(BAC zxcorn, BAH ybldg,
            BACF NOCORNROOF, BACF CONNIJ, LCL LOWCORNLIM)
{
      int j,i,k,m,n;          /* loop counters */
      int im1,nm1,np1,jCRN;   /* indices */

/* Initialization */
      for(j=0;j<=(Nb-1);j++) {
        for(i=0;i<=3;i++) {
          for(k=0;k<=1;k++) {
            NOCORNROOF[k][i][j] = 0;
            CONNIJ[k][i][j] = 0;
            }
          LOWCORNLIM[i][j] = -10000.0;
        }
      }

      for(j=0;j<=(Nb-1);j++) {
        for(i=0;i<=3;i++)  {
          im1 = cyc4neg(i);
          jCRN = j;
```

```
            if(!j)jCRN=99;
/* if the building index is zero, use 99 instead */
            for(m=(j+1);m<=(Nb-1);m++) {
              for(n=0;n<=3;n++)  {
                 nm1= cyc4neg(n);
                 np1 = cyc4pos(n);
                 if( (zxcorn[0][i][j]==zxcorn[0][n][m]) &&
                     (zxcorn[1][i][j]==zxcorn[1][n][m]) ) {
                     if( ybldg[m] > ybldg[j] ) {
                         NOCORNROOF[0][i][j] = 1;
                         LOWCORNLIM[n][m] = ybldg[j];
                         }
                     else  {
                         NOCORNROOF[0][n][m] = 1;
                         LOWCORNLIM[i][j] = ybldg[m];
                         }
                     if( (zxcorn[0][im1][j]==zxcorn[0][nm1][m]) &&
                         (zxcorn[1][im1][j]==zxcorn[1][nm1][m]) ) {
                         if( ybldg[m] > ybldg[j] ) {
                             NOCORNROOF[1][i][j] = 1;
                             CONNIJ[1][n][m] = jCRN;
                             CONNIJ[0][n][m] = i;
/* The value of CONNIJ[0][a][b] refers to a LOWER EDGE;
   (a,b) refers to an UPPER EDGE; and the two are connected.  */
                             }
                         else {
                             NOCORNROOF[1][n][m] = 1;
                             CONNIJ[1][i][j] = m;
                             CONNIJ[0][i][j] = n;
                             }
                         }
                     else {
                         if( (zxcorn[0][im1][j]==zxcorn[0][np1][m]) &&
                             (zxcorn[1][im1][j]==zxcorn[1][np1][m]) ) {
                             if( ybldg[m] > ybldg[j] ) {
                                 NOCORNROOF[1][i][j] = 1;
                                 CONNIJ[1][np1][m] = jCRN;
                                 CONNIJ[0][np1][m] = i;
                                 }
                             else  {
                                 NOCORNROOF[1][np1][m] = 1;
                                 CONNIJ[1][i][j] = m;
                                 CONNIJ[0][i][j] = np1;
                                 }
                             }
                         }  /* end else */
                     }  /* end if */
                 }  /* end for */
              }
            }
          }
        return;
}


/*************************************************************************
* FUNCTION: void InitAMPDAT
* DESCRIPTION: Initializes APDAT to zero
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: none
* OUTPUTS: APDAT
* CALLED BY: main, CornDiff1
* FUNCTION CALLS: none
*************************************************************************/
void InitAMPDAT(DA APDAT)
```

```
{
        int j,k;        /* loop counters */
        for(k=0;k<=1;k++)
                for(j=0;j<=(SZ-1);j++)
                    APDAT[k][j] = 0.0;
        return;
}


/*************************************************************************
* FUNCTION: InitAMPTMP
* DESCRIPTION: Initializes APTMP to zero
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: none
* OUTPUTS: APTMP
* CALLED BY: BldgDiff1, BldgDiff2, CornDiff
* FUNCTION CALLS: none
*************************************************************************/
void InitAMPTMP(DA2 APTMP)
{
        int i,j,k;      /* loop counters */
        for(k=0;k<=1;k++)
            for(i=0;i<=1;i++)
                    for(j=0;j<=(SZ-1);j++)
                        APTMP[k][i][j] = 0.0;
        return;
}


/*************************************************************************
* FUNCTION: InitRItmp
* DESCRIPTION: Initializes RItmp
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: none
* OUTPUTS: RItmp
* CALLED BY: CornDiff
* FUNCTION CALLS: none
*************************************************************************/
void InitRItmp(double RItmp[2][2])
{
        int j,k;        /* loop counters */
        for(j=0;j<=1;j++)
          for(k=0;k<=1;k++)
                RItmp[j][k] = 0.0;
        return;
}


/*************************************************************************
* FUNCTION: FindCornConn
* DESCRIPTION: Finds the index of the connected building,
        which is higher than the present building (jQ).
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        zxcorn: (z,x) building coord's
        iQ: index of corner which doesn't exist, connects to higher bldg
        jQ: index of present building
* OUTPUTS:
        *jConn: index of higher, connected, building
* CALLED BY: CornDiff1
* FUNCTION CALLS: none
*************************************************************************/
void FindCornConn(BAC zxcorn, int iQ, int jQ, int *jConn)
{
        double zQ,xQ;           /* coord's of corner Q */
```

```
        int i,j;
        zQ = zxcorn[0][iQ][jQ];
        xQ = zxcorn[1][iQ][jQ];
        for(j=0;j<=(Nb-1);j++)
          if(j != jQ)   /* if different building */
          for(i=0;i<=3;i++)
                if( (zxcorn[0][i][j] == zQ) &&
                        (zxcorn[1][i][j] ==xQ) ) {
                        *jConn = j;
                        return;
                        }
        printf("Didn't find Connected Building Corner\n");
        return;
}


/*************************************************************************
* FUNCTION: FindUpperEdgeConn
* DESCRIPTION: Finds the (higher) roof edge that connects to the present edge
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        CONNIJ: identifies connections between buildings
        iQ: index of present edge
        jQ: index of present building
* OUTPUTS:
        *iConn: index of connected edge
        *jConn: index of connected building
* CALLED BY: BldgDiff1, BldgDiff2
* FUNCTION CALLS: none
*************************************************************************/
void FindUpperEdgeConn(BACF CONNIJ, int iQ, int jQ, int *iConn, int *jConn)
{
        int i,j,jCRN;
        jCRN = jQ;
        if(jCRN == 0)jCRN = 99;
        for(i=0;i<=3;i++)
          for(j=0;j<=(Nb-1);j++)
                if( (CONNIJ[1][i][j] == jCRN) &&
                        (CONNIJ[0][i][j] == iQ) ) {
                    *iConn = i;
                    *jConn = j;
                    return;
                    }
        printf("Didn't find connected building edge\n");
        return;
}


/*************************************************************************
* FUNCTION: FindLowerEdgeConn
* DESCRIPTION: Finds the edge of the present building that is the
        lower edge of a connected pair
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        NOCORNROOF: flags for edges that don't diffract (don't exist)
        EDGE: indices for ordered roof edges
        j: index of present building
* OUTPUTS:
        *iConnQ: index of connected (lower) edge
* CALLED BY: BldgDiff1
* FUNCTION CALLS: none
*************************************************************************/
void FindLowerEdgeConn(BACF NOCORNROOF, int EDGE[3][2], int j, int *iConnQ)
{
        if( (NOCORNROOF[1][ EDGE[1][0] ][j]) )
```

```
                *iConnQ = EDGE[1][0];
        else
                *iConnQ = EDGE[1][1];
        return;
}
```

```
/*************************************************************************
* FILE NAME: MDB1.C
* CONTENTS:
        SumRoofDiff
        SumCornDiff
        ChooseDiff
        computediff
        computeYfield
        computeXfield
        FindnrEnd
        FillAmptmp
        FillAmpdat
        SmoothPhs2
* DESCRIPTION:
        This file contains upper-level diffraction computation routines
*************************************************************************/
#include <mdsrch.h>
#include <mdtypes.h>
void SumRoofDiff(int RoofCorn, DA2 APTMP,
        double RItmp[2][2], int cnt, int N1tmp[2], double RIA[3], int *N1roof);
void SumCornDiff(int RoofCorn, DA2 AMPTMP,
        double RItmp[2][2], int cnt,
        int N1tmp[2], DA AMPDAT, double RIA[3], int *N1corn);
void ChooseDiff(int rcnt, int kcnt, int RoofCorn, DA2 APTMP, DA APDAT,
        double RIAroof[3], double RIAcorn[3],
        int N1roof, int N1corn, double RIA[3], int *RCout, int *N1out);
void computediff(int RoofCorn, int RC1, int k,
        double yp, double rso, double rpo, double y1, double V1, double V2,
        double uq, /* These are the coord's of yorigin, NOT the field point */
        double lam, int N1, double D[2], double up, double ybldg1, double yt,
        DA APDAT, DA2 APTMP, double RItmp[2][2],
        int *cnt, int N1out[2], int *BLOCKEDout);
void computeYfield(int RC1, int kcnt,
        double yp, double V1, double V2,
        double uo,  /* These are the coord's of yorigin,
                                NOT the field point */
        double lam, int N1, double D[2], double up, double ybldg1, double yt,
        DA APDAT, DA2 APTMP, int *N1out, int *BLOCKEDout);
void computeXfield(int RC1, int kcnt,
        double rso1, double rpo1, double y1, double V1, double V2,
        double uq, /* These are the coord's of yorigin,
                                NOT the field point */
        double lam, int N1, double D[2], double up,
        DA APDAT, DA2 APTMP, int *N1out, int *BLOCKEDout);
double FindnrEnd(double nrStart);
void FillAmptmp(DA APDAT, DA2 APTMP, int klr);
void FillAmpdat(DA2 APTMP, DA APDAT, int klr);
void SmoothPhs2(DA2 APTMP, int kcnt, int Ntot, double y1);
/*  MDB2.C  */
extern void FldPar1(double u, double up, double yt, double yp,
        double *rso, double *rpo, double *yorigin);
extern void FldPar2(double rtot0, double rso0, double u, double yp,
        double yt, double y, double *rso, double *rpo, double *yorigin);
/*  EVALFLD.C  */
extern void EvalField(double rso, double rpo, double y1,
        double x1, double x2, double lam, int N1, double D[2], DA APDAT,
        double *real, double *imag, double *ampdb, double *phs);
extern void fieldcomp(double rso, double rpo, double y1,
          double x1, double x2, double lam,
          double *real, double *imag, double *ampdb, double *phs);
extern FILE *filephs;
extern int Printphs;

/*************************************************************************
* FUNCTION: SumRoofDiff
```

```
* DESCRIPTION: Combines the fields due to the (maximum) two roof edges.
        If P is the receiver, both the complex phasor sum and the power
        sum (mag squared of the mean field strength computed as the sum
        of the squares of the component field strengths) are computed
        and stored in RIA. If P is the first intermediate point on
        a diffracting edge, the dominant aperture field at P is chosen.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RoofCorn: identifies point P as (0:Receiver),(1:RoofEdge),(2:Corner)
        APTMP: sampled aperture fields (amp and phs) due to both roof edges
        RItmp: field at P due to both edges
        cnt: number of roof edges
        N1tmp: Region 1 samples for the two roof edge aperture fields
* OUTPUTS:
        APTMP: dominant aperture field, amp and phs, stored in index 0
        RIA: sum of fields at P, (0:real),(1:imag),(2:power sum, mean field)
        *N1roof: Region 1 samples of dominant aperture field
* CALLED BY: BldgDiff1, BldgDiff2
* FUNCTION CALLS: none
*************************************************************************/
void SumRoofDiff(int RoofCorn, DA2 APTMP,
        double RItmp[2][2], int cnt, int N1tmp[2], double RIA[3], int *N1roof)
{
        double rl,im,rln,imn;
        int n,k,nc;
        *N1roof = 0;
        for(k=0;k<=2;k++)
                RIA[k] = 0.0;

        if( cnt == 0 ) /* No Fields */
                return;

        if( cnt <= 1 )  { /* One Field */
            if( !RoofCorn ) {
                RIA[0] = RItmp[0][0];
                RIA[1] = RItmp[1][0];
                RIA[2] = RItmp[0][0]*RItmp[0][0] + RItmp[1][0]*RItmp[1][0];
                }
            else
                *N1roof = N1tmp[0];
            return;
            }
        /* Else cnt = 2 */

/* Sum fields */
        if( !RoofCorn ) {  /* Receiver point */
            for(n=0;n<=1;n++) {
                  RIA[0] = RIA[0] + RItmp[0][n];
                  RIA[1] = RIA[1] + RItmp[1][n];
                  RIA[2] = RIA[2] + RItmp[0][n]*RItmp[0][n] +
                                    RItmp[1][n]*RItmp[1][n];
                  } /* ampmn E squared */
            return;
            }

        /* Else Roof or Corner */
        if( APTMP[0][1][0] > APTMP[0][0][0] ) {
            for(n=0;n<=1;n++)
                for(k=0;k<=(SZ-1);k++)
                  APTMP[n][0][k] = APTMP[n][1][k];
            *N1roof = N1tmp[1];
            }
        else
            *N1roof = N1tmp[0];
        return;
```

```
}


/*************************************************************************
* FUNCTION: SumCornDiff
* DESCRIPTION: Combines the corner fields due to the (maximum) two sides.
        If P is the receiver, both the complex phasor sum and the power
        sum (mag squared of the mean field strength computed as the sum
        of the squares of the component field strengths) are computed
        and stored in RIA. If P is the first intermediate point on
        a diffracting edge, the dominant aperture field at P is chosen.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RoofCorn: identifies point P as (0:Receiver),(1:RoofEdge),(2:Corner)
        APTMP: sampled aperture fields (amp and phs) due to both sides
        RItmp: field at P due to both sides
        cnt: number of sides
        N1tmp: Region 1 samples for the two aperture fields
* OUTPUTS:
        APDAT: dominant aperture field, amp and phs
        RIA: sum of fields at P, (0:real),(1:imag),(2:power sum, mean field)
        *N1corn: Region 1 samples of dominant aperture field
* CALLED BY: CornDiff
* FUNCTION CALLS: none
*************************************************************************/
void SumCornDiff(int RoofCorn, DA2 APTMP,
        double RItmp[2][2], int cnt, int N1tmp[2],
        DA APDAT, double RIA[3], int *N1corn)
{
        double rl,im,rln,imn;
        int n,k,RC1,nc;
        for(k=0;k<=2;k++)
                RIA[k] = 0.0;

        if( cnt == 0 ) /* No Fields */
                return;

        if( cnt <= 1 )  { /* One Field */
            if( !RoofCorn ) {
                RIA[0] = RItmp[0][0];
                RIA[1] = RItmp[1][0];
                RIA[2] = RItmp[0][0]*RItmp[0][0] + RItmp[1][0]*RItmp[1][0];
                }
            else
                *N1corn = N1tmp[0];
            return;
            }

/* Sum fields */
        if( !RoofCorn ) {  /* Receiver point */
            for(n=0;n<=1;n++) {
                  RIA[0] = RIA[0] + RItmp[0][n];
                  RIA[1] = RIA[1] + RItmp[1][n];
                  RIA[2] = RIA[2] + RItmp[0][n]*RItmp[0][n] +
                                    RItmp[1][n]*RItmp[1][n];
                  } /* ampmn, E squared */
            return;
            }

        if( APTMP[0][1][0] > APTMP[0][0][0] ) {
            for(n=0;n<=1;n++)
                for(k=0;k<=(SZ-1);k++)
                  APDAT[n][k] = APTMP[n][1][k];
            *N1corn = N1tmp[1];
```

```
            }
        else {
            for(n=0;n<=1;n++)
                for(k=0;k<=(SZ-1);k++)
                  APDAT[n][k] = APTMP[n][0][k];
            *N1corn = N1tmp[0];
            }
        return;
}


/*************************************************************************
* FUNCTION: ChooseDiff
* DESCRIPTION: Combines the fields due to the roof and corner fields
        If P is the receiver, both the complex phasor sum and the power
        sum (mag squared of the mean field strength computed as the sum
        of the squares of the component field strengths) are computed
        and stored in RIA. If P is the first intermediate point on
        a diffracting edge, the dominant aperture field at P is chosen.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        rcnt: number of roof fields
        kcnt: number of corner fields
        RoofCorn: identifies point P as (0:Receiver),(1:RoofEdge),(2:Corner)
        APTMP: sampled aperture field, dominant roof edge (stored in index 0)
        APDAT: sampled aperture field, dominant corner field
        RIAroof: combined roof-edge fields
        RIAcorn: combined corner fields
        N1roof: Region 1 samples of dominant aperture fld due to a roof-edge
        N1corn: Region 1 samples of dominant aperture fld due to a corner
* OUTPUTS:
        APDAT: dominant aperture field, amp and phs
        RIA: sum of fields at P, (0:real),(1:imag),(2:power sum, mean field)
        *RCout: type of edge contributing dominant field (1:roof),(2:corner)
        *N1out: Region 1 samples of dominant aperture field
* CALLED BY: BldgDiff1, BldgDiff2
* FUNCTION CALLS: none
*************************************************************************/
void ChooseDiff(int rcnt, int kcnt, int RoofCorn, DA2 APTMP, DA APDAT,
        double RIAroof[3], double RIAcorn[3],
        int N1roof, int N1corn, double RIA[3], int *RCout, int *N1out)
{
        int kc,k,n;
        *N1out = 0;
        for(k=0;k<=2;k++)
                RIA[k] = 0.0;
        if( (!rcnt) && (!kcnt) ) {
            *RCout = 0;
            return;
            }
        if(!rcnt) {
            *RCout = 2;
            *N1out = N1corn;
            if( !RoofCorn ) {
                for(k=0;k<=2;k++)
                    RIA[k] = RIAcorn[k];
                }
            }
        else if(!kcnt) {
            *RCout = 1;
            *N1out = N1roof;
            if( !RoofCorn ) {
                for(k=0;k<=2;k++)
                    RIA[k] = RIAroof[k];
                }
```

```
            else {
                for(k=0;k<=1;k++)
                        for(n=0;n<=(SZ-1);n++)
                            APDAT[k][n] = APTMP[k][0][n];
                }
            }
        else if( kcnt && rcnt) {
            if( !RoofCorn ) {
                for(k=0;k<=2;k++)
                    RIA[k] = RIAroof[k] + RIAcorn[k];
                *RCout = 1;
                }
            else
                if( APTMP[0][0][0] > APDAT[0][0] ) {
                    *RCout = 1;
                    *N1out = N1roof;
                    for(k=0;k<=1;k++)
                            for(n=0;n<=(SZ-1);n++)
                                APDAT[k][n] = APTMP[k][0][n];
                    }
                else
                    *RCout = 2;
                    *N1out = N1corn;
            }
        return;
}


/*************************************************************************
* FUNCTION: computediff
* DESCRIPTION: Computes the diffraction field at P due to a diffracting
        edge at an earlier point Q. If P is the receiver, the field is computed
        at a single point, otherwise, the field in the aperture at P is
        sampled. If there is a source of diffraction prior to Q, the
        field at P is found by multiple diffraction, with the dimension
        of integration in the aperture at Q determined by the type of edge
        producing the diffraction field at Q. Otherwise, single diffraction
        by Q is computed at P.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RoofCorn: identifies point P as rcvr(0), roof edge(1) or corner(2)
        RC1: identifies source of diffraction field at Q (if any),
                                0:none, 1:roof edge, 2:corner
        k: identifies type of diffracting edge at Q, 0:roof edge, 1:corner
        yp: height of point P
        rso: distance from S to origin of aperture plane
        rpo: distance from P to origin of aperture plane
        y1: vertical distance from origin to bottom of aperture
        V1: horizontal distance from origin to left edge of aperture
        V2: horizontal distance from origin to right edge of aperture
    The aperture at Q is defined by (V1,V2) and (y1,y2), where y2 is infinity
        uq: u coord of Q
        lam: wavelength
        N1: number of Region 1 samples of aperture field at Q
        D: sampling intervals for aperture fields
        up: u coord of P
        ybldg1: height of building
        yt: height of transmitter
        APDAT: aperture field samples (amp and phs) at Q
        *cnt: counter of field components
* OUTPUTS:
        APTMP: aperture field samples (amp and phs) at P
        RItmp: field at P
        *cnt: incremented counter of fld components
        N1out: number of Region 1 samples at P
```

```
        *BLOCKEDout: diffraction field blocked (too weak)
* CALLED BY: BldgDiff1, BldgDiff2, CornDiff1
* FUNCTION CALLS:
        EvalField: evaluates multiple diffraction
        fieldcomp: evaluates single diffraction
        computeYfield: samples vertical aperture field at P
        computeXfield: samples horizontal aperture field at P
        SmoothPhs2: smooths, or un-wraps, the phase
*************************************************************************/

void computediff(int RoofCorn, int RC1, int k,
        double yp, double rso, double rpo, double y1, double V1, double V2,
        double uq, /* These are the coord's of yorigin, NOT the field point */
        double lam, int N1, double D[2], double up, double ybldg1, double yt,
        DA APDAT, DA2 APTMP, double RItmp[2][2],
        int *cnt, int N1out[2], int *BLOCKEDout)
{
/* k implies computing field past a corner, NOT at a corner */
        double real,imag,amp,phs; /* field descriptions */
        double v1;              /* distance to corner edge */
        int kcnt;               /* identifies storage location of outputs */
        int N1new;              /* Region 1 samples at P */
        int BLOCKED;            /* field blockage */
        int NTot;               /* total samples of aperture field at P */

        kcnt = *cnt;
        BLOCKED = 0;

        if( !RoofCorn ) /* Last source; Find field at rcvr */ {
            if( RC1 ) {  /* There is a source of diffraction prior to Q */
                if(!k)

                    if( RC1==1 )  /* roof to roof to P */
                        EvalField(rso,rpo,y1,V1,V2,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);

                    else  /* corner to roof to P,
                             flip y and v axes in input list */
                        EvalField(rso,rpo,0.0,y1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);

                else
                    if( RC1==1 )  /* roof to corner to P */
                        EvalField(rso,rpo,0.0,V1,V2,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);

                    else  {   /* corner to corner to P, flip axes,
                                 treat as a rotated roof-roof-P diff situation */
                        v1 = V1;  /* use v1 in place of y1 in input list */
                        if(v1 < -500)  /* If diffraction is around left side,
                                          reverse signs, upper limit will
                                          be taken as positive infinity */
                            v1 = -V2;
                        EvalField(rso,rpo,v1,y1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
                        }
                }

            else  /* single diffraction at P by edge at Q */
                fieldcomp(rso,rpo,y1,V1,V2,lam,&real,&imag,&amp,&phs);
            RItmp[0][kcnt] = real;
            RItmp[1][kcnt] = imag;
            }

        else {  /* P is an intermediate source;
```

```
                Sample field in an aperture at P */
            if(!k) /* Q is a roof edge */
                computeYfield(RC1,kcnt,
                        yp,V1,V2,
                        uq,lam,N1,D,up,ybldg1,yt,APDAT,
                        /*OUTPUTS*/ APTMP,&N1new,&BLOCKED);
            else  /* Q is a corner */
                computeXfield(RC1,kcnt,
                        rso,rpo,y1,V1,V2,
                        uq,lam,N1,D,up,APDAT,
                        /*OUTPUTS*/ APTMP,&N1new,&BLOCKED);
            NTot = N1new + 2;
            SmoothPhs2(APTMP,kcnt,NTot,yp);
            }

        if(BLOCKED) {
          *BLOCKEDout = 1;
          return;
          }

        N1out[kcnt] = N1new;
        kcnt++;
        *cnt = kcnt;
        *BLOCKEDout = 0;
        return;
}


/*************************************************************************
* FUNCTION: computeYfield
* DESCRIPTION: Computes the diffraction field in an aperture at P,
        where the samples are taken over the vertical dimension
        because the diffracting edge at Q is horizontal.
        P is itself an intermediate point at a diffracting edge,
        requiring the characterization of the aperture field.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RC1: identifies source of diffraction field at Q (if any),
                                0:none, 1:roof edge, 2:corner
        kcnt: number of field component
        yp: height of point P
        V1: v component of the left edge of building face
        V2: v component of the right edge of bldg face
        uq: u coord of Q
        lam: wavelength
        N1: number of Region 1 samples of aperture field at Q
        D: sampling intervals for aperture fields
        up: u coord of P
        ybldg1: height of building
        yt: height of transmitter
        APDAT: aperture field samples (amp and phs) at Q
* OUTPUTS:
        APTMP: aperture field samples (amp and phs) at P
        *N1out: number of Region 1 samples at P
        *BLOCKEDout: diffraction field blocked (too weak)
* CALLED BY: computediff
* FUNCTION CALLS:
        FldPar2: aperture field parameters for high transmitter
        FldPar1: aperture field parameters for all other cases
        EvalField: evaluates multiple diffraction
        fieldcomp: evaluates single diffraction
        FindnrEnd: finds next clearance value that gives a phase slope minimum
* NOTES:
        Algorithm for ending integration Region 1: Must sample at
```

```
        least nmin field points. If there is less than 0.55 Fresnel zone
        clearance at the nmin point, but greater than zero clearance,
        then end Region 1 there.  If there is greater than 0.55 zone clearance
        at nmin, find the next zone clearance value that provides a
        phase slope minimum and continue sampling to that point. If there
        is no clearance at nmin, then end Region 1 at the first point to
        have any clearance.
*************************************************************************/
void computeYfield(int RC1, int kcnt,
        double yp, double V1, double V2,
        double uq,  /* These are the coord's of yorigin,
                               NOT the field point */
        double lam, int N1, double D[2], double up, double ybldg1, double yt,
        DA APDAT, DA2 APTMP, int *N1out, int *BLOCKEDout)
{
        double yp1,y;           /* heights of aperture points */
        double y1;              /* height of edge relative to origin */
        double yorigin;         /* height of origin of aperture plane */
        double rso,rpo;         /* distances to source and observation points */
        double real,imag,amp,phs;  /* field descriptions */
        double rtot0,rso0;      /* distances for first point in aperture */
        double Delta1,Delta2;   /* sample intervals */
        double nrStart,nrEnd,nr;/* Fresnel zone clearance values */
        int n,n2;               /* counters */
        int N1P;                /* number of samples in Region 1 */
        int ContUntilMin;       /* continue Region 1 until next phs slope min*/
        int Continue;           /* keep sampling Region 1*/
        int nmin;               /* min number of samples required, Region 1 */

        Delta1 = D[0];
        Delta2 = D[1];
        nmin = 25;

        if( yt > yp ) {  /* need some more parameters */
                rtot0 = sqrt( (yt-yp)*(yt-yp) + up*up );
                rso0 = uq*(rtot0/up);
                }

        n = -1;
        ContUntilMin = 0;
        Continue = 0;
        do {
                n++;
                y = yp + n*Delta1; /* Where y is now hgt of fld point */

                if( yt > yp ) /* Special case: aperture plane must tilt
                                 and remain perpendicular to SP, otherwise
                                 the phase curvature comes out negative */
                    FldPar2(rtot0,rso0,uq,yp,yt,y,&rso,&rpo,&yorigin);
                else
                    FldPar1(uq,up,yt,y,&rso,&rpo,&yorigin);

                y1 = ybldg1 - yorigin;

                if( RC1 ) /* multiple diffraction */
                    if( RC1==1 ) /* roof to roof to P */
                        EvalField(rso,rpo,y1,V1,V2,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
                    else  /* corner to roof to P, flip axes in input list */
                        EvalField(rso,rpo,0.0,y1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);

                else /* single diffraction */
                    fieldcomp(rso,rpo,y1,V1,V2,lam,&real,&imag,&amp,&phs);
```

```
            APTMP[0][kcnt][n] = amp;
            APTMP[1][kcnt][n] = phs;

            if( (n==nmin) && (y1 < 0.0)
                && (y1*y1 > 0.3*lam*rso*rpo/(rso+rpo)) ) {
                    ContUntilMin = 1;
                    nrStart = y1*y1*(rso+rpo)/(lam*rso*rpo);
                    nrEnd = FindnrEnd(nrStart);
                    printf("nrStart: %.3lf; nrEnd: %.3lf\n",nrStart,nrEnd);
                    }

            if( ContUntilMin ) {
                    nr = y1*y1*(rso+rpo)/(lam*rso*rpo);
                    if( nr > (nrEnd-0.1) )
                            Continue = 0;
                    else
                            Continue = 1;
                    }

            } while( (n<nmin) ||
                    ( ((Continue)||(y1>0.0)) &&
                            (n<(SZ-3)) ) );
            /*End First loop (Region 1) through vertical region */

        N1P = n;
        printf("N1 = %d;  y= %.3lf  y1= %.3lf\n",N1P,y,y1);
        if( y1 > 0.0 ) {
            printf("POSSIBLE ERROR: y1 > 0\n");
            *BLOCKEDout = 1;
            return;
            }

        yp1 = yp + N1P*Delta1;
        for(n=1;n<=2;n++) {  /* sample Region 2 */
                n2 = n + N1P;
                y = yp1 + n*Delta2;
                if( yt > yp )
                        FldPar2(rtot0,rso0,uq,yp,yt,y,&rso,&rpo,&yorigin);
                else
                        FldPar1(uq,up,yt,y,&rso,&rpo,&yorigin);
                        /* New yorigin on intermediate source;
                        - yorigin is in absolute coord's
                        - y1 is in relative coord's     */
                y1 = ybldg1 - yorigin;
                if( RC1 )
                        if( RC1==1 )
                            EvalField(rso,rpo,y1,V1,V2,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
                        else
                            EvalField(rso,rpo,0.0,y1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
                else
                       fieldcomp(rso,rpo,y1,V1,V2,lam,&real,&imag,&amp,&phs);

                APTMP[0][kcnt][n2] = amp;
                APTMP[1][kcnt][n2] = phs;
                } /* end for */

        *N1out = N1P;
        return;
}
```

```
/*************************************************************************
* FUNCTION: computeXfield
* DESCRIPTION: Computes the diffraction field in an aperture at P,
        where the samples are taken over the horizontal dimension
        because the diffracting edge at Q is vertical.
        P is itself an intermediate point on a diffracting edge,
        requiring the characterization of the aperture field.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        RC1: identifies source of diffraction field at Q (if any),
                                0:none, 1:roof edge, 2:corner
        kcnt: number of field component
        rso1: origin to source distance for first point P in aperture
        rpo1: origin to P distance for first point P in aperture
        y1: vertical distance from origin to bottom of aperture
        V1: horizontal distance from origin to left edge of aperture
        V2: horizontal distance from origin to right edge of aperture
        uq: u coord of Q
        lam: wavelength
        N1: number of Region 1 samples of aperture field at Q
        D: sampling intervals for aperture fields
        up: u coord of P
        APDAT: aperture field samples (amp and phs) at Q
* OUTPUTS:
        APTMP: aperture field samples (amp and phs) at P
        *N1out: number of Region 1 samples at P
        *BLOCKEDout: diffraction field blocked (too weak)
* CALLED BY: computediff
* FUNCTION CALLS:
        EvalField: evaluates multiple diffraction
        fieldcomp: evaluates single diffraction
        FindnrEnd: finds next clearance value that gives a phase slope minimum
* NOTES:
        Algorithm for ending integration Region 1: Must sample at
        least nmin field points. If there is less than 0.55 Fresnel zone
        clearance at the nmin point, but greater than zero clearance,
        then end Region 1 there.  If there is greater than 0.55 zone clearance
        at nmin, find the next zone clearance value that provides a
        phase slope minimum and continue sampling to that point. If there
        is no clearance at nmin, then end Region 1 at the first point to
        have any clearance.
*************************************************************************/
void computeXfield(int RC1, int kcnt,
        double rso1, double rpo1, double y1, double V1, double V2,
        double uq, /* These are the coord's of yorigin,
                                NOT the field point */
        double lam, int N1, double D[2], double up,
        DA APDAT, DA2 APTMP, int *N1out, int *BLOCKEDout)
{
        double vp,vp1;          /* v coord's of points P in aperture */
        double v1a,v1;          /* distances from origin to edge at Q */
        double rso,rpo;         /* distances to source and observation points */
        double real,imag,amp,phs;  /* field descriptions */
        double rp;              /* total distance from source to P */
        double Delta1,Delta2;   /* sample intervals */
        double nrStart,nrEnd,nr;/* Fresnel zone clearance values */
        int n,n2;               /* counters */
        int N1P;                /* number of samples in Region 1 */
        int ContUntilMin;       /* continue Region 1 until next phs slope min*/
        int Continue;           /* keep sampling Region 1*/
        int nmin;               /* min number of samples required, Region 1 */

        Delta1 = D[0];
        Delta2 = D[1];
        nmin = 25;
```

```
if(V1 < -500.0)
        v1a = -V2;
else
        v1a = V1;
n= -1;
ContUntilMin = 0;
Continue = 0;

do {
        n++;
        vp = n*Delta1;    /* vp always is zero at first P
                             in aperture */
        v1 = v1a - vp*(uq/up);
        rp = sqrt(up*up + vp*vp);
        rso = rso1 * (rp/up);
        rpo = rpo1 * (rp/up);

        if( RC1 )  /* multiple diffraction */
            if( RC1==1 ) /* roof to corner to P */
                EvalField(rso,rpo,0.0,v1,1000.0,lam,N1,D,APDAT,
                    &real,&imag,&amp,&phs);
            else   /* corner to corner to P */
                EvalField(rso,rpo,v1,y1,1000.0,lam,N1,D,APDAT,
                    &real,&imag,&amp,&phs);

        else   /* single diffraction */
            fieldcomp(rso,rpo,y1,v1,1000.0,lam,&real,&imag,&amp,&phs);

        APTMP[0][kcnt][n] = amp;
        APTMP[1][kcnt][n] = phs;

        if( (n==nmin) && (v1 < 0.0)
            && (v1*v1 > 0.3*lam*rso*rpo/(rso+rpo)) ) {
                ContUntilMin = 1;
                nrStart = v1*v1*(rso+rpo)/(lam*rso*rpo);
                nrEnd = FindnrEnd(nrStart);
                printf("nrStart: %.3lf; nrEnd: %.3lf\n",nrStart,nrEnd);
                }

        if( ContUntilMin ) {
                nr = v1*v1*(rso+rpo)/(lam*rso*rpo);
                if( nr > (nrEnd-0.1) )
                        Continue = 0;
                else
                        Continue = 1;
                }

        }  while( (n<nmin) ||
                ( ((Continue)||(v1>0.0)) &&
                        (n<(SZ-3)) ) );
/*End First FOR loop through vertical region */

N1P = n;
printf("N1 = %d;  vp= %.3lf  v1= %.3lf\n",N1P,vp,v1);
if( v1 > 0.0 ) {
    printf("POSSIBLE ERROR: v1 > 0\n");
    *BLOCKEDout = 1;
    return;
    }

vp1 = N1P*Delta1;
for(n=1;n<=2;n++) {
    n2 = n + N1P;
    vp = vp1 + n*Delta2;
```

```
            v1 = v1a - vp*(uq/up);
            rp = sqrt(up*up + vp*vp);
            rso = rso1 * (rp/up);
            rpo = rpo1 * (rp/up);

            if( RC1 )
                if( RC1==1 )
                    EvalField(rso,rpo,0.0,v1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
                else
                    EvalField(rso,rpo,v1,y1,1000.0,lam,N1,D,APDAT,
                                &real,&imag,&amp,&phs);
            else
                fieldcomp(rso,rpo,y1,v1,1000.0,lam,&real,&imag,&amp,&phs);

            APTMP[0][kcnt][n2] = amp;
            APTMP[1][kcnt][n2] = phs;
            } /* end for */

        *N1out = N1P;
        return;
}


/*************************************************************************
* FUNCTION: double FindnrEnd
* DESCRIPTION: Returns the next Fresnel zone clearance for which the phase
        slope will be at a minimum, occurs at 2k+1.7; k=0,1,2...
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: nrStart: Fresnel zone clearance at the min number of samples
* OUTPUTS: none other than return value
* CALLED BY: computeYfield, computeXfield
* FUNCTION CALLS: none
*************************************************************************/
double FindnrEnd(double nrStart)
{
        int itemp;
        itemp = (int)(nrStart - 0.7) - 1;
        if(itemp < 0) itemp = itemp-1;
        itemp = itemp/2;
        itemp = itemp + 1;
        itemp = itemp*2;
        return(itemp+1.7);
}

/*************************************************************************
* FUNCTION: FillAmptmp
* DESCRIPTION: Fills APTMP at the right component index with the
        contents of APDAT
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:  APDAT: amp,phs samples of aperture field
           klr: component index
* OUTPUTS: APTMP: amp,phs samples of aperture field, indexed by component
* CALLED BY: BldgDiff1, BldgDiff2, CornDiff1
* FUNCTION CALLS: none
*************************************************************************/
void FillAmptmp(DA APDAT, DA2 APTMP, int klr)
{
        int j,k;        /* loop counters */
        for(k=0;k<=1;k++)
                for(j=0;j<=(SZ-1);j++)
                    APTMP[k][klr][j] = APDAT[k][j];
        return;
}
```

```
/*************************************************************************
* FUNCTION: FillAmpdat
* DESCRIPTION: Fills APDAT with the contents of a
        single field component in APTMP
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:  APTMP: amp,phs samples of aperture field, two field components
          klr: component index
* OUTPUTS: APDAT: amp,phs samples of a single component of the aperture field
* CALLED BY: CornDiff1
* FUNCTION CALLS: none
*************************************************************************/
void FillAmpdat(DA2 APTMP, DA APDAT, int klr)
{
        int j,k;        /* loop counters */
        for(k=0;k<=1;k++)
                for(j=0;j<=(SZ-1);j++)
                    APDAT[k][j] = APTMP[k][klr][j];
        return;
}


/*************************************************************************
* FUNCTION: SmoothPhs2
* DESCRIPTION: Removes 2Pi discontinuities
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        APTMP: amp and phs of the sample aperture field
        kc: component index
        NTot: total number of samples
        y0: height of first sample point
* OUTPUTS: APTMP
* CALLED BY: computediff
* FUNCTION CALLS: none
*************************************************************************/
void SmoothPhs2(DA2 APTMP, int kc, int Ntot, double y0)
{
        double temp;            /* holds last sample value */
        double offset;          /* cumulative factor to offset
                                   2Pi discontinuities */
        int k;                  /* loop counter */
        int N1;                 /* number of Region 1 samples */

/* Phase may decrease or increase; It should change by large
  amounts only in the outer region, where the change should be positive.
  Therefore: if the change is very near 2pi, assume it was a positive
  wrap-around, and subtract 2pi.   */
        N1 = Ntot - 2;
        offset = 0.0;
        temp =APTMP[1][kc][0];
        if(Printphs) {
          fprintf(filephs," New, y0 = %.3lf\n",y0);
          fprintf(filephs," %d  %.3lf  %.3lf\n",0,temp,APTMP[1][kc][0]);
          }

/* Region 1: characterized by large negative phase slopes and small
        positive phase slopes */

        for(k=1;k<=N1;k++) {
            if( APTMP[1][kc][k]-temp >  2.5 )
                    offset = offset - 2.0*pi;
            else  if( temp-APTMP[1][kc][k] >  (2.0*pi - 2.5) )
                        offset = offset + 2.0*pi;
```

```
            temp = APTMP[1][kc][k];
            APTMP[1][kc][k] = APTMP[1][kc][k] + offset;
            if(Printphs)
              fprintf(filephs," %d  %.3lf  %.3lf\n",k,temp,APTMP[1][kc][k]);
            }

/* Region 2: characterized by only positive phase slopes, may be very large,
        very rare to have an additional discontinuity in Region 2 */

        for(k=(N1+1);k<=Ntot;k++) {
            if( temp-APTMP[1][kc][k] >  1.0 )
                    offset = offset + 2.0*pi;
/*          else  if( APTMP[1][kc][k]-temp >  (2.0*pi - 0.4) )
                        offset = offset - 2.0*pi;  */
/* the check for positive discontinuities is removed to allow for
   natural, large positive phase differences in Region 2 */
            temp = APTMP[1][kc][k];
            APTMP[1][kc][k] = APTMP[1][kc][k] + offset;
            if(Printphs)
              fprintf(filephs," %d  %.3lf  %.3lf\n",k,temp,APTMP[1][kc][k]);
            }
        return;
}
```

```
/*************************************************************************
* FILE NAME: MDB2.C
* CONTENTS:
        uintercept
        PntCoordzx1
        PntCoordzxQ1
        BldgCoorduv1
        BldgCoorduvQ1
        BldgCoorduv1a
        FldPar1
        FldPar2
        BoundBldg1
        SortBldg1
        SortCorn1
        SortEdges
        SortCorn
        checkintermRoof
        GetPntR2
        checkFZ1
        checkintermCorn
        GetPntR
        MoveTrans
        checkFZcorn
        checkNotBlock
        checkFZ2
        checkFZ4
        qcoords
        rEdgeCoords
        qEdgeCoords
        NearestDiffBldg
        NearestBldg
        Sift
        cyc4neg
        cyc4pos
* DESCRIPTION:
        This file contains all of the geometrical manipulation routines
*************************************************************************/
#include <mdsrch.h>
#include <mdtypes.h>
void uintercept(BAC1 uvcorn, int i, double *u0);
void PntCoordzx1(ZXUVY P, ZXUVY Q);
void PntCoordzxQ1(ZXUVY P, ZXUVY Q);
void BldgCoorduv1(ZXUVY P, int j, BAC zxcorn, BAC1 uvcorn);
void BldgCoorduvQ1(ZXUVY Q, int j, BAC zxcorn, BAC1 uvcorn);
void BldgCoorduv1a(ZXUVY P, BAC1 zxcorn1, BAC1 uvcorn);
void FldPar1(double u, double up, double yt, double yp,
        double *rso, double *rpo, double *yorigin);
void FldPar2(double rtot0, double rso0, double u, double yp,
        double yt, double y, double *rso, double *rpo, double *yorigin);
void BoundBldg1(BAC1 uvcorn,
        double *umax, double *umin, double *vmax, double *vmin,
        int *iumax, int *iumin, int *ivmax, int *ivmin);
void SortBldg1(BAC1 uvcorn,
        int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2]);
void SortCorn1(BAC1 uvcorn, int CORN[2][4], int ncorn[2]);
void SortEdges(BAC1 uvcorn, int EDGE[3][2],
        int nedge[3], double *vmax1, double *vmin1,
        int *ivmax1, int *ivmin1);
void SortCorn(int jtot, int kc, double u[4], int iu[4], int CORN[2][4]);
int checkintermRoof(ZXUVY P, ZXUVY Q, BAH ybldg,
        double lam, BAC zxcorn,
        int iConn, int jConn, int iQ, int jQ);
int GetPntR2(ZXUVY Q, int iConn, int jConn,
                double ybldgConn, BAC zxcorn, ZXUVY R);
int checkFZ1(ZXUVY P, double yt, BAC1 uvcorn, int i,
```

```
        double ybldg1, double lam,
        /* OUTPUTS */ ZXUVY Q );
int checkintermCorn(ZXUVY P, ZXUVY Q, double yt,
        double lam, LCL LOWCORNLIM, BAC zxcorn,
        int CORN[2][4], int klr, int kc, int j);
int GetPntR(ZXUVY Q, int CORN[2][4], int kc, int klr, BAC zxcorn,
        LCL LOWCORNLIM, int j, double yt, double lam, ZXUVY R);
void MoveTrans(ZXUVY P, ZXUVY R, int j, BAC zxcorn, ZXUVY Pr, BAC1 uvcorn);
int checkFZcorn(ZXUVY P, ZXUVY Q, double yt, double lam,
        double LOWCORNLIM1, int klr,
        double *rso, double *rpo, double *y1, double *v1, double *v2);
int checkNotBlock(double rso, double rpo, double y1,
        double v1, double v2, double lam);
int checkFZ2(int Blev, ZXUVY P, double yt, int EDGE[3][2], int nedge[3],
        BAC1 uvcorn, double ybldg1, double lam, ZXUVY Q1, ZXUVY Q2,
        double rso[2], double rpo[2], double y1[2],
        double v1[2], double v2[2]);
int checkFZ4(double up, double yp,
                double yt, BAC1 uvcorn, double ybldg1, double lam);
void qcoords(BAC1 uvcorn, double ybldg1,
        int i, double *uq, double *vq, double *yq);
void rEdgeCoords(ZXUVY Q, BAC1 uvcorn, double ybldg1, int i, ZXUVY R);
void qEdgeCoords(ZXUVY P, BAC1 uvcorn, double ybldg1, int i, ZXUVY Q);
void NearestDiffBldg(ZXUVY P, double yt, BAH ybldg, double lam,
        BAC zxcorn, double umax[Nb], BAF OBSTRUCT, BAF TOOSHORT,
        BAC1 uvcorn, int *NOBLDGS, int *j);
void NearestBldg(BAF OBSTRUCT, BAF TOOSHORT, double umax[Nb], int *jnear);
void Sift(ZXUVY P, BAC zxcorn, BAF MASK,
        BAF OBSTRUCT, double umax[Nb], int *NOBLDGS);
int cyc4neg(int n);
int cyc4pos(int n);
/*  EVALFLD.C  */
extern int sign(double x);
/*             */


/*****************************************************************************
* FUNCTION: uintercept
* DESCRIPTION: finds u intercept of line connecting
        a single set of building corners, that is, the intersection of
        the u axis with a building face. This routine should only be
        called after it is determined that the u axis does indeed
        intersect the face.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: uvcorn: corner coordinates
          i: index of building edge of interest, and of one of the corners
* OUTPUTS: *u0: u intercept
* CALLED BY: qcoords, rEdgeCoords, qEdgeCoords
* FUNCTION CALLS: cyc4neg: finds i-1 on a cyclic domain of 0,3
*****************************************************************************/
void uintercept(BAC1 uvcorn, int i, double *u0)
{
        int im1;        /* index of the second corner bounding the face */
        double u1,u2,v1,v2; /* corner coordinates */

        im1 = cyc4neg(i);
        u1 = uvcorn[0][im1];
        u2 = uvcorn[0][i];
        v1 = uvcorn[1][im1];
        v2 = uvcorn[1][i];
        *u0 = u1-( (u2-u1)/(v2-v1) )*v1;
        return;
}

/*****************************************************************************
```

```
* FUNCTION: PntCoordzx1
* DESCRIPTION: Finds (z,x) coord's of a point Q
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: z,x,and u coord's of P (v coord of P is zero)
          u,v coord's of Q
* OUTPUTS: z,x coord's of Q
* CALLED BY: qEdgeCoords, CornDiff1
* FUNCTION CALLS: none
*****************************************************************************/
void PntCoordzx1(ZXUVY P, ZXUVY Q)
{
        Q[0] = Q[2] * (P[0]/P[2])  -  Q[3] * (P[1]/P[2]);
        Q[1] = Q[2] * (P[1]/P[2])  +  Q[3] * (P[0]/P[2]);
        return;
}


/*****************************************************************************
* FUNCTION: PntCoordzxQ1
* DESCRIPTION: Finds (z,x) coord's of a point R, based on Q
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: z,x, and u coord's of Q (where v coord of Q may be non-zero)
          u,v coord's of R
* OUTPUTS: z,x coord's of R
* CALLED BY: GetPntR2, GetPntR, rEdgeCoords
* FUNCTION CALLS: none
*****************************************************************************/
void PntCoordzxQ1(ZXUVY Q, ZXUVY R)
{
        double rq;  /* distance from source to Q */
        rq = sqrt( Q[0]*Q[0] + Q[1]*Q[1] );
        R[0] = R[2] * (Q[0]/rq)  -  R[3] * (Q[1]/rq);
        R[1] = R[2] * (Q[1]/rq)  +  R[3] * (Q[0]/rq);
        return;
}


/*****************************************************************************
* FUNCTION: BldgCoorduv1
* DESCRIPTION: Rotates (z,x) coordinates of a single building
        into a new (u,v) system defined such that the v coord of
        P is zero, and (u,v) of the source is (0,0).
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: P: coordinates of P
          j: index of building of interest
          zxcorn: (z,x) cord's of all buildings
* OUTPUTS: uvcorn: (u,v) coords of a single building
* CALLED BY: NearestDiffBldg, Sift, BldgDiff1, BldgDiff2, CornDiff1
* FUNCTION CALLS: none
*****************************************************************************/
void BldgCoorduv1(ZXUVY P, int j, BAC zxcorn, BAC1 uvcorn)
{
        int i;  /* loop counter */
        for(i=0;i<=3;i++) {
          uvcorn[0][i] = zxcorn[0][i][j] * (P[0]/P[2])
                + zxcorn[1][i][j] * (P[1]/P[2]);
          uvcorn[1][i] = -zxcorn[0][i][j] * (P[1]/P[2])
                + zxcorn[1][i][j] * (P[0]/P[2]);
        }
        return;
}


/*****************************************************************************
```

```
* FUNCTION: BldgCoorduvQ1
* DESCRIPTION: Rotates (z,x) coordinates of a single building
        into a new (u,v) system defined such that the v coord of
        P is zero, and (u,v) of the source is (0,0); however,
        performs rotation based on Q coord's.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: Q: coordinates of Q
          j: index of building of interest
          zxcorn: (z,x) cord's of all buildings
* OUTPUTS: uvcorn: (u,v) coords of a single building
* CALLED BY: NearestDiffBldg, Sift, BldgDiff1, BldgDiff2, CornDiff1
* FUNCTION CALLS: none
*****************************************************************************/
void BldgCoorduvQ1(ZXUVY Q, int j, BAC zxcorn, BAC1 uvcorn)
{
        double rq;      /* distance from source to Q */
        int i;          /* loop counter */
        rq = sqrt( Q[0]*Q[0] + Q[1]*Q[1] );
        for(i=0;i<=3;i++) {
          uvcorn[0][i] = zxcorn[0][i][j] * (Q[0]/rq)
                + zxcorn[1][i][j] * (Q[1]/rq);
          uvcorn[1][i] = -zxcorn[0][i][j] * (Q[1]/rq)
                + zxcorn[1][i][j] * (Q[0]/rq);
        }
        return;
}


/*****************************************************************************
* FUNCTION: BldgCoorduv1a
* DESCRIPTION: Rotates (z,x) coordinates of a single building
        into a new (u,v) system defined such that the v coord of
        P is zero, and (u,v) of the source is (0,0).
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: P: coordinates of P
          j: index of building of interest
          zxcorn: (z,x) cord's of a single building
* OUTPUTS: uvcorn: (u,v) coords of a single building
* CALLED BY: MoveTrans
* FUNCTION CALLS: none
*****************************************************************************/
void BldgCoorduv1a(ZXUVY P, BAC1 zxcorn1, BAC1 uvcorn)
{
        int i;          /* loop counter */
        for(i=0;i<=3;i++) {
          uvcorn[0][i] = zxcorn1[0][i] * (P[0]/P[2])
                + zxcorn1[1][i] * (P[1]/P[2]);
          uvcorn[1][i] = -zxcorn1[0][i] * (P[1]/P[2])
                + zxcorn1[1][i] * (P[0]/P[2]);
        }
        return;
}


/*****************************************************************************
* FUNCTION: void FldPar1
* DESCRIPTION: Includes the vertical dimension, using similar triangles,
        to find some geometrical parameters necessary in computing diff field
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u: distance, in horizontal plane, from source to edge
        up: distance (horiz) from source to P
        yt: transmitter height
        yp: height of P
```

```
* OUTPUTS:
        *rso: distance (3-D) from source to origin of aperture plane
        *rpo: distance (3-D) from P to origin of aperture plane
        *yorigin: height of point on aperture plane that intersects SP,
                defined as the origin height
* CALLED BY: computeYfield, checkFZ1, checkFZCorn, checkFZ2, checkFZ4
* FUNCTION CALLS: none
******************************************************************************/
void FldPar1(double u, double up, double yt, double yp,
        double *rso, double *rpo, double *yorigin)
{
        double rtot;
        rtot = sqrt( (yt-yp)*(yt-yp) + up*up );
        *rso = u * (rtot/up); /* include y-dimension */
        *rpo = rtot - *rso;
        *yorigin = ( (up-u) * (yt-yp) / up) + yp; /* by similar triangles */
        return;
}


/******************************************************************************
* FUNCTION: void FldPar2
* DESCRIPTION: Computes some necessary diffraction parameters based on
        an aperture that tilts away from the vertical plane, remaining
        perpendicular to the SP line, in order to avoid negative phase
        curvature in the case of the transmitter higher than the receiver.
        This routine should only be called instead of FldPar1 in that
        special case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        rtot0: distance from S to P for first point on aperture plane
        rso0: distance from S to origin at Q, based on first pnt P
        rpo0: distance from origin at Q to first point P
        u: distance, in horizontal plane, from source to edge
        yp: height of P
        yt: transmitter height
        y: height of origin of plane at Q, based on first pnt P
* OUTPUTS:
        *rso: distance (3-D) from source to origin of aperture plane
        *rpo: distance (3-D) from P to origin of aperture plane
        *yorigin: height of point on aperture plane that intersects SP,
                defined as the origin height
* CALLED BY: computeYfield
* FUNCTION CALLS: none
******************************************************************************/
void FldPar2(double rtot0, double rso0, double u, double yp,
        double yt, double y, double *rso, double *rpo, double *yorigin)
{
        double rtot;            /* total distance to new point P */
        double uy;              /* u coord of new P */
        rtot = sqrt( rtot0*rtot0 + (y-yp)*(y-yp) );
        *rso = rtot*(rso0/rtot0);
        *rpo = rtot - *rso;
        uy = sqrt( rtot*rtot + (y-yp)*(y-yp) );
        *yorigin = ( (uy-u) * (yt-y) / uy) + y;
        return;
}


/******************************************************************************
* FUNCTION: BoundBldg1
* DESCRIPTION: Determines the max and min values of u and v for
        a single building
```

```
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: uvcorn: corner coord's of a building
* OUTPUTS:
        *umax, *umin: max and min values of u
        *vmax, *vmin: max and min values of v
        *iumax, *iumin: indices of corners with max, min u values
        *ivmax, *ivmin: indices of corners with max, min v values
* CALLED BY: SortEdges, Sift
* FUNCTION CALLS: none
*****************************************************************************/
void BoundBldg1(BAC1 uvcorn,
        double *umax, double *umin, double *vmax, double *vmin,
        int *iumax, int *iumin, int *ivmax, int *ivmin)
{
        int i;          /* loop counter */
        *umax = -10000.0;
        *umin = 10000.0;
        *vmax = -10000.0;
        *vmin = 10000.0;
        for(i=0;i<=3;i++) {
           if( uvcorn[1][i] < *vmin ) {
                *vmin = uvcorn[1][i];
                *ivmin = i;
                }
           if( uvcorn[1][i] > *vmax ) {
                *vmax = uvcorn[1][i];
                *ivmax = i;
                }
           if( uvcorn[0][i] < *umin ) {
                *umin = uvcorn[0][i];
                *iumin = i;
                }
           if( uvcorn[0][i] > *umax ) {
                *umax = uvcorn[0][i];
                *iumax = i;
                }
           }
        return;
}


/*****************************************************************************
* FUNCTION: SortBldg1
* DESCRIPTION: Groups and sorts edges and corners of a single building
        according to their order in diffracting field
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: uvcorn: corner coord's of a building
* OUTPUTS:
        EDGE: ordered indices of roof edges of each level
        CORN: ordered indices of corner edges
        nedge: number of roof edges of each level
        ncorn: number of corners each side of SP
* CALLED BY: BldgDiff1, CornDiff1
* FUNCTION CALLS: SortEdges: sort roof edges
                  SortCorn1: sort corners
*****************************************************************************/
void SortBldg1(BAC1 uvcorn,
        int EDGE[3][2], int CORN[2][4], int nedge[3], int ncorn[2])
{
        double vmax,vmin;
        int ivmax,ivmin;
        SortEdges(uvcorn,EDGE,nedge,&vmax,&vmin,&ivmax,&ivmin);
        if( sign(vmax) == -sign(vmin) )
            SortCorn1(uvcorn,CORN,ncorn);
```

```
        else
        /* Special Case: Building is next to line of sight ray */
            if(vmax <= 0) {
                ncorn[0] = -1;
                ncorn[1] = 0;
                CORN[1][0] = ivmax;
                }
            else  {
                ncorn[0] = 0;
                ncorn[1] = -1;
                CORN[0][0] = ivmin;
                }
        return;
}


/*****************************************************************************
* FUNCTION: SortCorn1
* DESCRIPTION: Groups and sorts corners of a single building
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: uvcorn: corner coord's of a building
* OUTPUTS:
        CORN: ordered indices of corners
        ncorn: number of corners on each side
* CALLED BY: SortBldg1
* FUNCTION CALLS: SortCorn: sort corners of a single side
*****************************************************************************/
void SortCorn1(BAC1 uvcorn, int CORN[2][4], int ncorn[2])
{
        double u0[4],u1[4];
        int j0,j1,i,iu0[4],iu1[4];
        j0=0;
        j1=0;
        for(i=0;i<=3;i++) {
            if( uvcorn[1][i] > 0 ) {
                u1[j1] = uvcorn[0][i];
                iu1[j1] = i;
                j1++;
                }
            else {
                u0[j0] = uvcorn[0][i];
                iu0[j0] = i;
                j0++;
                }
            }
        if( j0 > 1 )
            SortCorn(j0,0,u0,iu0,CORN);
        else if(j0 > 0 )
            CORN[0][0] = iu0[0];

        if( j1 > 1 )
            SortCorn(j1,1,u1,iu1,CORN);
        else if(j1 > 0 )
            CORN[1][0] = iu1[0];

        ncorn[0] = j0-1;
        ncorn[1] = j1-1;
        return;
}


/*****************************************************************************
* FUNCTION: SortEdges
* DESCRIPTION: Groups and sorts building roof edges
* PROGRAMMER/DATE: Tom Russell, 10/27/91
```

```
* INPUTS: uvcorn: corner coord's of a building
* OUTPUTS:
        EDGE: ordered indices of roof edges of each level
        nedge: number of roof edges of each level
        *vmax1, *vmin1: max and min values of v
        *ivmax1, *ivmin1: indices of corners with max, min v values
* CALLED BY: SortBldg1
* FUNCTION CALLS: BoundBldg1: find max and min values of u and v
*************************************************************************/
void SortEdges(BAC1 uvcorn, int EDGE[3][2],
        int nedge[3], double *vmax1, double *vmin1,
        int *ivmax1, int *ivmin1)
{
        double umax,umin,vmax,vmin;
        int iumax,iumin,ivmax,ivmin;
        int im1,i2,i1,i0,i,ip2;
/* Init.: avoid later indexing by unassigned index */
        ivmax= 0;
        ivmin= 0;
        iumax= 0;
        iumin= 0;

        BoundBldg1(uvcorn,&umax,&umin,&vmax,&vmin,
                &iumax,&iumin,&ivmax,&ivmin);

        i0 = 0;
        i1 = 0;
        i2 = 0;
        for(i=0;i<=3;i++) {
            im1 = cyc4neg(i);
            if( uvcorn[1][im1] == uvcorn[1][i] ) {
            /* Special Case: edges parallel to line SP */
                EDGE[0][0] = i;
                EDGE[0][1] = cyc4neg(im1);
                if(uvcorn[0][im1] == umax) {
                    EDGE[2][0] = im1;
                    EDGE[1][0] = cyc4pos(i);
                    EDGE[2][1] = 0;/* Just a meaningless initialization here*/
                    EDGE[1][1] = 0;/* Just a meaningless initialization here*/
                    }
                else {
                    EDGE[2][0] = cyc4pos(i);
                    EDGE[1][0] = im1;
                    EDGE[2][1] = 0;/* Just a meaningless initialization here*/
                    EDGE[1][1] = 0;/* Just a meaningless initialization here*/
                    }
                nedge[0] = 1;
                nedge[1] = 0;
                nedge[2] = 0;
                *vmax1  = vmax;
                *vmin1 = vmin;
                *ivmax1 = ivmax;
                *ivmin1 = ivmin;
                return;
                }
            else {
                if(im1 == ivmin) {
                    if( uvcorn[0][i] > uvcorn[0][ivmin] ) {
                        EDGE[2][i2] = i;
                        i2++;
                        }
                    else {
                        EDGE[1][i1] = i;
                        i1++;
                        }
```

```
                }
            else {
            if(im1 == ivmax) {
                if( uvcorn[0][i] > uvcorn[0][ivmax] ) {
                    EDGE[2][i2] = i;
                    i2++;
                    }
                else {
                    EDGE[1][i1] = i;
                    i1++;
                    }
                }
            else {
            if(i == ivmin) {
                if( uvcorn[0][im1] > uvcorn[0][ivmin] ) {
                    EDGE[2][i2] = i;
                    i2++;
                    }
                else {
                    EDGE[1][i1] = i;
                    i1++;
                    }
                }
            else {
            if(i == ivmax) {
                if( uvcorn[0][im1] > uvcorn[0][ivmax] ) {
                    EDGE[2][i2] = i;
                    i2++;
                    }
                else {
                    EDGE[1][i1] = i;
                    i1++;
                    }
                }
            }
            }
            }
            }
        }
    nedge[0] = i0-1;
    nedge[1] = i1-1;
    nedge[2] = i2-1;
    *vmax1  = vmax;
    *vmin1 = vmin;
    *ivmax1 = ivmax;
    *ivmin1 = ivmin;
    return;
}


/*****************************************************************************
* FUNCTION: SortCorn
* DESCRIPTION: Uses a bubble-sort algorithm to sort corners of
        a single side of a building according to order of diffraction
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        jtot: number of corners on a particular side
        kc: index of side being considered
        u: corner coord's
        iu: indices of corners
* OUTPUTS: CORN: ordered indices of corners
* CALLED BY: SortCorn1
* FUNCTION CALLS: none
*****************************************************************************/
```

```
void SortCorn(int jtot, int kc, double u[4], int iu[4], int CORN[2][4])
{
        double TEMP;    /* holds corner coord */
        int k,i;        /* loop counters */
        int itmp;       /* holds corner index */
        for(k=0;k<=(jtot-2);k++) {
            for(i=(jtot-2);i>=k;i--)  {
                if( u[i] > u[i+1] ) {
                    TEMP = u[i];
                    itmp = iu[i];
                    u[i] = u[i+1];
                    iu[i] = iu[i+1];
                    u[i+1] = TEMP;
                    iu[i+1] = itmp;
                    }
                }
            CORN[kc][k] = iu[k];
            }
        CORN[kc][jtot-1] = iu[jtot-1];
        return;
}


/*************************************************************************
* FUNCTION: checkintermRoof
* DESCRIPTION: Returns a true value if the roof edge at Q still has a
        diffraction effect when the preceding roof edge on a connected
        building is considered.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P,Q: coord's of P and Q
        ybldg: building heights
        lam: wavelength
        zxcorn: coord's of all buildings
        iConn,jConn: edge and building indices of connected building
        iQ,jQ: edge and building indices of original building at Q
* OUTPUTS: none except return value
* CALLED BY: BldgDiff1
* FUNCTION CALLS:
        GetPntR2: get coord's of primary point R on earlier diffracting edge
        MoveTrans: place transmitter at point R, translate P,Q,and uvcorn
        checkFZ1: check for Fresnel zone clearance
*************************************************************************/
int checkintermRoof(ZXUVY P, ZXUVY Q, BAH ybldg,
        double lam, BAC zxcorn,
        int iConn, int jConn, int iQ, int jQ)
{
        double rso,rpo,y1,v1,v2,yorigin1; /* diffraction parameters */
        BAC1 uvcornr;           /* coord's of building with respect to R */
        ZXUVY R;                /* R, referenced to source, S */
        ZXUVY Pr,Qr;            /* P and Q referenced to R */

/* Get Point R */
        if( GetPntR2(Q,iConn,jConn,ybldg[jConn],zxcorn,R) ) ;
                MoveTrans(P,R,jQ,zxcorn,Pr,uvcornr);

/* Place trans at R on jConn,
        Find P, Q, and uvcorn referenced to R: Pr, Qr, uvcornr;
        where uvcornr holds the coords of Bldg jQ */

                if( checkFZ1(Pr,R[4],uvcornr,iQ,ybldg[jQ],lam,Qr) )
                        return(1); /* Both Edges diffract */
                else
                        return(0); /* R diffracts, but Q does not, move on */
```

```
}


/*************************************************************************
* FUNCTION: GetPntR2
* DESCRIPTION: Gets the coordinates of the primary point on a diffracting
        edge at a connected building.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        Q: coord's of diffracting edge
        iConn,jConn: edge and building indices of connected building
        ybldgConn: height of connected building
        zxcorn: building coord's
* OUTPUTS: R: coord's of point on connected building edge
* CALLED BY: checkintermRoof
* FUNCTION CALLS:
        BldgCoorduvQ1: rotates zx bldg coord's to uv system
        rEdgeCoords: finds the location of primary point, R on an edge
        PntCoordzxQ1: finds zx coord's based on uv coord's
*************************************************************************/
int GetPntR2(ZXUVY Q, int iConn, int jConn,
                double ybldgConn, BAC zxcorn, ZXUVY R)
{
        BAC1 uvcorn; /* This uvcorn holds coords of jConn,
                        referenced to S and Q */
/* Set field point at Q, look at point R */
        BldgCoorduvQ1(Q,jConn,zxcorn,uvcorn);
/* Resolve Bldg jConn into coord system based on P(prime) = Q */
        rEdgeCoords(Q,uvcorn,ybldgConn,iConn,R);
        PntCoordzxQ1(Q,R);
        return(1);
}


/*************************************************************************
* FUNCTION: checkFZ1
* DESCRIPTION: checks for diffraction by edge at Q, based on the source
        placed at an earlier edge, R. Returns a true response if
        the edge at Q does introduce significant diffraction
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P: coord's of observation point
        yt: transmitter height
        uvcorn: coord's of a single building
        i: edge index, also index of corner at one end of edge
        ybldg1: building height
        lam: wavelength
* OUTPUTS: Q: coord's of intermediate point
* CALLED BY: checkintermRoof
* FUNCTION CALLS:
        qEdgeCoords: finds the location of the primary point, Q on an edge
        FldPar1: computes some necessary diffraction field parameters
        cyc4neg: finds preceding corner by wrapping around if necessary
*************************************************************************/
int checkFZ1(ZXUVY P, double yt, BAC1 uvcorn, int i,
        double ybldg1, double lam,
        /* OUTPUTS */ ZXUVY Q )
{
        double yorigin;                 /* height of aperture plane origin */
        double Pnt55FrZn;               /* value of unit-less diffraction
                                parameters at 0.55 Fresnel zone clearance */
        double eta1;                    /* unit-less diffraction parameter */
        double diffac;                  /* intermediate factor */
        double rso,rpo,y1;              /* diffraction parameters in meters */
```

```
        int TRUE;                       /* return value: 1 for diffraction */

        TRUE = 0;
        Pnt55FrZn = 0.78;
        qEdgeCoords(P,uvcorn,ybldg1,i,Q);
        FldPar1(Q[2],P[2],yt,P[4],&rso,&rpo,&yorigin);
        y1 = ybldg1 - yorigin;
        if( (rpo > 5.0*lam) && (rso > 5.0*lam) ) { /* check diffraction */
                diffac = sqrt(2.0*(rso+rpo)/(lam*rso*rpo));
                eta1 = y1 * diffac;
/* Don't test xi1 or xi2  because of danger with second edge */
                        if( eta1 > -Pnt55FrZn )
                                TRUE = 1;
                }
        return(TRUE);
}


/*************************************************************************
* FUNCTION: checkintermCorn
* DESCRIPTION: Returns a true value if the currently considered corner
        still has a diffraction effect when the preceding corner
        is considered.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P,Q: coord's of observation point and intermediate point
        yt: transmitter height
        lam: wavelength
        LOWCORNLIM: lower limits of corners
        zxcorn: coord's of all buildings
        CORN: ordered indices of corners of a single building
        klr: index representing side: 0:left, 1:right
        kc: number of remaining corners - 1 (0: 1 more corner)
        j: building index
* OUTPUTS: none except for return value
* CALLED BY: CornDiff1
* FUNCTION CALLS:
        GetPntR: get coord's of primary point R on earlier diffracting edge
        MoveTrans: place transmitter at point R, translate P,Q,and uvcorn
        checkFZcorn: check for Fresnel zone clearance
*************************************************************************/
int checkintermCorn(ZXUVY P, ZXUVY Q, double yt,
        double lam, LCL LOWCORNLIM, BAC zxcorn,
        int CORN[2][4], int klr, int kc, int j)
{
        double rso,rpo,y1,v1,v2,yorigin1; /* diffraction field parameters */
        BAC1 uvcornr;           /* coord's of building referenced to R */
        ZXUVY R;                /* R referenced to S */
        ZXUVY Pr,Qr;            /* P, Q referenced to R */
        int iCorn;              /* index of current corner */

        if(!kc)
            return(-2); /* No more corners */

/* Get Point R */
        if( GetPntR(Q,CORN,kc,klr,
                        zxcorn,LOWCORNLIM,j,yt,lam,R) ) {
                        /* Then R does diffract energy on way to Q */
                MoveTrans(P,R,j,zxcorn,Pr,uvcornr);
/* Place trans at R, find P, Q, and uvcorn referenced to R: Pr, Qr, uvcornr */

                iCorn = CORN[klr][kc];
                Qr[2] = uvcornr[0][iCorn];
                Qr[3] = uvcornr[1][iCorn];
```

```
                Qr[4] = Q[4];
                if( checkFZcorn(Pr,Qr,R[4],lam,
                                LOWCORNLIM[iCorn][j],klr,
                                &rso,&rpo,&y1,&v1,&v2) )
                        return(1); /* Both Edges diffract */
                else
                        return(0); /* R diffracts, but Q does not, move on */
                }
        return(1); /* R does not diffract, Q of course still does */
}


/*************************************************************************
* FUNCTION: GetPntR
* DESCRIPTION: Gets the coord's of the primary point on a corner
        preceding the current corner.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        Q: intermediate point on diffracting edge
        CORN: ordered indices of corners
        kc: number of remaining corners
        klr index representing side
        zxcorn: coord's of all buildings
        LOWCORNLIM: lower limits of corners
        j: building index
        yt: transmitter height
        lam: wavelength
* OUTPUTS: R: coord's of intermediate point on diffracting edge prior to Q
* CALLED BY: checkintermCorn
* FUNCTION CALLS:
        BldgCoorduvQ1: rotates zx bldg coord's to uv system
        checkFZcorn: check for Fresnel zone clearance
*************************************************************************/
int GetPntR(ZXUVY Q, int CORN[2][4], int kc, int klr, BAC zxcorn,
        LCL LOWCORNLIM, int j,
        double yt, double lam, ZXUVY R)
{
        BAC1 uvcorn;                    /* coord's referenced to S and Q */
        double rso,rpo,y1,v1,v2;        /* diffraction field parameters */
        int iCorn;                      /* index of preceding corner */

        if(!kc)
            return(0);

        iCorn = CORN[klr][kc-1];
/* First set field point at Q, look at point R */
        BldgCoorduvQ1(Q,j,zxcorn,uvcorn);
        R[2] = uvcorn[0][iCorn];
        R[3] = uvcorn[1][iCorn];
        PntCoordzxQ1(Q,R);
        if( checkFZcorn(Q,R,yt,lam,
                                LOWCORNLIM[iCorn][j],klr,
                                &rso,&rpo,&y1,&v1,&v2) )
                return(1);
        else
                return(0);  /* R does not diffract */
}


/*************************************************************************
* FUNCTION: MoveTrans
* DESCRIPTION: Move transmitter to R, compute building and P
        coordinates referenced to this new source point
```

```
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P,R: point coord's referenced to S,P system
        j: building index
        zxcorn: building coord's, referenced to S
* OUTPUTS:
        Pr: P referenced to R instead of S
        uvcorn: building coord's referenced to R and P
* CALLED BY: checkintermCorn
* FUNCTION CALLS: BldgCoorduv1a: rotates zx bldg coord's to uv system
*****************************************************************************/
void MoveTrans(ZXUVY P, ZXUVY R, int j,
        BAC zxcorn, ZXUVY Pr, BAC1 uvcorn)
        /* Outputs Pr coords in z,x,u */
{
        BAC1 zxC;       /* zx building coord's referenced to R */
        int i,k;        /* loop counters */

        for(k=0;k<=1;k++) {
            for(i=0;i<=3;i++)
                zxC[k][i] = zxcorn[k][i][j] - R[k];
            Pr[k] = P[k] - R[k]; /* Pr is P referenced to R */
            }

        Pr[2] = sqrt(Pr[0]*Pr[0] + Pr[1]*Pr[1]);
        Pr[3] = 0.0;
        Pr[4] = P[4];
        BldgCoorduv1a(Pr,zxC,uvcorn);
        return;
}


/*****************************************************************************
* FUNCTION: checkFZcorn
* DESCRIPTION: Returns true if there is significant Fresnel zone
        obstruction by a particular corner
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P,Q: coord's of observation and intermediate points
        yt: transmitter height
        lam: wavelength
        LOWCORNLIM1: lower limit of a single corner
        klr: indicates side of building, left(0) or right(1)
* OUTPUTS:
        *rso,*rpo: distances from aperture plane origin to source and P
        *y1: vertical distance from origin to lower bound of aperture
        *v1,*v2: horizontal distances to left and right edges of aperture
* CALLED BY: checkintermCorn, GetPntR, CornDiff1
* FUNCTION CALLS: FldPar1: computes some necessary diffraction parameters
*****************************************************************************/
int checkFZcorn(ZXUVY P, ZXUVY Q, double yt, double lam,
        double LOWCORNLIM1, int klr,
        double *rso, double *rpo, double *y1, double *v1, double *v2)
{
        double rso1,rpo1;       /* distances of origin to source and P */
        double yorigin1;        /* height of origin */
        double xi1,xi2,diffac;  /* diffraction parameters */
        double Pnt55FrZn;       /* value of a diff parameter corresponding
                                        to 0.55 Fresnel zone clearance */
        Pnt55FrZn = 0.78;
        FldPar1(Q[2],P[2],yt,P[4],&rso1,&rpo1,&yorigin1);
        if( klr ) {
                *v1 = Q[3];
                *v2 = 10000.0;
                }
```

```
        else  {
                *v1 = -10000.0;
                *v2 = Q[3];
                }
        *rso = rso1;
        *rpo = rpo1;
        *y1 = LOWCORNLIM1 - yorigin1;
        Q[4] = yorigin1;
        if( (*rpo > 5.0*lam) && (*rso > 5.0*lam) ) {
            diffac = sqrt(2.0*(rso1+rpo1)/(lam*rso1*rpo1));
            xi1 = *v1 * diffac;
            xi2 = *v2 * diffac;
            if( klr ) {
                if( xi1 > -Pnt55FrZn )
                        return(1);
                else
                        return(0);
                }
            else {
                if( xi2 < Pnt55FrZn )
                        return(1);
                else
                        return(0);
                }
            }
        return(1);/* If, for example, rpo is negative, return 1 and let
                        checkNotBlock return a 0; a false return from this
                        routine signifies free space prop, no diff. */
}


/*****************************************************************************
* FUNCTION: checkNotBlock
* DESCRIPTION: Returns false for such severe diffractive shadowing that
        the field can be considered blocked
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: rso,rpo,y1,v1,v2: diffraction parameters defined elsewhere
        lam: wavelength
* OUTPUTS: none except return value
* CALLED BY: BldgDiff1, BldgDiff2, CornDiff1
* FUNCTION CALLS: none
*****************************************************************************/
int checkNotBlock(double rso, double rpo, double y1,
        double v1, double v2, double lam)
{
        double MAXPAR;  /* maximum value of the unit-less diffraction
                           parameters, beyond which the field is considered
                           blocked */
        double eta1,xi1,xi2; /* unit-less diffraction parameters */
        double diffac;  /* intermediate multiplier */

        MAXPAR = 20.0;
        if( (rpo > 5.0*lam) && (rso > 5.0*lam)
                    && (v2 > v1) ) {
            diffac = sqrt(2.0*(rso+rpo)/(lam*rso*rpo));
            eta1 = y1 * diffac;
            xi1 = v1 * diffac;
            xi2 = v2 * diffac;
            if( (eta1 < MAXPAR) && (xi1 < MAXPAR) && (xi2 > -MAXPAR) )
                return(1);
            else
                return(0);
            }
        else
            return(0);
```

```
}


/*************************************************************************
* FUNCTION: checkFZ2
* DESCRIPTION: Checks for explicit Fresnel zone blockage by either of the
        two nearest building faces; computes diffraction parameters.
        Returns true if either one of the edges at Q obstruct the first
        Fresnel zone sufficiently to cause significant diffraction.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        Blev: level of diffraction, 1:leading edges, 2:trailing edges
        P: field observation point
        yt: transmitter height
        EDGE: ordered indices of roof edges
        nedge: number of roof edges of each level
        uvcorn: coord's of a single building
        ybldg1: height of a single building
        lam: wavelength
* OUTPUTS:
        Q1,Q2: primary intermed. points on two roof edges of same level
        rso,rpo,y1,v1,v2: parameters needed to evaluate diffraction integrals
* CALLED BY: BldgDiff1, BldgDiff2
* FUNCTION CALLS:
        qEdgeCoords: finds the location of the primary point Q on an edge
        FldPar1: computes some necessary diffraction field parameters
*************************************************************************/
int checkFZ2(int Blev, ZXUVY P, double yt, int EDGE[3][2], int nedge[3],
        BAC1 uvcorn, double ybldg1, double lam,
        /* OUTPUTS */ ZXUVY Q1, ZXUVY Q2, double rso[2], double rpo[2],
        double y1[2], double v1[2], double v2[2])
{
        double yorigin;         /* height of origin */
        double Pnt55FrZn;       /* value of unit-less diff para's at .55 clrnce */
        double eta1,xi1,xi2;    /* unit-less diffraction parameters */
        double diffac;          /* intermediate factor */
        ZXUVY Q;                /* coord's of intermediate point */
        int i,im1;              /* indices of corners defining the edge */
        int k,n;                /* loop counters */
        int TRUE;               /* return value: 1 for diffraction */

        TRUE = 0;
        Pnt55FrZn = 0.78;

        for(k=0;k<=nedge[Blev];k++)  {
            i = EDGE[Blev][k];
            qEdgeCoords(P,uvcorn,ybldg1,i,Q);
            FldPar1(Q[2],P[2],yt,P[4],&rso[k],&rpo[k],&yorigin);
            for(n=0;n<=4;n++) {
                if(!k)
                    Q1[n] = Q[n];
                else
                    Q2[n] = Q[n];
                }
            im1 = cyc4neg(i);

            if( uvcorn[1][im1] < uvcorn[1][i] ) {
                v1[k] = uvcorn[1][im1];
                v2[k] = uvcorn[1][i];
                }
            else  {
                v1[k] = uvcorn[1][i];
                v2[k] = uvcorn[1][im1];
                }
            y1[k] = ybldg1 - yorigin;
```

```
            if( (rpo[k] > 5.0*lam) && (rso[k] > 5.0*lam) ) {
                diffac = sqrt(2.0*(rso[k]+rpo[k])/(lam*rso[k]*rpo[k]));
                eta1 = y1[k] * diffac;
                xi1 = v1[k] * diffac;
                xi2 = v2[k] * diffac;
                if( (xi1 < Pnt55FrZn) && (xi2 > -Pnt55FrZn) )
                        if( eta1 > -Pnt55FrZn )
                                TRUE = 1; /* Don't return yet,
                                             need v1[2],v2[2]... */
                }
            } /* end for */
        return(TRUE);
}


/*************************************************************************
* FUNCTION: checkFZ4
* DESCRIPTION: checks for sufficient Fresnel zone obstruction by any
        of the four vertical faces of a building that the building as a
        whole may be considered a diffracting building.
        Returns true if it is a diffracting building.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        up: distance from S to P
        yp: height of P
        yt: height of S
        uvcorn: building coord's
        ybldg1: building height
        lam: wavelength
* OUTPUTS: none except for return value
* CALLED BY: NearestDiffBldg
* FUNCTION CALLS:
        qcoords: locates primary point, Q on a diffracting edge
        FldPar1: computes some necessary diffraction field parameters
*************************************************************************/
int checkFZ4(double up, double yp,
                double yt, BAC1 uvcorn, double ybldg1, double lam)
/* Called by NearestDiffBldg to check for an edge that diffracts */
{
        double yorigin;         /* height of origin */
        double Pnt55FrZn;       /* value of unit-less diff para's at .55 clrnce */
        double eta1,xi1,xi2;    /* unit-less diffraction parameters */
        double diffac;          /* intermediate factor */
        double uq,vq,yq;        /* coord's of intermediate point */
        double rso,rpo,y1,v1,v2;/* distances defined with respect to origin */
        int i,im1;              /* indices of corners defining the edge */
        int k,n;                /* loop counters */
        Pnt55FrZn = 0.78;
        for(i=0;i<=3;i++)  {
            qcoords(uvcorn,ybldg1,i,&uq,&vq,&yq);
            FldPar1(uq,up,yt,yp,&rso,&rpo,&yorigin);
            im1 = cyc4neg(i);
            if( uvcorn[1][im1] < uvcorn[1][i] ) {
                v1 = uvcorn[1][im1];
                v2 = uvcorn[1][i];
                }
            else  {
                v1 = uvcorn[1][i];
                v2 = uvcorn[1][im1];
                }
            y1 = ybldg1 - yorigin;
            if( (rpo > 5.0*lam) && (rso > 5.0*lam) ) {
                diffac = sqrt(2.0*(rso+rpo)/(lam*rso*rpo));
                eta1 = y1 * diffac;
```

```
                xi1 = v1 * diffac;
                xi2 = v2 * diffac;
                if( (xi1 < Pnt55FrZn) && (xi2 > -Pnt55FrZn) )
                        if( eta1 > -Pnt55FrZn )
                                return(1);
                }
            } /* end for */
        return(0);
}


/*****************************************************************************
* FUNCTION: qcoords
* DESCRIPTION: computes u,v,y coordinates of the primary point, Q,
        on a diffracting edge
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        uvcorn: horiz coordinates of a single building
        ybldg1: height of a single building
        i: edge index
* OUTPUTS:
        *uq,*vq,*yq: coord's of point Q
* CALLED BY: checkFZ4
* FUNCTION CALLS: sign: returns 1 for positive, -1 for negative
                  uintercept: finds intersection of bldg edge with u axis
*****************************************************************************/
void qcoords(BAC1 uvcorn, double ybldg1,
        int i, double *uq, double *vq, double *yq)
/* computes u,v,y coordinates of Q, where i is edge index */
{
        double vq1,uq1;         /* coord's of Q */
        int im1;                /* second corner index defining edge */
        im1 = cyc4neg(i);
        if( sign(uvcorn[1][im1]) == - sign(uvcorn[1][i]) ) {
                uintercept(uvcorn,i,&uq1);
                vq1 = 0;
                }
        else {
                if( fabs(uvcorn[1][im1]) < fabs(uvcorn[1][i]) ) {
                        uq1 = uvcorn[0][im1];
                        vq1 = uvcorn[1][im1];
                        }
                else    {
                        uq1 = uvcorn[0][i];
                        vq1 = uvcorn[1][i];
                        }
                }
        *uq = uq1;
        *vq = vq1;
        *yq = ybldg1;
        return;
}


/*****************************************************************************
* FUNCTION: rEdgeCoords
* DESCRIPTION: computes u,v,y and z,x coordinates of R, the primary
        point on an edge, given the uv coords of the edge,
        and the coords of Q, where Q[3] (the v coord) is not
        necessarily 0, as is P[3]
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        Q: the primary point on a later edge
        uvcorn: coord's of a single building
```

```
        ybldg: height of a single building
        i: edge index, also index of one corner defining edge
* OUTPUTS: R: coord's of the first point in the aperture (primary point)
                at a diffracting edge
* CALLED BY: GetPntR2
* FUNCTION CALLS:
        sign: returns +1 or -1
        uintercept: finds intersection of building edge with u axis
        PntCoordzxQ1: finds zx coord's based on uv coord's
*************************************************************************/
void rEdgeCoords(ZXUVY Q, BAC1 uvcorn, double ybldg1, int i, ZXUVY R)
{
        int im1;                /* second corner index */
        im1 = cyc4neg(i);
        if( sign(uvcorn[1][im1]) == - sign(uvcorn[1][i]) ) {
                uintercept(uvcorn,i,&R[2]);
                R[3] = 0;
                }
        else {
                if( fabs(uvcorn[1][im1]) < fabs(uvcorn[1][i]) ) {
                        R[2] = uvcorn[0][im1];
                        R[3] = uvcorn[1][im1];
                        }
                else    {
                        R[2] = uvcorn[0][i];
                        R[3] = uvcorn[1][i];
                        }
                }
        PntCoordzxQ1(Q,R);
        R[4] = ybldg1;
        return;
}


/*************************************************************************
* FUNCTION: qEdgeCoords
* DESCRIPTION: computes u,v,y and z,x coordinates of Q, the primary
        point on an edge, given the uv coords of the edge,
        and the coords of P, where P[3] (the v coord) is always zero
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P: the observation point
        uvcorn: coord's of a single building
        ybldg: height of a single building
        i: edge index, also index of one corner defining edge
* OUTPUTS: Q: coord's of the first point in the aperture (primary point)
                at a diffracting edge
* CALLED BY: checkFZ1, checkFZ2
* FUNCTION CALLS:
        sign: returns +1 or -1
        uintercept: finds intersection of building edge with u axis
        PntCoordzx1: finds zx coord's based on uv coord's
*************************************************************************/
void qEdgeCoords(ZXUVY P, BAC1 uvcorn, double ybldg1, int i, ZXUVY Q)
/* computes u,v,y and z,x coordinates of Q, the first point on the edge,
        given the uv coords of the edge, and the coords of P */
{
        int im1;                /* second corner index */
        im1 = cyc4neg(i);
        if( sign(uvcorn[1][im1]) == - sign(uvcorn[1][i]) ) {
                uintercept(uvcorn,i,&Q[2]);
                Q[3] = 0;
                }
        else {
                if( fabs(uvcorn[1][im1]) < fabs(uvcorn[1][i]) ) {
```

```
                        Q[2] = uvcorn[0][im1];
                        Q[3] = uvcorn[1][im1];
                        }
                else    {
                        Q[2] = uvcorn[0][i];
                        Q[3] = uvcorn[1][i];
                        }
                }
        PntCoordzx1(P,Q);
        Q[4] = ybldg1;
        return;
}


/*************************************************************************
* FUNCTION: NearestDiffBldg
* DESCRIPTION: The nearest building that obstructs the first Fresnel zone
        sufficiently to cause significant diffraction effects is identified
        by this routine.  If there are no diffracting buildings a flag
        is returned to indicate this.
        First the nearest bldg to obstruct in the horizontal plane only
        is found, and checked for Fresnel zone clearance when all 3 dimensions
        are considered. If this building does not diffract then it is
        excluded and the process is repeated.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P: observation point
        yt: transmitter height
        ybldg: heights of all buildings
        lam: wavelength
        zxcorn: coord's of all buildings
        umax: maximum u values of each building
        OBSTRUCT: flags indicating obstruction in horizontal plane
        TOOSHORT: flags indicating bldgs too short to diffract significantly
* OUTPUTS:
        uvcorn: coord's of a single building
        *NOBLDGS: true if no diffracting buildings found
        *j: index of nearest diffracting building
* CALLED BY: NearestDiffBldg, FindFieldP
* FUNCTION CALLS:
        NearestBldg: finds nearest building that obstructs in horiz plane
        BldgCoorduv1: get bldg coord's in uv system
        checkFZ4: checks for Fresnel zone obstruction
        NearestDiffBldg: a recursive call needed when a bldg is too short
*************************************************************************/
void NearestDiffBldg(ZXUVY P, double yt, BAH ybldg, double lam,
        BAC zxcorn, double umax[Nb], BAF OBSTRUCT, BAF TOOSHORT,
        BAC1 uvcorn, int *NOBLDGS, int *j)
{
        int NOBLDGStmp;         /* temp flag */
        int jtmp,jnear;         /* index of nearest bldg */

        NearestBldg(OBSTRUCT,TOOSHORT,umax,&jnear);
        if( jnear == -99 ) {
            *NOBLDGS = 1;
            return;
            }
/* Check building for height, blockage */
        BldgCoorduv1(P,jnear,zxcorn,uvcorn);
        if( checkFZ4(P[2],P[4],yt,uvcorn,ybldg[jnear],lam) ) {
            *j = jnear;
            return;
            }
        else {
            TOOSHORT[jnear] = 1;  /* Also signifies too far away */
```

```
            NOBLDGStmp = 0;
            NearestDiffBldg(P,yt,ybldg,lam,zxcorn,umax,
                OBSTRUCT,TOOSHORT,uvcorn,&NOBLDGStmp,&jtmp);
            *j = jtmp;
            *NOBLDGS = NOBLDGStmp;
            }
        return;
}


/*************************************************************************
* FUNCTION: NearestBldg
* DESCRIPTION: Using only the horizontal-plane projection of the signal
        path, the nearest building to obstruct this path is identified.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        OBSTRUCT: flags indicating obstruction in horizontal plane
        TOOSHORT: flags indicating bldgs too short to diffract significantly
        umax: maximum u values of each building
* OUTPUTS:
        *jnear: index of nearest obstructing building
* CALLED BY: NearestDiffBldg
* FUNCTION CALLS: none
*************************************************************************/
void NearestBldg(BAF OBSTRUCT, BAF TOOSHORT, double umax[Nb], int *jnear)
{
        double unear;
        int j;
        unear = 0;
        *jnear = -99;
        for(j=0;j<=(Nb-1);j++) {
                if( (OBSTRUCT[j]) && !TOOSHORT[j] ) {
                        if( umax[j] > unear ) {
                                unear = umax[j];
                                *jnear = j;
                                }
                        }
                }
        return;
}


/*************************************************************************
* FUNCTION: Sift
* DESCRIPTION: Finds and flags all of the buildings obstructing the
        horizontal projection of the radio path
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        P: observation point
        zxcorn: coord's of all buildings
        MASK: flags for disregard of specific buildings
* OUTPUTS:
        OBSTRUCT:
        umax:
        *NOBLDGS:
* CALLED BY: FindFieldP
* FUNCTION CALLS:
        BldgCoorduv1: finds uv coord's of a building
        BoundBldg1: finds max and min values of u and v
        sign: returns +1 or -1 based on sign of input
*************************************************************************/
void Sift(ZXUVY P, BAC zxcorn, BAF MASK,
        BAF OBSTRUCT, double umax[Nb], int *NOBLDGS)
{
        BAC1 uvcorn;
```

```
        double umin,vmax,vmin,MAXDIST;
        int j,i1,i2,i3,i4;
        MAXDIST = 2.5;
        *NOBLDGS = 1;
        for(j=0;j<=(Nb-1);j++) {
            OBSTRUCT[j]=0;
            if( !MASK[j] ) {
                BldgCoorduv1(P,j,zxcorn,uvcorn);
                BoundBldg1(uvcorn,&umax[j],&umin,
                        &vmax,&vmin,&i1,&i2,&i3,&i4);
                if( (umax[j] > 0) && (umin < P[2]) )
                    if( sign(vmax) == -sign(vmin) ) {
                        OBSTRUCT[j] = 1;
                        *NOBLDGS = 0;
                        }
                    else
                        if( (vmax > -MAXDIST) && (vmin < MAXDIST) ) {
                            OBSTRUCT[j] = 1;
                            *NOBLDGS = 0;
                            }
                }
            }
        return;
}


/*****************************************************************************
* FUNCTION: cyc4neg
* DESCRIPTION: cycles around the four corners in a negative direction
        to find the next corner
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: n: corner index
* OUTPUTS: return value
* CALLED BY: many fcn's
* FUNCTION CALLS: none
*****************************************************************************/
int cyc4neg(int n)
{
        int nm1;
        nm1 = n-1;
        if( nm1 < 0 )
                nm1 = nm1 + 4;
        return(nm1);
}


/*****************************************************************************
* FUNCTION: cyc4pos
* DESCRIPTION: cycles around the four corners in a positive direction
        to find next corner
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: n: corner index
* OUTPUTS: return value
* CALLED BY: many fcn's
* FUNCTION CALLS: none
*****************************************************************************/
int cyc4pos(int n)
{
        int np1;
        np1 = n+1;
        if( np1 > 3 )
                np1 = np1 - 4;
        return(np1);
}
```

```
/*************************************************************************
* FILE NAME: EVALFLD.C
* CONTENTS:
        EvalField
        EvalIntegralx
        EvalIntegraly
        fieldcomp
        EvalIntegralyP1
        EvalIntegralyP2
        SegmentCompute
        intQa
        intQb
        intQc
        intLa
        intLb
        intLc
        coeff
        IntermCoeff
        CoeffQuadPhs
        CoeffLinPhs
        FC
        FS
        fcos
        fsin
        FCasympt
        FSasympt
        sign
* DESCRIPTION:
        This file contains routines for evaluting the diffraction field
*************************************************************************/
#include <mdsrch.h>
#include <mdtypes.h>
void EvalField(double rso, double rpo, double y1,
        double x1, double x2, double lam, int N1, double D[2], DA APDAT,
        double *real, double *imag, double *ampdb, double *phs);
void EvalIntegralx(double x1, double x2, double rso, double rpo,
        double lam, double *ampx, double *phsx );
void EvalIntegraly(double y1, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *ampy, double *phsy);
void fieldcomp(double rso, double rpo, double y1,
          double x1, double x2, double lam,
          double *real, double *imag, double *ampdb, double *phs);
void EvalIntegralyP1(double y0, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *realy, double *imagy);
void EvalIntegralyP2(double y0, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *realy, double *imagy);
void SegmentCompute(int QUADPHS, double u1, double u2, int eps,
        double psi, double gambet, double a, double b, double c,
        double *amp, double *phs);
void intQa(double u1, double u2, int eps,
        double a, double *reala, double *imaga);
void intQb(double u1, double u2, int eps,
        double b, double *realb, double *imagb);
void intQc(double u1, double u2, int eps,
        double c, double *realc, double *imagc);
void intLa(double u1, double u2, double a,
                double *reala, double *imaga);
void intLb(double u1, double u2, double b,
                double *realb, double *imagb);
void intLc(double u1, double u2, double c,
                double *realc, double *imagc);
void coeff(int CALL, double Anm1, double An, double Anp1,
           double Pnm1, double Pn, double Pnp1, double t1,
           double t2, double *psi, int *eps, double *gambet,
           double *a, double *b, double *c, double *u1,
```

```
          double *u2, int *QUADPHS);
void IntermCoeff(double fnm1, double fn, double fnp1,
                 double *a, double *b, double *c);
void CoeffQuadPhs(double A, double B, double C,
        double beta, double gamma, double t1, double t2,
        double *a, double *b, double *c, double *u1, double *u2);
void CoeffLinPhs(double A, double B, double C,
        double beta, double t1, double t2,
        double *a, double *b, double *c, double *u1, double *u2);
double FC(double b);
double FS(double b);
double fcos(double x);
double fsin(double x);
double FCasympt(double x);
double FSasympt(double x);
int sign(double x);

extern FILE *filefld;
extern FILE *fileint;
extern int Printfld;
extern int Printint;


/*****************************************************************************
* FUNCTION: EvalField
* DESCRIPTION: Evaluates the multiple diffraction field at a single point
        by integrating over the aperture bounding the diffracting edge
        directly causing diffraction at the observation point P. The field
        samples in the aperture characterize the diffraction by a previous
        obstacle, necessitating the multiple diffraction formulation.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        rso,rpo,y1,x1,x2: geometrical inputs to diffraction integral,
                (see MS thesis of Tom Russell for more details),
                rso: (s in thesis) distance from origin of aperture plane
                     to source (transmitter)
                rpo: (p in thesis) distance from origin of aperture plane
                     to observation point P
                y1: vertical distance from origin to lower bound of aperture
                    (upper bound, y2, always taken as positive infinity)
                x1,x2: distances from origin to left and right sides of
                        aperture
        NOTE: the inputs to y1 and (x1,x2) assume roof-edge diffraction,
              for corner-edge diffraction, the axes are flipped and
              y1 represents the corner edge, and (x1,x2) represent the
              bottom and top, respectively, of the aperture.
        lam: wavelength
        N1: number of field samples in aperture Region 1
        D: sampling intervals, Regions 1 and 2
        APDAT: amplitude and phase samples of field in aperture bounding
                diffracting edge
* OUTPUTS:
        *real,*imag,*amp,*phs: field descriptions
* CALLED BY: computediff, computeYfield, computeXfield
* FUNCTION CALLS:
        EvalIntegralx: evaluates integral over x dimension (or whichever
                       dimension does not involve explicit field samples)
        EvalIntegraly: evaluates integral over y dimension (or whichever
                       dimension is characterized by field samples)
*****************************************************************************/
void EvalField(double rso, double rpo, double y1,
        double x1, double x2, double lam, int N1, double D[2],
        DA APDAT, double *real, double *imag, double *amp, double *phs)
{
```

```
        double phs1,amp1;       /* factors outside of the integrals */
        double ampx,phsx;       /* results of integration over x dimension */
        double ampy,phsy;       /* results of integration over y dimension */

        phs1 = (2.0*pi/lam)*rpo - (pi/2.0);
        amp1 =  sqrt(rso/(2.0*lam*rpo*(rso+rpo)));

        EvalIntegralx(x1,x2,rso,rpo,lam,&ampx,&phsx);

        EvalIntegraly(y1,rpo,lam,N1,D,APDAT,&ampy,&phsy);

/*  Multiply to get the total field   */
        amp1 = amp1 * ampy * ampx;
        phs1 = phs1 + phsy + phsx;
        *real = amp1 * cos(phs1);
        *imag = amp1 * sin(phs1);
        *amp = amp1;
        *phs = phs1;
        return;
}


/*****************************************************************************
* FUNCTION: EvalIntegralx
* DESCRIPTION: Evaluates the integral over the dimension of the aperture
        that is not explicitly characterized by field samples, because
        variation in this dimension does not cause a significant change
        in degree of shadowing by the previous obstacle.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        rso,rpo,x1,x2: geometrical inputs to diffraction integral,
                (see header for EvalField)
        lam: wavelength
* OUTPUTS:
        *ampx,*phsx: complex result of integration
* CALLED BY: EvalField
* FUNCTION CALLS:
        FC: Evaluates Fresnel cosine integral
        FS: Evaluates Fresnel sine integral
*****************************************************************************/
void EvalIntegralx(double x1, double x2, double rso, double rpo,
        double lam, double *ampx, double *phsx )
{
        double diffac;          /* intermediate transformation factor */
        double xi1,xi2;         /* unit-less diffraction parameters,
                                   limits on varable of integration for
                                   Fresnel integrals */
        double cxi1,cxi2,sxi1,sxi2; /* real (c) and imaginary (s) parts
                        of Fresnel integrals evaluated at each of
                        the two limits */
        double cxi,sxi; /* total real and imag parts of the Fresnel integral */

        int INFINITEX;  /* special flag for neglecting x dimension */

        INFINITEX = 0;  /* X-dimension of region assumed infinite
                                in each direction */
        if(INFINITEX) {
          *ampx = sqrt(2.0);
          *phsx = pi/4.0;
          return;
        }

        diffac = sqrt(2.0*(rso+rpo)/(lam*rso*rpo));
        xi1 = x1 * diffac;
```

```
        xi2 = x2 * diffac;

/*    Find the integral over the x-dimension */
        if(xi1 > xi2) {
          printf("ERROR: x2 < x1 \n");
          return;  }

        if(xi1 < -60.0) {  /* 60 is used as the limit on the diffraction
                              parameter for which the integral needs
                              to be evaluated, this number can probably
                              be reduced */
          cxi1 = -0.5;
          sxi1 = -0.5;  }
        else if(xi1 > 60.0) {
          cxi1 = 0.0;
          sxi1 = 0.0;  }
        else  {
          cxi1 = FC(xi1);
          sxi1 = FS(xi1);  }

        if(xi2 > 60.0) {
          cxi2 = 0.5;
          sxi2 = 0.5;  }
        else if(xi2 < -60.0) {
          cxi2 = 0.0;
          sxi2 = 0.0;  }
        else  {
          cxi2 = FC(xi2);
          sxi2 = FS(xi2);  }

        cxi = cxi2 - cxi1;
        sxi = sxi2 - sxi1;

        *ampx = sqrt( cxi*cxi + sxi*sxi );
        *phsx = atan2( sxi, cxi );
        return;
}



/*************************************************************************
* FUNCTION: EvalIntegraly
* DESCRIPTION: Numerically evaluates the multiple diffraction integral
        over a region of the aperture that is explicitly characterized
        by field samples.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        y1, rpo: distances from origin to lower bound of aperture and to P
        lam: wavelength
        N1: number of samples in Region 1
        D: sampling intervals, Regions 1 and 2
        APDAT: field samples in aperture, amp and phs
* OUTPUTS:
        *amp,*phs: complex result of integration
* CALLED BY: EvalField
* FUNCTION CALLS:
        EvalIntegralyP1: Integrates over the finely-sampled Region 1
        EvalIntegralyP2: Integrates to infinity, using three Region 2 samples
*************************************************************************/
void EvalIntegraly(double y1, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *ampy, double *phsy)
{
        double realy1,imagy1;   /* complex result of Region 1 integration */
        double realy2,imagy2;   /* complex result of Region 2 integration */
        double REAL,IMAG;       /* complex sum of the two results */
```

```
        double y2;              /* value of y at first sample of Region 2 */

/* N+1 samples (numbered 0,N) implies N intervals and N sets of coefficients -
    leading to N different amplitude and phase interpolation fcn's */

        y2 = y1 + N1*D[0];
        EvalIntegralyP1(y1,rpo,lam,N1,D,APDAT,&realy1,&imagy1);
        EvalIntegralyP2(y2,rpo,lam,N1,D,APDAT,&realy2,&imagy2);
        REAL = realy1 + realy2;
        IMAG = imagy1 + imagy2;
        *ampy = sqrt( REAL*REAL + IMAG*IMAG );
        *phsy = atan2( IMAG,REAL );
        return;
}


/*************************************************************************
* FUNCTION: fieldcomp
* DESCRIPTION: Evaluates single diffraction at an observation point P
        by integrating over a rectangular aperture, whose field
        variation is described analytically rather than sampled
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        rso,rpo,y1,x1,x2: geometrical inputs to diffraction integral,
                (see header for EvalField)
        lam: wavelength
* OUTPUTS:
        *real,*imag,*amp,*phs: field descriptions
* CALLED BY: computediff, computeYfield, computeXfield
* FUNCTION CALLS:
        FC: Evaluates Fresnel cosine integral
        FS: Evaluates Fresnel sine integral
*************************************************************************/
void fieldcomp(double rso, double rpo, double y1,
          double x1, double x2, double lam,
          double *real, double *imag, double *amp, double *phs)
{
        double xi1,xi2,eta1;    /* unit-less diffraction parameters */
        double ceta,seta,ampy,phsy; /* results of integral over y dimension */
        double cxi,sxi,ampx,phsx;   /* results of integral over x dimension */
        double ampfs;           /* free space path loss */
        double amp1,phs1;       /* factors outside the integrals */
        double diffac;          /* intermediate transformation factor */
        double ceta1,ceta2,seta1,seta2; /* intermediate integration results */
        double cxi1,cxi2,sxi1,sxi2;     /* intermediate integration results */

        diffac = sqrt(2.0*(rso+rpo)/(lam*rso*rpo));
        xi1 = x1 * diffac;
        xi2 = x2 * diffac;
        eta1 = y1 * diffac;

        phs1 = (2.0*pi/lam)*(rso + rpo) - (pi/2.0);
        ampfs = 1.0 / (rso + rpo);
        amp1 =  ampfs / 2.0;

/*    Find the integral over the y-dimension   */
        if(eta1 < -60.0) {
            ceta1 = -0.5;
            seta1 = -0.5;
            }
        else
            if(eta1 > 60.0) {
                ceta1 = 0.0;
                seta1 = 0.0;
                }
```

```
            else  {
                ceta1 = FC(eta1);
                seta1 = FS(eta1);
                }

        ceta2 = 0.5;
        seta2 = 0.5;

        ceta = ceta2 - ceta1;
        seta = seta2 - seta1;
        ampy = sqrt( ceta*ceta + seta*seta );
        phsy = atan2( seta, ceta );

/*      Find the integral over the x-dimension */
        if(xi1 < -60.0) {
          cxi1 = -0.5;
          sxi1 = -0.5;  }
        else if(xi1 > 60.0) {
          cxi1 = 0.0;
          sxi1 = 0.0;  }
        else  {
          cxi1 = FC(xi1);
          sxi1 = FS(xi1);  }

        if(xi2 > 60.0) {
          cxi2 = 0.5;
          sxi2 = 0.5;  }
        else if(xi2 < -60.0) {
          cxi2 = 0.0;
          sxi2 = 0.0;  }
        else  {
          cxi2 = FC(xi2);
          sxi2 = FS(xi2);  }

        cxi = cxi2 - cxi1;
        sxi = sxi2 - sxi1;
        ampx = sqrt( cxi*cxi + sxi*sxi );
        phsx = atan2( sxi, cxi );

/*      Find the field due to a rectangular aperture.  */
        amp1 = amp1 * ampy * ampx;
        phs1 = phs1 + phsy + phsx;
        *real = amp1 * cos(phs1);
        *imag = amp1 * sin(phs1);
        *amp = amp1;
        *phs = phs1;
        return;
}


/*****************************************************************************
* FUNCTION: EvalIntegralyP1
* DESCRIPTION: Evaluates the integral over Region 1 of the total
        integration range. Region 1 is characterized with many samples.
        Interpolating polynomials are found to fit the amplitude and phase
        variations, separately, for each sub-interval, or segment, and
        the integrations are performed over these polynomials.
        The polynomials used to fit the amplitude are always second degree,
        but the phase function may be taken as first or second degree
        depending on the value of the computed quadratic coefficient.
        Integrations over each segment are summed to find the total field.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        y0: height of a sample point in the aperture, relative to the origin
```

```
        rpo: distance from the origin to P
        lam: wavelength
        N1: number of Region 1 samples
        D: sampling intervals
        APDAT: field samples in aperture, amp and phs
* OUTPUTS:
        *realy,*imagy: the complex result of the integration
* CALLED BY: EvalIntegraly
* FUNCTION CALLS:
        coeff: computes the coefficients of interpolating polynomials
        SegmentCompute: computes the integral over a segment
* REFERENCE: Stamnes et al., "Evaluation of diffraction integrals using
        local phase and amplitude approximations," Optica Acta, Vol.30,
        No.2, pg.207-222, 1983.
*************************************************************************/
void EvalIntegralyP1(double y0, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *realy, double *imagy)
{
        double REAL,IMAG;       /* running sums */
        double delta;           /* sample spacing */
        double amptmp;          /* intermediate amplitudes for debugging */
        double Anm1,An,Anp1,Pnm1,Pn,Pnp1; /* amplitude and phase samples
                                used to characterize variation over a segment*/
        double t1,t2,u1,u2;     /* changes of integration variable */
        double a,b,c;           /* constant,linear,and quadratic coefficients
                                of amplitude function after changes of variable*/
        double psi;             /* phase factor outside the integral */
        double gambet;          /* multi-use variable, represents beta in
                                the linear phase assumption, and gamma
                                for quadratic phase */
        double y,yd1;           /* intermediate y values */
        double amp,phs;         /* result of integration over a segment */
        int QUADPHS;            /* signals if quadratic phase is being used */
        int eps;                /* accounts for negative phase curvature */
        int n;                  /* loop counter */
        int N0,N;               /* first and last sample numbers */

        N0 = 0;
        N = N1;
        delta = D[0];
        REAL = 0.0;
        IMAG = 0.0;
        if(N < 2) {
                printf("Error: Too few intervals\n");
                *realy = 0.0;
                *imagy = 0.0;
                return; }
/*  Sample values used for first 2 segments */
        Anm1 = APDAT[0][N0];
        An = APDAT[0][N0+1];
        Anp1 = APDAT[0][N0+2];
        Pnm1 = APDAT[1][N0] + pi*y0*y0/(lam*rpo);
        y = y0 + delta;
        Pn = APDAT[1][N0+1] + pi*y*y/(lam*rpo);
        yd1 = y;
        y = y0 + 2*delta;
        Pnp1 = APDAT[1][N0+2] + pi*y*y/(lam*rpo);
        if(Printfld) {
            fprintf(filefld," New, y0 = %.3lf\n",y0);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf \n",
                        y0,Anm1,APDAT[1][N0],Pnm1);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf\n",
                        yd1,An,APDAT[1][N0+1],Pn);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf\n",
                        y,Anp1,APDAT[1][N0+2],Pnp1);
```

```
        }

/*    Zeroth interval (n=0) */
      t1 = -1.0;
      t2 = -0.5;
      coeff(1,Anm1,An,Anp1,Pnm1,Pn,Pnp1,t1,t2,
         &psi, &eps, &gambet, &a, &b, &c, &u1, &u2, &QUADPHS);

      SegmentCompute(QUADPHS, u1, u2, eps, psi, gambet,
            a, b, c, &amp, &phs);

      REAL = REAL + delta * amp*cos(phs);
      IMAG = IMAG + delta * amp*sin(phs);

      if(Printint) {
          amptmp = sqrt(REAL*REAL + IMAG*IMAG);
          fprintf(fileint," New, y0 = %.3lf\n",y0);
          fprintf(fileint," %d  %.3lf  %d  %.6lf  %.3lf  %.6lf\n",
              0,delta,QUADPHS,amp,phs,amptmp);
          }


/*    First interval (n=1) */
      t1 = -0.5;
      t2 = 0.5;

      coeff(1,Anm1,An,Anp1,Pnm1,Pn,Pnp1,t1,t2,
         &psi, &eps, &gambet, &a, &b, &c, &u1, &u2, &QUADPHS);

      SegmentCompute(QUADPHS, u1, u2, eps, psi, gambet,
            a, b, c, &amp, &phs);

      REAL = REAL + delta * amp*cos(phs);
      IMAG = IMAG + delta * amp*sin(phs);

      if(Printint)  {
         amptmp = sqrt(REAL*REAL + IMAG*IMAG);
         fprintf(fileint," %d  %.3lf  %d  %.6lf  %.3lf  %.6lf\n",
              1,delta,QUADPHS,amp,phs,amptmp);
         }


/*    Intervals 2 through N-1  */
      for(n=2;n<=(N-1);n++)  {
        y = y0 + (n+1)*delta;
        Anm1 = An;
        Pnm1 = Pn;
        An = Anp1;
        Pn = Pnp1;
        Anp1 = APDAT[0][N0+n+1];
        Pnp1 = APDAT[1][N0+n+1] + pi*y*y/(lam*rpo);
        if(Printfld)
           fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf\n",
              y,Anp1,APDAT[1][N0+n+1],Pnp1);

        coeff(1,Anm1,An,Anp1,Pnm1,Pn,Pnp1,t1,t2,
           &psi, &eps, &gambet, &a, &b, &c, &u1, &u2, &QUADPHS);

        SegmentCompute(QUADPHS, u1, u2, eps, psi, gambet,
              a, b, c, &amp, &phs);

        REAL = REAL + delta * amp*cos(phs);
        IMAG = IMAG + delta * amp*sin(phs);

        if(Printint) {
```

```
            amptmp = sqrt(REAL*REAL + IMAG*IMAG);
            fprintf(fileint," %d  %.3lf  %d  %.6lf  %.3lf  %.6lf\n",
                n,delta,QUADPHS,amp,phs,amptmp);
            }


        }

/*    Last interval  */
        t1 = 0.5;
        t2 = 1.0;
        coeff(1,Anm1,An,Anp1,Pnm1,Pn,Pnp1,t1,t2,
           &psi, &eps, &gambet, &a, &b, &c, &u1, &u2, &QUADPHS);
        SegmentCompute(QUADPHS, u1, u2, eps, psi, gambet,
                       a, b, c, &amp, &phs);

        REAL = REAL + delta * amp*cos(phs);
        IMAG = IMAG + delta * amp*sin(phs);

        if(Printint) {
            amptmp = sqrt(REAL*REAL + IMAG*IMAG);
            fprintf(fileint," %d  %.3lf  %d  %.6lf  %.3lf  %.6lf\n",
                n,delta,QUADPHS,amp,phs,amptmp);
            }

        *realy = REAL;
        *imagy = IMAG;

        return;
}


/*************************************************************************
* FUNCTION: EvalIntegralyP2
* DESCRIPTION: Approximates the integral to infinity. Polynomial
        interpolating functions are derived to fit the three samples
        characterizing Region 2, both amplitude and phase functions are
        quadratic in this case. The functions are then extrapolated to
        infinity and integrated, using a single integration segment.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        y0: height of a sample point in the aperture, relative to the origin
        rpo: distance from the origin to P
        lam: wavelength
        N1: number of Region 1 samples
        D: sampling intervals
        APDAT: field samples in aperture, amp and phs
* OUTPUTS:
        *realy,*imagy: the complex result of the integration
* CALLED BY: EvalIntegraly
* FUNCTION CALLS:
        coeff: computes the coefficients of interpolating polynomials
        SegmentCompute: computes the integral over a segment
* REFERENCES:
        1) Stamnes et al., "Evaluation of diffraction integrals using
        local phase and amplitude approximations," Optica Acta, Vol.30,
        No.2, pg.207-222, 1983.
        2) MS thesis, Tom Russell.
        3) Several papers by J.H. Whitteker, cited in the above.
*************************************************************************/
void EvalIntegralyP2(double y0, double rpo, double lam, int N1,
        double D[2], DA APDAT, double *realy, double *imagy)
{
        double REAL,IMAG;       /* running sums */
```

```
        double delta;           /* sample spacing */
        double amptmp;          /* intermediate amplitudes for debugging */
        double Anm1,An,Anp1,Pnm1,Pn,Pnp1; /* amplitude and phase samples
                                used to characterize variation over a segment*/
        double t1,t2,u1,u2;     /* changes of integration variable */
        double a,b,c;           /* constant,linear,and quadratic coefficients
                                of amplitude function after changes of variable*/
        double psi;             /* phase factor outside the integral */
        double gambet;          /* multi-use variable, represents beta in
                                the linear phase assumption, and gamma
                                for quadratic phase */
        double y,yd1;           /* intermediate y values */
        double amp,phs;         /* result of integration over a segment */
        int QUADPHS;            /* signals if quadratic phase is being used */
        int eps;                /* accounts for negative phase curvature */
        int N0;                 /* first and last sample numbers */

        N0 = N1;
        delta = D[1];

        REAL = 0.0;
        IMAG = 0.0;

        Anm1 = APDAT[0][N0];
        An = APDAT[0][N0+1];
        Anp1 = APDAT[0][N0+2];
        Pnm1 = APDAT[1][N0] + pi*y0*y0/(lam*rpo);
        y = y0 + delta;
        Pn = APDAT[1][N0+1] + pi*y*y/(lam*rpo);
        yd1 = y;
        y = y0 + 2*delta;
        Pnp1 = APDAT[1][N0+2] + pi*y*y/(lam*rpo);
        if(Printfld) {
            fprintf(filefld," New, y0 = %.3lf\n",y0);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf \n",
                        y0,Anm1,APDAT[1][N0],Pnm1);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf\n",
                        yd1,An,APDAT[1][N0+1],Pn);
            fprintf(filefld," %.3lf  %.6lf  %.3lf  %.3lf\n",
                        y,Anp1,APDAT[1][N0+2],Pnp1);
            }

        t1 = -1.0;
        t2 = 1.0;
        coeff(2,Anm1,An,Anp1,Pnm1,Pn,Pnp1,t1,t2,
           &psi, &eps, &gambet, &a, &b, &c, &u1, &u2, &QUADPHS);
        u2 = -1000.0;
        SegmentCompute(QUADPHS, u1, u2, eps, psi, gambet,
                a, b, c, &amp, &phs);

        REAL = delta * amp*cos(phs);
        IMAG = delta * amp*sin(phs);

        if(Printint) {
            amptmp = sqrt(REAL*REAL + IMAG*IMAG);
            fprintf(fileint," %d  %.3lf  %d  %.6lf  %.3lf  %.6lf\n",
                0,delta,QUADPHS,amp,phs,amptmp);
            }

        *realy = REAL;
        *imagy = IMAG;

        return;
}
```

```
/*************************************************************************
* FUNCTION: SegmentCompute
* DESCRIPTION: Computes the integral over the interpolating functions
        on a single segment (either finite or infinite).
        The integral is separated into the sum of three integrals which
        are evaluated individually.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        QUADPHS: signals the quadratic phase assumption
        u1,u2: limits of integration
        eps: sign factor, accounts for negative phase curvature
        phi0: phase factor outside the integral
        gambet: amplitude factor outside the integral (gamma or beta)
        a,b,c: amplitude polynomial coefficients
* OUTPUTS:
        *amp,*phs: results of segment integration
* CALLED BY:
* FUNCTION CALLS:
        intQa,intQb,intQc: integral evaluations, quadratic phase case
        intLa,intLb,intLc: integral evaluations, linear phase case
* REFERENCE: Stamnes et al., "Evaluation of diffraction integrals using
        local phase and amplitude approximations," Optica Acta, Vol.30,
        No.2, pg.207-222, 1983.
*************************************************************************/
void SegmentCompute(int QUADPHS, double u1, double u2, int eps,
        double phi0, double gambet, double a, double b, double c,
        double *amp, double *phs)
{
        double reala,realb,realc,imaga,imagb,imagc; /* intermediate results*/
        double real,imag;       /* total complex field on segment */

/*      printf("%.3f %.3f %d %.3f %d\n",u1,u2,eps,gambet,QUADPHS); */

        if (QUADPHS) {
            intQa(u1,u2,eps,a,&reala,&imaga);
            intQb(u1,u2,eps,b,&realb,&imagb);
            intQc(u1,u2,eps,c,&realc,&imagc);}
          else {
            intLa(u1,u2,a,&reala,&imaga);
            intLb(u1,u2,b,&realb,&imagb);
            intLc(u1,u2,c,&realc,&imagc); }

        real = reala + realb + realc;
        imag = imaga + imagb + imagc;

        *amp = sqrt( real*real + imag*imag );
        *phs = atan2( imag,real );

        if( QUADPHS ) {
            *amp = *amp * sqrt(1.0 / fabs(gambet));
            *phs = *phs + phi0;
            }
          else  {
            *amp = *amp / gambet;
            *phs = *phs + phi0;
            }
        return;

}


/*************************************************************************
* FUNCTION: intQa
```

```
* DESCRIPTION: Evaluates the integral involving only a constant
        amplitude term; quadratic phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        eps: sign factor
        a: constant amplitude factor
* OUTPUTS: *reala,*imaga: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS:
        FC: Evaluates Fresnel cosine integral
        FS: Evaluates Fresnel sine integral
*****************************************************************************/
void intQa(double u1, double u2, int eps,
        double a, double *reala, double *imaga)
{
        double limfac,multfac;  /* factors required in a change of variable,
                                   to fit an alternative definition
                                   of the Fresnel Integral  */
        limfac = sqrt(2.0/pi);
        multfac = 1.0/limfac;
        if(u2 == -1000.0) {
            *reala = a*multfac*( 0.5 - FC(limfac*u1) ) ;
            *imaga = eps*a*multfac*( 0.5 - FS(limfac*u1) ) ; }
          else  {
            *reala = a*multfac*( FC(limfac*u2) - FC(limfac*u1) ) ;
            *imaga = eps*a*multfac*( FS(limfac*u2) - FS(limfac*u1) ) ; }
        return;
}


/*****************************************************************************
* FUNCTION: intQb
* DESCRIPTION: Evaluates the integral involving only a linear
        amplitude term; quadratic phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        eps: sign factor
        b: linear amplitude factor
* OUTPUTS: *realb,*imagb: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS: none
*****************************************************************************/
void intQb(double u1, double u2, int eps,
        double b, double *realb, double *imagb)
{
        if(u2 == -1000.0) {
            *realb = 0.5*b*( - sin(u1*u1) );
            *imagb = 0.5*b*eps*( cos(u1*u1) ); }
          else  {
            *realb = 0.5*b*( sin(u2*u2) - sin(u1*u1) );
            *imagb = 0.5*b*eps*( cos(u1*u1) - cos(u2*u2) ); }
        return;
}


/*****************************************************************************
* FUNCTION: intQc
* DESCRIPTION: Evaluates the integral involving only a quadratic
        amplitude term; quadratic phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        eps: sign factor
```

```
        c: quadratic amplitude factor
* OUTPUTS: *realc,*imagc: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS:
        FC: Evaluates Fresnel cosine integral
        FS: Evaluates Fresnel sine integral
*****************************************************************************/
void intQc(double u1, double u2, int eps,
        double c, double *realc, double *imagc)
{
        double lf,mf;           /* factors required in a change of variable,
                                   to fit an alternative definition
                                   of the Fresnel Integral  */
        lf = sqrt(2.0/pi);
        mf = 1.0/lf;
        if(u2 == -1000.0) {
            *realc = 0.5*c*( - u1*sin(u1*u1) +
                mf*( FS(lf*u1) - 0.5 ) );
            *imagc = 0.5*c*eps*( u1*cos(u1*u1)  +
                mf*( 0.5 - FC(lf*u1) ) );  }
          else  {
            *realc = 0.5*c*( u2*sin(u2*u2) - u1*sin(u1*u1) +
                mf*( FS(lf*u1) - FS(lf*u2) ) );
            *imagc = 0.5*c*eps*( u1*cos(u1*u1) - u2*cos(u2*u2) +
                mf*( FC(lf*u2) - FC(lf*u1) ) );  }
        return;
}


/*****************************************************************************
* FUNCTION: intLa
* DESCRIPTION: Evaluates the integral involving only a constant
        amplitude term; linear phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        a: constant amplitude factor
* OUTPUTS: *reala,*imaga: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS: none
*****************************************************************************/
void intLa(double u1, double u2, double a,
                double *reala, double *imaga)
{
        *reala = a*( sin(u2) - sin(u1) );
        *imaga = a*( cos(u1) - cos(u2) );
        return;
}


/*****************************************************************************
* FUNCTION: intLb
* DESCRIPTION: Evaluates the integral involving only a linear
        amplitude term; linear phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        b: linear amplitude factor
* OUTPUTS: *realb,*imagb: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS: none
*****************************************************************************/
void intLb(double u1, double u2, double b,
                double *realb, double *imagb)
{
```

```
        *realb = b*( cos(u2) - cos(u1) + u2*sin(u2) - u1*sin(u1) );
        *imagb = b*( sin(u2) - sin(u1) + u1*cos(u1) - u2*cos(u2) );
        return;
}


/*************************************************************************
* FUNCTION: intLc
* DESCRIPTION: Evaluates the integral involving only a quadratic
        amplitude term; linear phase case.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        u1,u2: limits of integration
        c: quadratic amplitude factor
* OUTPUTS: *realc,*imagc: complex results of integration
* CALLED BY: SegmentCompute
* FUNCTION CALLS:
        intLb: evaluates an integral involving a linear amplitude factor
*************************************************************************/
void intLc(double u1, double u2, double c,
                double *realc, double *imagc)
{
        double realb, imagb;    /* intermediate results */

        intLb(u1, u2, 2.0, &realb, &imagb);
        *realc = c*( u2*u2*sin(u2) - u1*u1*sin(u1) - imagb );
        *imagc = c*( u1*u1*cos(u1) - u2*u2*cos(u2) + realb );
        return;
}



/*************************************************************************
* FUNCTION: coeff
* DESCRIPTION: Determines the coefficients and limits on the new variable of
        integration based on the amp and phase samples on a particular
        segment of the integration range
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        REGION: Region 1 (lin or quad phs) or 2 (always quadratic phase)
        Anm1,An,Anp1: samples used to determine amplitude function
        Pnm1,Pn,Pnp1: samples used to determine phase function
        t1,t2: limits on integration variable t
* OUTPUTS:
        *psi: phase factor taken outside of integral
        *eps: sign factor in quadratic phase case
        *gambet: amplitude factor taken outside of integral,
                takes on gamma in quad phs case, beta in lin phs case
        *a,*b,*c: coeff's of amplitude function after change of variables
        *u1,*u2: limits on integration variable u
        *QUADPHS: true if quadratic phase is to be used
* CALLED BY: EvalIntegralyP1, EvalIntegralyP2
* FUNCTION CALLS:
        IntermCoeff: computes polynomial interpolating coefficients
        CoeffQuadPhs: performs change of variables, quadratic phase case
        CoeffLinPhs: performs change of variables, linear phase case
*************************************************************************/
void coeff(int REGION, double Anm1, double An, double Anp1,
           double Pnm1, double Pn, double Pnp1, double t1,
           double t2, double *psi, int *eps, double *gambet,
           double *a, double *b, double *c, double *u1,
           double *u2, int *QUADPHS)
{
        double A,B,C;           /* coeff's of amplitude function */
        double alpha,beta,gamma;/* coeff's of phase function */
```

```
        double ap,bp,cp;        /* amp coeff's after change of variable */
        double u1p,u2p;         /* limits on integration variable u */

        IntermCoeff(Anm1,An,Anp1,&A,&B,&C);
        IntermCoeff(Pnm1,Pn,Pnp1,&alpha,&beta,&gamma);

        if( (REGION==2 && gamma == 0.0) )
                printf("ERROR: gamma=0 on Region 2; Phase taken Linear\n");

        if( (REGION==2 && gamma != 0.0) ||
            (fabs(gamma) > 0.6) ) {          /* Forced to Quad (infinity) */

            *QUADPHS = 1;
/*          printf("gamma = %.3lf\n",gamma); */
            if( gamma < 0.0 )
                  printf("POSSIBLE ERROR: gamma = %.3lf\n",gamma);
            CoeffQuadPhs(A,B,C,beta,gamma,t1,t2,
                                &ap,&bp,&cp,&u1p,&u2p);
            *psi = alpha - beta*beta/(4.0*gamma);
            *eps = sign(gamma);
            *gambet = gamma;
            }

        else {
            *QUADPHS = 0;
            CoeffLinPhs(A,B,C,beta,t1,t2,
                        &ap,&bp,&cp,&u1p,&u2p);
            *psi = alpha;
            *eps = 0;
            *gambet = beta;
            }

        *a = ap;
        *b = bp;
        *c = cp;
        *u1 = u1p;
        *u2 = u2p;
        return;
}


/*************************************************************************
* FUNCTION: IntermCoeff
* DESCRIPTION: Determines the coefficients of the three terms of a
        second degree interpolating polynomial
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        fnm1,fn,fnp1: function samples
* OUTPUTS:
        *a,*b,*c: coefficients of the terms of degree 0, 1, and 2,
                respectively, of an interpolating parabola
* CALLED BY: coeff
* FUNCTION CALLS: none
*************************************************************************/
void IntermCoeff(double fnm1, double fn, double fnp1,
                double *a, double *b, double *c)
{
        *a = fn;
        *b = (fnp1 - fnm1) / 2.0;
        *c = (fnm1 + fnp1)/2.0 - fn;
        return;
}
```

```
/*************************************************************************
* FUNCTION: CoeffQuadPhs
* DESCRIPTION: Performs a change of variables to leave only a
        quadratic phase term inside the integral.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        A,B,C: coeff's of degree 0,1,2 of amplitude interpolation fcn
        beta,gamma: coeff's of degree 1,2 of phase interpolation fcn
        t1,t2: limits on integration variable t
* OUTPUTS:
        *a,*b,*c: coeff's of degree 0,1,2 of amplitude function
        *u1,*u2: limits on integration variable u
* CALLED BY: coeff
* FUNCTION CALLS: none
*************************************************************************/
void CoeffQuadPhs(double A, double B, double C,
        double beta, double gamma, double t1, double t2,
        double *a, double *b, double *c, double *u1, double *u2)
{
        double sfg;
        sfg = sqrt(fabs(gamma));
        *a = A - B*beta/(2.0*gamma) + C*beta*beta/(4.0*gamma*gamma);
        *b = B/sfg - C*beta/(gamma*sfg);
        *c = C/fabs(gamma);
        *u1 = sfg*(t1 + beta/(2.0*gamma));
        *u2 = sfg*(t2 + beta/(2.0*gamma));
        return;
}


/*************************************************************************
* FUNCTION: CoeffLinPhs
* DESCRIPTION: Performs a change of variables to leave only a
        linear phase term inside the integral.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        A,B,C: coeff's of degree 0,1,2 of amplitude interpolation fcn
        beta,: coeff of degree 1 of phase interpolation fcn
        t1,t2: limits on integration variable t
* OUTPUTS:
        *a,*b,*c: coeff's of degree 0,1,2 of amplitude function
        *u1,*u2: limits on integration variable u
* CALLED BY: coeff
* FUNCTION CALLS: none
*************************************************************************/
void CoeffLinPhs(double A, double B, double C,
        double beta, double t1, double t2,
        double *a, double *b, double *c, double *u1, double *u2)
{
        *a = A;
        *b = B/beta;
        *c = C/(beta*beta);
        *u1 = beta*t1;
        *u2 = beta*t2;
        return;
}


/*************************************************************************
* FUNCTION: FC
* DESCRIPTION: Performs a Romberg adaptive numerical integration on
        the Fresnel cosine integral evaluated on (0,b), where b
        can be positive or negative.  If the absolute value of b
```

```
        is greater than 5.0, an asymptotic approximation is made instead.
        The integration algorithm subdivides the range into smaller
        segments on each pass, computing an approximation to the
        integral each time. When an error tolerance is satisfied on
        difference between successive approximations, the last value
        is returned. Two tolerances must be satisfied: tol1 on the difference
        between the current approx. and the one made two cycles prior;
        and tol2 on the difference between the current and the immediately
        preceding estimate. Also at least 4 cycles are required.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: b: the second limit of integration
* OUTPUTS: only the return value
* CALLED BY: EvalIntegralx, fieldcomp, intQa, intQc
* FUNCTION CALLS:
        FCasympt: asymptotic approximation to the integral for |b| > 5.0
        fcos: evaluates the function at a sample point
*************************************************************************/
double FC(double b)
{
        double a;       /* limit of integration (always zero here)*/
        double diff;    /* overall integration range */
        double h;       /* current segment size (delta x) */
        double m;       /* intermediate result */
        double sumc;    /* running sum of segment contributions */
        double toldc[50],tnewc[50]; /* successive approximations to the value*/
        double valc;    /* current approximation to the value */
        double errc;    /* relative difference between the
                                two most recent approx's */
        double tol2;    /* error tolerance on errc */
        double err;     /* relative difference between the current estimate
                                and the estimate two cycles prior */
        double tol1;    /* error tolerance on err */
        double ca,cb;   /* function evaluations at either extreme */
        double x;       /* variable of integration */
        double cy;      /* function evaluation */
        long int n;     /* current number of intervals */
        long int i;     /* loop counter */
        int k;          /* loop counter */
        int depth;      /* current cycle number */

/*  Initialize variables  */
        depth = 0;
        sumc = 0.0;
        valc = 0.1;
        err = 1.0;
        errc = 1.0;

/*  Set tolerance on integration  */
        tol1 = 0.01;
        tol2 = 0.002;

/*  Define limits of integration as zero and b  */
        a = 0.0;
        diff = b-a;

/*  If diff equals zero, return the value zero */
        if (diff == 0.0)
                return(0.0);

/*  Use asymptotic functions for large absolute value */
        if ( b > 5.0 )
                return( FCasympt(b) );
        if ( b < -5.0 )
                return( -FCasympt(-b) );
```

```
/*  Evaluate the functions at the limits  */
        ca = fcos(a);
        cb = fcos(b);

/*  Evaluate first approximation to the integral  */
        toldc[1] = (diff/2.0)*(cb + ca);

/*  Do until error < tolerance                    */
        while( ((fabs(err) > tol1) || (fabs(errc) > tol2)
        || (depth < 4)) && (depth < 25) ) {
                depth++;
                n = pow(2.0,(depth-1));

                for(i=1;i<=n;i++) {
                        h = diff/(double)n;
/*  Choose more points x, evaluate functions at x, and sum up  */
                        x = a + ( (double)i - 0.5)*h;
                        cy = fcos(x);
                        sumc = sumc + cy;
                }

                m = h * sumc;
                tnewc[1] = (toldc[1] + m)/2.0;
                sumc = 0.0;

                for(k=1;k<=depth;k++) {
/*  Evaluate next approximation to the integral   */
                        tnewc[k+1] = tnewc[k] + (tnewc[k]-toldc[k])/
                           pow(2.0,(2*k)-1);
                        toldc[k] = tnewc[k];
                }

/*  Assign new values to valc and evaluate error  */
                err = errc;
                toldc[depth+1] = tnewc[depth+1];
                errc = (toldc[depth+1] - valc)/valc;
                valc = toldc[depth+1];

        }

/*              printf("valc: %f errc: %fdepth: %d\n",valc,errc,depth); */
/*      printf("\n"); */
        return(valc);
}



/*************************************************************************
* FUNCTION: FS
* DESCRIPTION: Performs a Romberg adaptive numerical integration on
        the Fresnel sine integral evaluated on (0,b). See the description
        for function FC for further details.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: b: the second limit of integration
* OUTPUTS: only the return value
* CALLED BY: EvalIntegralx, fieldcomp, intQa, intQc
* FUNCTION CALLS:
        FSasympt: asymptotic approximation to the integral for |b| > 5.0
        fsin: evaluates the function at a sample point
*************************************************************************/
double FS(double b)
{
/*  This function performs a numerical integration of the Fresnel */
/*   sin integral over the interval zero to b, where b is input.  */
```

```
        double a;       /* limit of integration (always zero here)*/
        double diff;    /* overall integration range */
        double h;       /* current segment size (delta x) */
        double m;       /* intermediate result */
        double sums;    /* running sum of segment contributions */
        double tolds[50],tnews[50]; /* successive approximations to the value*/
        double vals;    /* current approximation to the value */
        double errs;    /* relative difference between the
                                two most recent approx's */
        double tol2;    /* error tolerance on errs */
        double err;     /* relative difference between the current estimate
                                and the estimate two cycles prior */
        double tol1;    /* error tolerance on err */
        double sa,sb;   /* function evaluations at either extreme */
        double x;       /* variable of integration */
        double sy;      /* function evaluation */
        long int n;     /* current number of intervals */
        long int i;     /* loop counter */
        int k;          /* loop counter */
        int depth;      /* current cycle number */

/*  Initialize variables  */
        depth = 0;
        sums = 0.0;
        vals = 0.1;
        err = 1.0;
        errs = 1.0;

/*  Set tolerance on integration  */
        tol1 = 0.01;
        tol2 = 0.002;

/*  Define limits of integration as zero and b  */
        a = 0.0;
        diff = b-a;

/*  If b equals zero, return the value zero */
        if (diff == 0.0)
                return(0.0);

/*  Use asymptotic functions for large absolute value */
        if ( b > 5.0 )
                return( FSasympt(b) );
        if ( b < -5.0 )
                return( -FSasympt(-b) );

/*  Evaluate the functions at the limits  */
        sa = fsin(a);
        sb = fsin(b);

/*  Evaluate first approximation to the integral  */
        tolds[1] = (diff/2.0)*(sb + sa);

/*  Do until error < tolerance              */
        while( ((fabs(err) > tol1) || (fabs(errs) > tol2)
        || (depth < 4)) && (depth < 25) ) {
                depth++;
                n = pow(2.0,(depth-1));

                for(i=1;i<=n;i++) {
                        h = diff/(double)n;
/*  Choose more points x, evaluate functions at x, and sum up  */
                        x = a + ( (double)i - 0.5)*h;
                        sy = fsin(x);
                        sums = sums + sy;
```

```
                }

                m = h * sums;
                tnews[1] = (tolds[1] + m)/2.0;
                sums = 0.0;

                for(k=1;k<=depth;k++) {
/*  Evaluate next approximation to the integral   */
                        tnews[k+1] = tnews[k] + (tnews[k]-tolds[k])/
                           pow(2.0,(2*k)-1);
                        tolds[k] = tnews[k];
                }

/*  Assign new values to vals and evaluate error  */
                err = errs;
                tolds[depth+1] = tnews[depth+1];
                errs = (tolds[depth+1] - vals)/vals;
                vals = tolds[depth+1];

        }

/*              printf("vals: %f errs: %fdepth: %d\n",vals,errs,depth); */
/*      printf("\n"); */
        return(vals);
}


/*****************************************************************************
* FUNCTION: fcos
* DESCRIPTION: evaluation of integrand of Fresnel cosine integral
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: x: sample point
* OUTPUTS: return value
* CALLED BY: FC
* FUNCTION CALLS: none
*****************************************************************************/
double fcos(double x)
{
        return(cos(pi*x*x/2.0));
        }


/*****************************************************************************
* FUNCTION: fsin
* DESCRIPTION: evaluation of integrand of Fresnel cosine integral
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: x: sample point
* OUTPUTS: return value
* CALLED BY: FS
* FUNCTION CALLS: none
*****************************************************************************/
double fsin(double x)
{
        return(sin(pi*x*x/2.0));
        }



/*****************************************************************************
* FUNCTION: FCasympt
* DESCRIPTION: Asymptotic approximation to Fresnel cosine integral,
        valid for large x
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: x: evaluation point
* OUTPUTS: return value
```

```
* CALLED BY: FC
* FUNCTION CALLS: none
* REFERENCE: M. Abramowitz and I.A. Segun, "Handbook of Mathematical
        Functions", New York, Dover, 1965.
******************************************************************************/
double FCasympt(double x)
{
        double t1,t2,t2a,t3,t3a;  /* intermediate results */
/*      printf("Asymptote used at %f\n",x); */
        t1 = 0.5;
        t2 = 0.3183099 - 0.0968/(x*x*x*x);
        t2a = sin( pi*x*x/2.0 ) / x;
        t3 = 0.10132 - 0.154/(x*x*x*x);
        t3a = cos( pi*x*x/2.0 ) / (x*x*x);
        return( t1 + t2*t2a - t3*t3a );
}


/******************************************************************************
* FUNCTION: FSasympt
* DESCRIPTION: Asymptotic approximation to Fresnel sine integral,
        valid for large x
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: x: evaluation point
* OUTPUTS: return value
* CALLED BY: FS
* FUNCTION CALLS: none
* REFERENCE: M. Abramowitz and I.A. Segun, "Handbook of Mathematical
        Functions", New York, Dover, 1965.
******************************************************************************/
double FSasympt(double x)
{
        double t1,t2,t2a,t3,t3a;
        t1 = 0.5;
        t2 = 0.3183099 - 0.0968/(x*x*x*x);
        t2a = cos( pi*x*x/2.0 ) / x;
        t3 = 0.10132 - 0.154/(x*x*x*x);
        t3a = sin( pi*x*x/2.0 ) / (x*x*x);
        return( t1 - t2*t2a - t3*t3a );
}


/******************************************************************************
* FUNCTION: sign
* DESCRIPTION: Returns sign of input, as either -1 or +1
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS: x: any double
* OUTPUTS: return value
* CALLED BY: coeff, many other fcn's
* FUNCTION CALLS: none
******************************************************************************/
int sign(double x)
{
        if(x >= 0)
           return(1);
         else
           return(-1);
}
```

```
/*************************************************************************
* FILE NAME: INPTDATS.C
* CONTENTS:
        FindRcvr
        inputdata1
* DESCRIPTION: This file provides locations of the transmitter,
        receiver, and buildings, and building heights. This is
        only a temporary method of data input.
*************************************************************************/
#include <mdsrch.h>
#include <mdtypes.h>
void FindRcvr(int path, double delp, double zro[Np], double xro[Np],
        double yro[Np], ZXUVY R);
void inputdata1(int TRANS, double *yt, double zro[Np],
        double xro[Np], double yro[Np], BAH ybldg, BAC zxcorn);


/*************************************************************************
* FUNCTION: FindRcvr
* DESCRIPTION: Changes any of the three coordinates of the
        mobile receiver as it travels down a path. Several
        different paths are provided.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        path: integers between 0 and (Np-1),
                where Np is the maximum number of paths, set in MDSRCH.H
        delp: distance along path from original receiver position
        zro,xro,yro: original position for each path
* OUTPUTS:
        R: new coordinates of receiver point, indexed by:
           0:z, 1:x, 2:u, 3:v, 4:y
* CALLED BY: main
* FUNCTION CALLS: none
*************************************************************************/
void FindRcvr(int path, double delp, double zro[Np], double xro[Np],
        double yro[Np], ZXUVY R)
{
        switch( path ) {
            case 0:
                R[0] = zro[path];
                R[1] = xro[path] + delp;
                R[4] = yro[path];
                break;
            case 1:
                R[0] = zro[path];
                R[1] = xro[path];
                R[4] = yro[path];
                break;
            case 2:
                R[0] = zro[path] + delp;
                R[1] = xro[path];
                R[4] = yro[path] - delp*(13.0/300.0);
                break;
            case 3:
                R[0] = zro[path];
                R[1] = xro[path] - delp;
                if( R[1] >= -160.0 )
                    R[4] = yro[path] - delp*(1.0/40.0);
                  else
                    R[4] = 2054.0 *FTtoMTRS;
                break;
            case 4:
                R[0] = zro[path];
                R[1] = xro[path];
                R[4] = yro[path];
```

```
                break;
            case 5:
                R[0] = zro[path];
                R[1] = xro[path];
                R[4] = yro[path];
                break;
            case 6:
                R[0] = zro[path];
                R[1] = xro[path] - delp;
                R[4] = yro[path];
                break;
            }
        return;
}


/*************************************************************************
* FUNCTION: inputdata1
* DESCRIPTION: Fills the arrays of (z,x) building coordinates
        and building heights, the first positions on various receiver
        paths, and the transmitter location. If the (z,x) coord's of
        the transmitter are not (0,0), the buildings and receiver
        are translated to make it so.
        These are the receiver paths and building arrangements used
        in example runs and measurement comparisons.
        When routines are written so input can be taken from other sources,
        the functions needed will be translation to set the (z,x)
        transmitter coord's at (0,0), and conversion to meters if not
        already in those units.
        Building corners must be specified sequentially, encircling
        the building in either direction.
* PROGRAMMER/DATE: Tom Russell, 10/27/91
* INPUTS:
        TRANS: index of transmitter location
* OUTPUTS:
        *yt: transmitter height
        zro,xro,yro: original receiver positions for several possible paths
        ybldg: heights of all buildings
        zxcorn: (z,x) coordinates of all buildings
                field 1:  0 - z coordinate, 1 - x coordinate
                field 2:  0-3 correspond to the four corners
                field 3:  building index, 0-(Nb-1), where Nb is the maximum
                        number of buildings, set in MDSRCH.H
* CALLED BY: main
* FUNCTION CALLS: none
*************************************************************************/
void inputdata1(int TRANS, double *yt,
        double zro[Np], double xro[Np], double yro[Np],
        BAH ybldg, BAC zxcorn)
{
        double zbldg[2][Nb], xbldg[2][Nb]; /* z,x coord's of two buildings
                                    that are aligned with the z and x axes */
        double T[2];                /* transmitter coordinates before coordinate
                                    system is translated to set to (0,0) */
        int i,j,k;                  /* loop counters */

        switch( TRANS ) {
            case 0:
                *yt = 30.0;
                T[0] = 0.0;
                T[1] = 0.0;
                break;
            case 1:
                *yt = (2046.3+15.0)*FTtoMTRS;
```

```
                T[0] = 0.0;
                T[1] = 50.0*FTtoMTRS;
                break;
            case 2:
                *yt = (2118.0+6.0)*FTtoMTRS;
                T[0] = 0.0;
                T[1] = 0.0;
                break;
            }

/* Path 0 */
        zro[0] = 250.0;
        xro[0] = -100.0;
        yro[0] = 0.0;
/* Path 1 */
        zro[1] = 250.0;
        xro[1] = 0.0;
        yro[1] = 2.0;
/* Path 2: Davidson */
        zro[2] = 0;
        xro[2] = 0;
        yro[2] = (2043.0 + 6.0)*FTtoMTRS;
/* Path 3: Patton */
        zro[3] = 260.0*FTtoMTRS;
        xro[3] = 60.0*FTtoMTRS;
        yro[3] = (2058.0+5.75)*FTtoMTRS;
/* Path 4 */
        zro[4] = 250.0;
        xro[4] = 0.0;
        yro[4] = 0.0;
/* Path 5 */
        zro[5] = 200.0;
        xro[5] = 0.0;
        yro[5] = 10.0;
/* Path 6 */
        zro[6] = 250.0;
        xro[6] = 0.0;
        yro[6] = 2.0;

        ybldg[0] = 10.0;
        zbldg[0][0] = 150.0;
        zbldg[1][0] = 200.0;
        xbldg[0][0] = -500;
        xbldg[1][0] = 500.0;

        ybldg[1] = 15.0;
        zbldg[0][1] = 100.0;
        zbldg[1][1] = 120.0;
        xbldg[0][1] = -500.0;
        xbldg[1][1] = 500.0;

        for(i=0;i<=1;i++) {
          zxcorn[0][0][i] = zbldg[0][i];
          zxcorn[0][1][i] = zbldg[1][i];
          zxcorn[0][2][i] = zbldg[1][i];
          zxcorn[0][3][i] = zbldg[0][i];

          zxcorn[1][0][i] = xbldg[0][i];
          zxcorn[1][1][i] = xbldg[0][i];
          zxcorn[1][2][i] = xbldg[1][i];
          zxcorn[1][3][i] = xbldg[1][i];
          }

/* Davidson */
        ybldg[2] = 2074;
```

```
        zxcorn[0][0][2] = 20;
        zxcorn[0][1][2] = 70;
        zxcorn[0][2][2] = 68;
        zxcorn[0][3][2] = 18;
        zxcorn[1][0][2] = 22;
        zxcorn[1][1][2] = 25;
        zxcorn[1][2][2] = 187;
        zxcorn[1][3][2] = 187;

        ybldg[3] = 2062;
        zxcorn[0][0][3] = 70;
        zxcorn[0][1][3] = 206;
        zxcorn[0][2][3] = 202;
        zxcorn[0][3][3] = 68;
        zxcorn[1][0][3] = 25;
        zxcorn[1][1][3] = 30;
        zxcorn[1][2][3] = 188;
        zxcorn[1][3][3] = 187;

        ybldg[4] = 2086;
        zxcorn[0][0][4] = 206;
        zxcorn[0][1][4] = 256;
        zxcorn[0][2][4] = 252;
        zxcorn[0][3][4] = 202;
        zxcorn[1][0][4] = 30;
        zxcorn[1][1][4] = 29;
        zxcorn[1][2][4] = 192;
        zxcorn[1][3][4] = 188;

/* Patton */
        ybldg[5] = 2105;
        zxcorn[0][0][5] = 160;
        zxcorn[0][1][5] = 225;
        zxcorn[0][2][5] = 225;
        zxcorn[0][3][5] = 160;
        zxcorn[1][0][5] = -245;
        zxcorn[1][1][5] = -245;
        zxcorn[1][2][5] = -40;
        zxcorn[1][3][5] = -40;

/* Buildings indexed 2 through 5 are specified in feet and must be
   converted to meters, both in heights and (z,x) coord's  */
        for(j=2;j<=5;j++) {
          ybldg[j] = ybldg[j] * FTtoMTRS;
          for(k=0;k<=1;k++)
            for(i=0;i<=3;i++)
              zxcorn[k][i][j] = zxcorn[k][i][j] * FTtoMTRS;
          }


        ybldg[6] = 12.0;
        zxcorn[0][0][6] = 170.0;
        zxcorn[0][1][6] = 240.0;
        zxcorn[0][2][6] = 240.0;
        zxcorn[0][3][6] = 170.0;
        zxcorn[1][0][6] = -30.0;
        zxcorn[1][1][6] = -30.0;
        zxcorn[1][2][6] = -2.0;
        zxcorn[1][3][6] = -2.0;

/* Translate all buildings so that chosen transmitter location is (0,0) */
        for(k=0;k<=1;k++)
            for(j=0;j<=(Nb-1);j++)
                for(i=0;i<=3;i++)
                    zxcorn[k][i][j] = zxcorn[k][i][j] - T[k];
```

```
/* Translate receiver coordinates according to chosen transmitter */
        for(i=0;i<=(Np-1);i++) {
          zro[i] = zro[i] - T[0];
          xro[i] = xro[i] - T[1];
        }

        return;
}
```

VITA

Thomas A. Russell was born in Camden, NJ, in 1963 and moved to Greenlawn, N.Y. in 1972. He graduated from Harborfields High School in 1981 and received the B.S.E.E degree from the University of Virginia, Charlottesville, in 1986. Since then he has been employed with Stanford Telecommunications, Inc. in Reston, VA, where he has worked on propagation modeling and link simulation for microwave satellite links. He began taking night classes towards his M.S.E.E. in August, 1987.

In August, 1990 he took an educational leave of absence from STel to complete his M.S.E.E full-time at Virginia Polytechnic Institute and State University, Blacksburg, where he has served as a graduate research assistant with the Mobile and Portable Radio Research Group.