INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

MUTLI-CRITERIA ASSESSMENT OF ECOLOGICAL PROCESS MODELS USING PARETO OPTIMIZATION

by

Joel Howard Reynolds

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by _____
Chairperson of Supervisory Committee

_____

_____

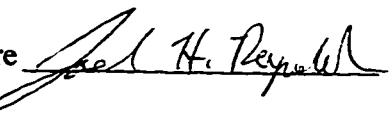
_____

Program Authorized
to Offer Degree Quantitative Ecology & Resource Management

Date 5 December 1996

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date 5 Dec 96

University of Washington

Abstract

MUTLI-CRITERIA ASSESSMENT OF ECOLOGICAL PROCESS MODELS USING PARETO OPTIMIZATION

by Joel Howard Reynolds

Chairperson of the Supervisory Committee: Professor E. David Ford
Interdisciplinary Program in Quantitative Ecology and Resource Management

Assessment is the essential step in using an ecological process model as a heuristic for investigating hypotheses. Assessment investigates the model's capacity to adequately simulate the phenomenon, as represented by selected criteria. A model's inability to satisfy all assessment criteria simultaneously reveals inadequacies in either the assessment decisions -- the criteria formulations or parameter space search, or the model structure - the mathematical representations or the model's collection of underlying hypotheses. An assessment procedure must be able not only to detect each type of deficiency but to distinguish between them, guiding model revision by locating each deficiency's source. Multiple criteria increase an assessment's capacity to do this. Currently there are no multiple criteria model assessment techniques designed both to detect and locate these different types of deficiencies.

The Pareto Optimal Model Assessment technique introduced here retains the multiple criteria as a vector rather than aggregating them into a single measure of performance. Optimizing the vector of criteria, generating the Pareto Optimal Set, may reveal that the model requires different parameterizations to satisfy different criteria, can not satisfy a particular criterion with any parameterization, or requires unrealistic parameterizations to satisfy all criteria simultaneously. Investigation of the Pareto Optimal Set reveals these

different deficiencies types, and their sources, whether in the assessment decisions or the two levels of model structure specification.

The Pareto Optimal Model Assessment technique is applied to the spatially explicit canopy competition model WHORL using ten criteria, binary interval error measures, and simulated evolution optimization to generate the Pareto Optimal Set. Assessment reveals a deficient mathematical representation of physiological plasticity. Assessment of the revised model reveals an inability to limit the growth rate of the tallest dominant trees to the observed range, as well as poor formulation of two criteria. Application of the Pareto Optimal Model Assessment technique to model development and model comparison is discussed. The revised model, WHORL2, is used to critique the representation of the canopy competition process in models of self-thinning of forest stands.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

LIST OF TABLES

PREFACE

The original motivation for this work arose from a discussion group on Ecological Process Models and Multiple Outputs which convened in the Spring of 1990 under the auspices of Drs. E. D. Ford and R. M. Cormack. The topic may well have engendered no further thought if it wasn't for the work and effort of Kristin Sorrensen-Cothern, under the guidance of E. D. Ford and D. Sprugel, in developing the canopy competition model WHORL. The ready availability of WHORL, it's clearly described purpose and development, and the initial investigations of Ms. Sorrensen-Cothern (detailed in Sorrensen-Cothern et al, 1993), provided a perfect problem-at-hand on which to develop, implement, and refine the Pareto Optimal Model Assessment Technique. The work presented here was possible precisely because of her earlier efforts.

As with experimental and observational methodologies, process modeling (mechanistic, scientific) provides a means of scientific inference. Model assessment, though in its infancy, is a problem of inference awaiting proper methodological development. Process models can be viewed as conditionally independent component hypotheses nested within the model's greater process structure. Assessment (inference) must then take account of both the component hypotheses as well as the greater connecting system: the process structure. This work is an attempt at advancing toward the goal of a formalized statistical methodology for this general problem of inference.

ACKNOWLEDGMENTS

I am indebted to a number of researchers, colleagues, and friends for their criticism, suggestions, support, and patience. The suggestions and criticisms of my supervisory committee, Drs. Peter Guttorp, Tom Hinckley, and Brian Mar, have improved both the research reported here and, more importantly, my critical thinking and exposition. Dr. Mark Kot brought Pareto Optimality to my attention, for which I am quite grateful. This work's greatest intellectual debt is owed to my chairman and advisor, Dr. E. David Ford, whose encouragement, belief, and guidance allowed me to pursue an initially rather ill-defined topic until it finally revealed itself.

The Ford Lab, Elizabeth Freeman, Owen Hamel, Marianne Turley, and especially Hiroaki Ishii, have been valuable, and generally willing, readers, reviewers, critics, and companions. I would still be writing C code were it not for the patience and clarity of Troy Frever. Rich Hinrichsen and Lorraine Read have been essential companions in the odyssey of my graduate career and life, a debt I cannot express let alone repay.

The interdisciplinary degree program in Quantitative Ecology and Resource Management was the perfect environment in which to develop this research, which falls outside the traditional departmental domains of both Ecology and Statistics. I am lucky to have been a part of this group.

I would like to thank my parents, Jerry and Carol Reynolds, siblings, PJ and Ric Green, Stacy and Jeff Sizemore, and TJ and Susan Reynolds, and my 'second' family, Mary Louise Viger, Peter, and Anne Muhich, for their encouragement, patience, and support throughout my student career, even when they weren't quite sure what I was doing. Most importantly, Jane Muhich has been a steadfast companion and friend throughout this

experience, even when the demands of graduate school conflicted with her own desires. I will always love her for that, and for keeping her word on a bet made 8 years ago.

Lastly, my deepest thanks go to Chelsea Bolan who has been an understanding, encouraging, eternally patient, and caring partner during these rather crazy last few years. If only I could promise the future was going to be simpler...guess we'll just have to see.

DEDICATION

This dissertation is dedicated to Chelsea Bolan, Jane M. Muhich, and Jerry Reynolds. Merry Christmas.

INTRODUCTION

Researchers are developing increasingly complex process models to investigate ecological theories and hypotheses. Mooney (1991) warns that models of whole plant and whole ecosystem operation cannot "be verified." Oreskes et al (1994) point out that no numerical model can ever be 'verified' or 'validated', it can only be confirmed relative to incomplete observations. This does not deny the use of models as heuristics for insight into ecological processes. Rather, it reinforces the importance of assessment for discovering the context within which the model can be usefully applied.

The methodology for assessment presented here requires a perceptual shift from one-time validation to a cyclic process. The Model Assessment Cycle iterates (i) assessment of a proposed model structure to detect its deficiencies, with (ii) revision of the model structure in light of the deficiencies (Beck 1985, 1987). The cycle repeats as each new model structure is developed, uniting model use and assessment into an evolving process.

To direct the model revision, the assessment procedure must not only detect the existence of deficiencies but locate their source in the decisions underlying the model structure. It must determine whether a deficiency is due to an inadequate parameterization, inadequate mathematical representation, inadequate selection of component hypotheses, or some combination of these. Utilizing a collection of assessment criteria increases the procedure's capacity to detect and locate these different types of deficiencies. A single criterion can be satisfied by different model structures; this problem of nonuniqueness underlies Oreskes et al's argument. Each additional assessment criterion increases the demands a model structure must satisfy to be deemed an acceptable simulation of the phenomenon. Assessing a model's ability to satisfy multiple criteria simultaneously is therefore a more stringent, and informative, critique than single or even sequential criteria assessment.

With multiple criteria assessment, no model or model parameterization is simply 'good' or 'bad' as it may simulate some aspects of the phenomena better than others. Assessing a model becomes a question of which aspects it simulates well and which it fails to capture. The trade-offs focus attention on model structure deficiencies, pinpointing either conflicts among the component hypotheses selected for inclusion in the model, i.e., the model's Process Structure, or improper equation forms, i.e., the model's Mathematical Structure.

Multiple criteria assessment methods are lacking in ecological modeling. Many ecological process models continue to be assessed with regard to a single criterion: for example, all of the canopy competition models in the review of Ford & Sorrensen (1992). The rigor gained from multiple criteria assessment has been acknowledged in discussions of Individual Based Models (Gross et al, 1992; Murdoch et al 1992; DeAngelis and Rose, 1992; Sorrensen-Cothern et al, 1993), though no techniques have been proposed to utilize the many possible outputs this type of model can produce (Gross et al, 1992). Subsequently, they are still viewed as resistant to assessment and investigation (Botsford, 1992; Clark, 1992a).

The multiple criteria calibrations that do appear in the ecological literature (e.g., Halfon, 1979; Gentil & Blake, 1981; Beck, 1987; Sievanen et al, 1988; Sorrensen-Cothern et al, 1993), and multiple criteria assessments regularly employed in General Circulation Models (Wigley & Santer, 1988; Cubasch & Cess, 1990; Gates et al, 1990), storm surge models (Dingman & Bedford, 1986), and water quality models (Beck & Halfon, 1991; van Straten & Keesman, 1991) use optimization techniques that are inappropriate for assessing ecological process models as they do not locate the sources of detected deficiencies. All of these techniques convert the multiple criteria optimization problem to univariate optimization, though their means of doing so differ, e.g., aggregating criteria into a scalar cost function via a weighted sum of error measures, trade-off methods, or sequential search methods. Through such conversion an overall assessment of model performance is made relative to a single parameter which is itself constructed from a

number of different measures. But it is exactly the model's simultaneous performance on these different measures that needs to be observed in order to locate the source of any deficiency. By preventing direct observation of the different performance measures, techniques of converting multiple outputs to a single variable restrict the researcher's ability to assess the theory underlying the model.

A heuristic for model assessment is presented which not only detects deficiencies but guides the researcher in locating their sources among the assessment decisions, the model's Process Structure, or its Mathematical Structure. Chapter 1 defines three levels of decisions required in developing a process model, each of which is a potential source of model deficiencies. Common model assessment techniques are shown to focus on only a subset of these decision levels. The Pareto Optimal Set is introduced as a source of assessment information on all three levels of decisions. Chapter 2 describes the Pareto Optimal Model Assessment Technique, the heuristic for model assessment which is the major product of this research. The technique is illustrated by the assessment of the canopy competition model WHORL (Sorrensen-Cothern et al, 1993). The chapter concludes with a discussion of other applications for the technique. Chapter 3 describes the simulated evolution optimization program developed for generating the Pareto Optimal Set used in WHORL's assessment. Chapter 4 employs the revised canopy competition model, WHORL2, to critique the representation of the competition process found in models proposed to explain self-thinning, the relationship between biomass or volume and stand density in crowded, even-aged monospecific stands of plants. Chapter 5 concludes this work with a discussion of investigating the Pareto Optimal Model Assessment Technique.

CHAPTER 1: ASSESSMENT OF ECOLOGICAL PROCESS MODELS

BOUNDING DECISIONS

A process model's form, (the variables, parameters, and relationships among the variables), is based on ecological theory or hypotheses (e.g. Clark and Holling, 1979). Such models are often constructed to investigate ecological or biological theories (Oderwald and Hans, 1990).

Model Assessment judges the adequacy and influence of the decisions made in constructing the model, the bounding decisions. These decisions determine which processes are included in the model and their mathematical representations (Overton 1977, Holling 1979), [e.g., whether a linear or nonlinear relation between variables should be used (see, for example, Model Description in Sorrensen-Cothern et al, 1993)].

A process model has three levels of specification (Fig. 1.1), each with its associated bounding decisions. Level 1, the Process Structure, defines which processes are to be included in simulating the phenomenon. This includes deciding on the system inputs and variables, which processes should be represented by constants or integrating functions and which explicitly modeled, and how an organism is considered to interact with the environment. These decisions are based on the hypotheses the researcher holds regarding the important processes driving the phenomenon being modeled. These decisions cannot be directly assessed without being translated into a quantitative form.

Level 2, the Mathematical Structure, defines the mathematical representations of the Level 1 processes. It is essential to recognize that in process models the Mathematical Structure is determined, to a large degree, by the Process Structure. The Level 2 decisions simply quantitatively specify the Level 1 decisions.

Taken together, the Level 1 and 2 decisions constitute the model structure, defining the component equations and their interactions. A model is often conceived of as simply these two levels of specification, a structure that only needs parameter values specified to generate a simulation. However, specifying the parameter values constitute a third level of bounding decisions, Level 3, for that very reason: they too must be defined in order to produce a simulation. They are defined in a context determined by the Level 2 and 1 decisions.

Model assessment judges the adequacy of these three levels of description. An inadequate simulation results from either an (i) inadequate parameterization (Level 3), (ii) inadequate Mathematical Structure (Level 2), (iii) inadequate Process Structure (Level 1), or a combination of these inadequacies. Each level of specification has a corresponding model assessment. By moving through the hierarchy of corresponding assessments, one is able to locate the source of model deficiencies as they are detected.

COMMON MODEL ASSESSMENT TECHNIQUES

Assessment techniques tend to focus only on levels 3 or 2 of the model's bounding decisions. In some cases, these techniques can reveal information about deficiencies in the model's Process Structure, but no techniques currently focus on this as an assessment goal. Parametric sensitivity analysis investigates the relative sensitivity of a chosen criterion to changes in specific parameters. If manipulating a parameter does not produce changes in the model's performance relative to the criteria chosen for assessment, the parameter is unidentifiable. Calibration can not determine optimal values for unidentifiable parameters. The presence of unidentifiable parameters may signify superfluous components in the model (Level 1), or inadequate mathematical representations (Level 2). However, as a parameter's identifiability is a function of the criteria used to assess the model, one needs to consider how much the parametric sensitivity analysis reflects the criteria selection as opposed to the model's bounding

decisions. New criteria may need to be selected in order to detect the influence of a particular parameter and its component process.

Components of the model's mathematical structure, the individual Level 2 decisions, can be directly assessed in a dynamical sensitivity analysis. This investigates the effect of selecting different mathematical representations on the model's ability to simulate the observations. E.g., using a linear versus nonlinear functional form, or discrete versus continuous representation of a variable. Such an analysis assumes the Process Structure is sufficient at the component level at least, and that the assessment criteria are well formulated.

The complete model structure, the combined decisions of Levels 1 and 2 rather than just particular components, is the focus of a structural assessment. This uses the results of calibration to detect inadequacies in a model structure and is similar to structural analysis in engineering (Beck and Halfon, 1991). If no parameterizations, or only unrealistic parameterizations, allow the model to simulate the observations acceptably, then there is a deficiency in the model structure. It is generally sufficient in engineering to assume all such deficiencies originate in the model's Mathematical Structure. For ecological process models, these deficiencies may arise from either an inadequate mathematical representation, a Level 2 decision, or to an inadequate Process Structure, a Level 1 decision. An inadequate Process Structure either is missing processes or has an inadequate process interaction representation. The Pareto Optimal Model Assessment Technique, (Chapter 2), extends the engineering concept of structural analysis by locating the deficiency sources among the model's Process Structure, it's Mathematical Structure, or the assessment decisions (Fig. 1.2).

PARETO OPTIMIZATION

Retaining a vector of criteria rather than aggregating them into a scalar error measure makes it easier to detect model deficiencies and locate their sources. Parameterizations can be compared using Pareto Optimality, a method of defining an optimum set with regard to a vector of incommensurable criteria (for further mathematical development, see Vincent and Grantham, 1981). Pareto Optimization was first developed in economics and is used mainly there and in engineering (e.g. Taylor et al 1975, Olenik and Haimes 1979; Vincent, 1987). Its general use is for developing other multi-criteria optimization techniques (see Steuer, 1986 or Yu, 1985), not as a tool for gaining insight into model structure deficiencies as proposed here.

A simple example will illustrate. Assume three criteria are used to judge a canopy competition model for trees: cumulative mortality in the stand, median live tree height of the stand, and crown angle of a tree grown in isolation from neighbors. Each parameterization of the model produces a three component vector, one component for each of the three outputs. Assume a binary interval error function is used for each criterion: a prediction falling within the interval [a, b] of acceptable values does an acceptable job with regard to this criterion and is labeled 'good'; otherwise it is labeled 'bad'. Each simulation produces an Assessment Vector:

(Mortality, Median Height, Crown Angle)

Assessment Vector for simulation 1 = (bad, good, bad)

Assessment Vector for simulation 2 = (bad, good, good)

Assessment Vector for simulation 3 = (good, bad, good)

Assessment Vector for simulation 4 = (good, bad, good)

As simulation 2 is at least as good as simulation 1 with regard to Mortality and Median height while actually doing better with regard to Crown Angle, simulation 2 dominates (equivalently, 'Pareto dominates') simulation 1: Sim. 2 $>_p$ Sim. 1. Formally, parameterization X dominates parameterization Y (X $>_p$ Y) if and only if X is at least as good as Y with respect to all criteria and there is a criterion for which X is strictly better than Y.

Simulations 3 and 4 produce distinct simulations but give identical Assessment Vectors. They neither dominate nor are dominated by simulations 2 or 1; any improvement in satisfying one criterion comes at a cost in satisfying another criterion. No judgment can be made regarding precedence among simulations 2 and (3, 4) without resorting to some weighting or preference among the assessment criteria (e.g. Mortality is more important than Median Height so simulations 3 and 4 both dominate simulation 2).

The Pareto Optimal Set is the set of undominated parameterizations; parameterizations {2, 3, 4} in this example. Everything that can be improved is discarded. The Pareto Set is partitioned into groups of parameterizations which generate the same Assessment Vector. The Pareto Set above has two groups: parameterization 2 and parameterizations (3, 4).

The Pareto Optimal Set, being the set of undominated parameterizations, need not necessarily contain a parameterization which achieves all the criteria. A number of results may occur:

(i) the model may require different parameterizations to satisfy different assessment criteria (as in the example),

(ii) the model may fail ever to satisfy a specific assessment criterion with the parameter values investigated, or

(iii) the model may only satisfy assessment criteria simultaneously with unrealistic parameterizations.

These three scenarios can arise from an insufficient parameter search, improper criteria formulation, or model structure deficiencies (see Chapter 2). The adequacy of the parameter search is investigated by analyzing which parameterizations occur in the Pareto Optimal Set. Analysis of these parameterizations, particularly the similarities and differences between those being classified into different groups of the Pareto Optimal Set, can reveal deficiencies in both the model's Mathematical Structure and in the criteria formulations. Any remaining deficiencies arise from the model's Process Structure. The scenarios above will guide the researcher in locating the underlying deficiencies (see Chapter 2).

Error Measures

The form of each assessment criterion's error measure dictates which simulations satisfy the criterion, influencing the parameter space search and hence the Pareto Optimal Set. Squared deviance error measures may too quickly restrict the parameter space search to regions which produce results very near the expected value of a particular criterion (Kursawe, 1991). In multiple criteria optimization, this premature focus on criterion-specific optimal parameterizations may prevent the search from locating parameterizations which achieve a better overall satisfaction of the collected criteria but which are suboptimal for any specific criterion -- exactly the parameterizations of interest in model assessment. Criterion-specific optimal parameterizations reveal little about the integrative adequacy of the model.

These error measures may also result in a greater variety of Assessment Vectors in the Pareto Optimal Set representing smaller trade-offs in criteria achievement. In contrast, binary interval error measures reveal the more fundamental conflicts among criteria achievement, masking the small scale trade-offs. They also provide direct insight into the parametric sensitivities of the model with regard to each criterion.

Binary error measurements arose in engineering (Hornberger and Spear, 1981; Hornberger and Cosby, 1985), and have been applied to parametric uncertainty analysis, including preliminary model calibration (Hornberger and Spear, 1981; Jaffe et al, 1987), sensitivity analysis with sparse data (Hornberger and Cosby, 1985; Mäkelä, 1988), and investigation of the influence of parametric uncertainty on model forecasts (Rose et al, 1991; van Straten and Keesman, 1991). Keesman and van Straten (1989) mention that using a binary error measurement to 'filter' the parameter space can provide insight into model structure uncertainty. Their focus, however, was on the prediction of the order of time-series models and no details or techniques for pinpointing structural deficiencies were given. All of these applications utilize one criterion.

Coupling vectors of criteria assessed with binary error measurements and Pareto Optimization gives a new technique for assessing ecological process models. This technique not only detects deficiencies, but can locate their source in either the model's Process Structure, its Mathematical Structure, or the assessment decisions. Analysis of the Pareto Optimal Set often can pinpoint the deficient model components (Chapter 2). This detailed level of information is unavailable from multi-criteria techniques which convert to scalar optimization.

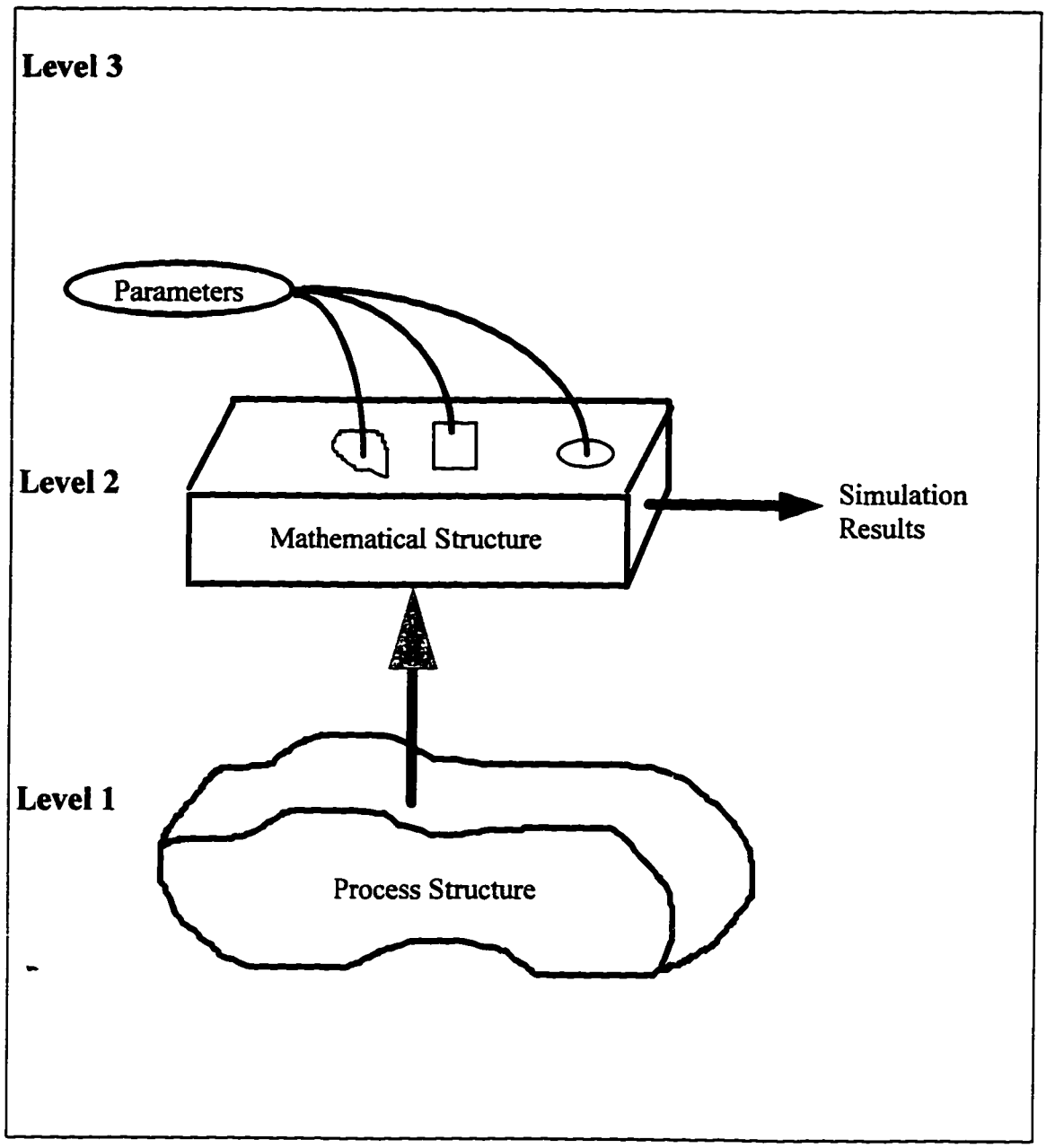


Figure 1.1 A Process Model's Three Levels of Structure.

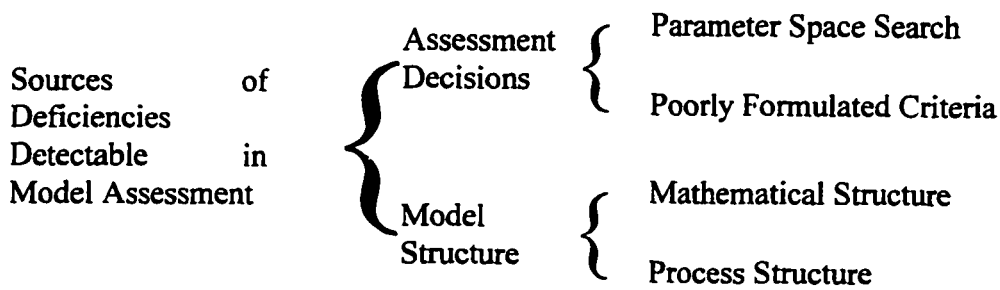


Figure 1.2 Potential Sources of Deficiencies detected during Process Model Assessment.

CHAPTER 2: THE PARETO OPTIMAL MODEL ASSESSMENT CYCLE

There are four sources of deficiencies in model performance: the parameterization, the model's Mathematical Structure, the criteria formulations, or the model's Process Structure. Each potential model deficiency source is the focus of a different stage in the Pareto Optimal Model Assessment cycle (Fig. 2.1). The activities of each stage are illustrated by application to the spatially explicit canopy competition model WHORL (Sorrensen-Cothern et al, 1993).

THE TREE CANOPY COMPETITION MODEL WHORL

WHORL was developed to investigate the dynamics of the spatial processes underlying competition: (i) resource distribution and acquisition and (ii) growth and resource allocation (Sorrensen-Cothern et al, 1993). WHORL was constructed to simulate competition between trees of *Abies amabalis*, a shade tolerant species, in clustered natural regeneration of a density of 9.08 trees ha^{-1} and growing up to 7 m tall. WHORL is based on a modular, three-dimensional representation of a tree (see Sorrensen-Cothern et al, 1993). The modular representation directly models the interaction between plant modules (branches and the foliage they support) and the local above-ground resource environment rather than just interactions between whole plants. Resource acquisition is modeled at the level of a cell of foliage (a cube 10 cm on a side). Plant response is modeled at the modular level of individual tree construction by growth of a whorl sector representing a branch, and at the whole plant level by main stem height increment. Modular autonomy among branches allows plasticity to be incorporated into both the resource acquisition and allocation processes through differences in foliage characteristics, e.g. radiation interception efficiency.

WHORL has two modes of simulation: growth of a stand of competing trees and growth of a single tree in a neighborless environment (open grown). Both modes were assessed, little faith being placed in a model that successfully simulates a stand of competing trees by generating obviously erroneous open grown crown structures.

STAGE 1: SELECTING PARAMETER RANGES, ASSESSMENT CRITERIA AND ERROR MEASURES, AND THE PARAMETER SPACE SEARCH TECHNIQUE

Search ranges for WHORL's parameters were selected based on prior calibrations (Sorrensen-Cothern et al, 1993) (Table 2.1). While smaller step sizes refine the parameter search and may improve the criteria achievement displayed in the Pareto Optimal Set, this must be balanced against the computational demands of the model simulation and search technique employed. The influence of too limited a parameter search is obvious -- too limited an exploration of the model's possible dynamics. The influence of the criteria and their respective error measures is often not as readily recognized.

Assessment Criteria Selection

Different criteria provide different perspectives on model performance and have different sensitivities to different model components. The quantitative formulation of an assessment criterion may generate deficiencies in the model assessment by masking deficiencies in the model structure. This occurs if the formulation biases the criterion in a way to make inadequate simulations appear adequate, or the converse.

As it is WHORL's initial assessment, general criteria were used such as stand height distribution at age X, cumulative mortality at age Y, etc. rather than highly specific characteristics such as change in yearly branch increment during the course of a tree's life. This precisely focused criterion is influenced in complex ways by the interactions of the modeled processes, and hence will not provide direct, easily interpretable insights into

model deficiencies. The ten criteria chosen for WHORL's structural assessment (Table 2.2) follow the guidelines of (i) assessing both modes of the model, open grown and crown competition, (ii) focusing on general characteristics of crown growth competition. The criteria form a hierarchy of refinement: six focus on general stand and open grown crown characteristics while the remaining four focus on the more specific growth rates of a stand of trees competing for light.

Binary interval error measures were constructed for each criterion using data from a permanent plot (Table 2.2) (see Sorrensen-Cothern et al 1993 for more details). Intervals, centered on the observed values from the data set when such information was available, were constructed from the data set in consultation with researchers familiar with the species (Hinckley and Brookes, pers. comm.).

Search Technique

A parameterization's Pareto Optimality (Chapter 1) depends on both individual criterion performance and the particular combination of criteria satisfied. It is not captured by the number of criteria satisfied nor any other aggregated measure of performance across the various criteria, and hence cannot be summarized in a simple univariate score function. This excludes the use of gradient-based optimization methods, both deterministic 'hill-climbing' techniques and stochastic methods such as simulated annealing (Uhry, 1989; Kan and Timmer, 1989), in generating the Pareto Optimal Set. Brute-force forward searches, Monte Carlo methods (assuming some prior information on the parameter distributions), or simulated evolution methods are all viable techniques for generating the Pareto Optimal Set. Of these three classes, only simulated evolution techniques provide a feedback mechanism for recursively refining the parameter space search (see Chapter 3).

Initially, a forward search of the parameter space was used to generate WHORL's Pareto Optimal Set. This involved 7560 simulations over the lattice defined by Table 2.1.

STAGE 2: GENERATING THE PARETO OPTIMAL SET

Generating the Pareto Optimal Set is the most computationally intensive stage of the Model Assessment cycle. The simulation demands place computational limits on the size of model to which this assessment technique can be applied. Simulation time, memory costs and the dimension of the parameter space being searched will affect the computational demands of a simulation.

The key insights of the Pareto Set come from optimizing simultaneous achievement of criteria. Reducing the optimization problem by disaggregating the criteria set or working on clusters of criteria sequentially undermines the very purpose of calculating the Pareto Set. Similarly, disaggregating the model structure and assessing the component models independently, while providing great information and insight regarding the component model, undermines the heuristic's focus on the complete model structure -- the interactions of the components defined by the Process Structure.

Automating the simulations required a complete rewriting of WHORL's computer code. The input routine was rewritten, allowing a single file to list multiple parameterizations. WHORL's code was amended to calculate and report the assessment criteria results of each simulation, rather than producing raw data files which had to then be read into a statistical package for analysis. In the process, the code was modularized, documented, and assessed for programming errors, which were corrected . The revised code (see Stage 4 below) appears in Appendix A.

Each parameterization produced a ten-component output vector, one component for each criteria. Each component is classified as good if the prediction falls within the associated criterion's error measure, otherwise it is classified as bad. Each vector of goods and bad, called the Assessment Vector, is compared to the other vectors until only the undominated vector remain, the Pareto Optimal Set (Table 2.3). The 25 undominated

vectors clustered into four groups (Table 2.3). For example, 9 different parameterizations each generated the Assessment Vector labeled 'Group 4' in Table 2.3.

STAGE 3: ASSESSING THE PARAMETER SEARCH RANGES

WHORL's model structure can satisfy every criterion, but not all ten simultaneously (Table 2.3). The deficiencies preventing complete simultaneous satisfaction of all criteria may originate in the model structure, the parameter space search, or the criteria formulations (Fig. 1.2). Too limited a parameter search may not reveal the full capacity or deficiencies of the model structure, producing a misleading Pareto Optimal Set.

The parameter search ranges are assessed by plotting the parameter settings in the Pareto Optimal Set against the available search ranges (Fig. 2.2). If all elements of the Pareto Optimal Set use a parameter setting at an extreme of its search range, then the range may have been too limited. If so, the search ranges must be extended and a new Pareto Optimal Set generated.

Every parameterization in the Pareto Optimal Set used the smallest Sector Increment settings investigated and either the median or smallest Height Increment settings (Table 2.4). The settings for the other parameters do not reveal any noticeable deficiencies in the search ranges. For example, almost the full range of investigated Dead and E settings appear in the Pareto Optimal Set (Table 2.4).

The Pareto Optimal Model Assessment was restarted (Fig. 2.1). The parameter search ranges for Sector and Height Increment were extended (Table 2.5).

An evolutionary simulation optimization routine was developed to search WHORL's parameter space and generate its Pareto Optimal Set (see Chapter 3). Simulated evolution is a suite of techniques to evolve solutions to optimization problems by simulating natural selection (Michalewicz, 1992; Fogel, 1994). These include Genetic Algorithms

(Goldberg, 1989; Holland, 1992), Evolutionary Strategies (Bäck et al, 1991), Evolutionary Programming (Fogel, 1994), and their extensions (Michalewicz et al 1992). The terms used are borrowed from genetics.

The evolutionary simulation optimization routine employed both single parameter mutations (within a given search range) and crossover recombination of two parameterizations. A parameterization's fitness was a function of both the number of criteria satisfied and membership in the Pareto Optimal Set. The Pareto Optimal Set was updated each generation, the routine running for 50 generations with an offspring population of 75 simulations each generation. The parent population, from which the offspring parameterizations were bred, was of fluctuating size, though always larger than 75; the parent population consisted of all parameterizations in the Pareto Set supplemented by dominated parameterizations from the last generation's offspring to provide genetic variety. The optimization routine's initial population consisted of the Pareto Set generated from the forward search (Table 2.3), augmented with randomly selected parameterizations for genetic diversity (see Chapter 3). The final Pareto Set (Table 2.6) was judged stable, having existed without change in the group Assessment Vectors for 13 generations.

While such an optimization technique is not required for generating the Pareto Optimal Set, it is more efficient than a forward search. Evolution programs require only that a procedure be defined for selecting the next generation's parents using a measure of achievement of the selection criteria. Multiple criteria can be successfully employed in this domain in a vector form, even with binary error functions, allowing membership in the Pareto Set to be used as the optimization criterion.

Again, the parameter search was investigated by plotting the Pareto Optimal settings against the available search ranges (Fig. 2.3). No parameters congregate at the extremes

of the search space, suggesting that WHORL's deficiencies arise from its model structure or the assessment criteria, not the optimization search.

STAGE 4: INVESTIGATING THE MATHEMATICAL STRUCTURE

As the parameter search ranges appear adequate, the focus moves to the model's Mathematical Structure (Fig. 2.1) as a possible source of the deficiencies revealed in the Pareto Optimal Set (Table 2.6). The Pareto Optimal Set is a filter capturing the most informative parameterizations. These 181 simulations can be investigated in more detail than the 11310 simulations in the search routine, with specific attention placed on whether criteria are satisfied in acceptable ways.

Model inadequacies traceable to deficiencies in the Mathematical Structure may originate in the decisions of the Process Structure which the mathematics is coding for. However, such deficiencies are generally due to the particular mathematical representation chosen for the hypothesis rather than the general hypothesis itself.

The Pareto Optimal simulation results clearly reveal (Fig. 2.4) unnaturally enhanced clustering of tree heights in groups 2, 3, 5, 7, 8, and somewhat less in groups 4 and 6. This clustering biases the regression which determines the dominant slope and dominant R^2 criteria (for example, Fig. 4 (d) or (h)). If the regressions are recalculated with these clusters removed from the fit, only parameterizations from Groups 2 and 3 achieve the dominant slope criterion, and only one parameterization from Group 3 satisfies the dominant R^2 criterion . Groups 4 - 8 (Table 2.6) achieve membership in the Pareto Optimal Set as a result of the regression bias produced by the clustering.

The clustering occurs at the height class boundaries. The height classes are dynamically set each year and determine the physiological parameter values a tree will have for the ensuing growth period (see footnote, Table 2.1). Clustering occurs under combination of the parameters Dead and E with extreme differences between their height class values.

Consider a simulation with Dead setting (10, 10, 40) and E setting (2.5, 2.0, 1.0). As the stand grows, a tree slightly taller than the moderate / tall height class boundary suffers extreme branch maintenance costs, i.e. Dead = 40, and will fail to grow at the same rate as its height class companions. It eventually enters the moderate height class as one of its tallest members. This entails a marked change in the tree's physiological characteristics - branch maintenance costs decrease by a factor of 4, 40 -> 10, while conversion efficiency of the foliage is doubled, 1 -> 2. Growing tremendously, the tree's height increment exceeds that of taller trees (Fig. 2.4, f). It is reclassified into the tall height class the following year -- albeit, again, one of the shortest. The cycle repeats, attracting more and more trees to heights around the class boundaries for which the parameters Dead or E experience extreme changes in value. Note that clustering did not occur in the Group 1 simulations (Table 2.6) as there is at most only slight physiological plasticity and so little difference between height classes in parameters E and Dead (Fig. 2.3). This inadequate representation of physiological plasticity must be revised and the Pareto Optimal Assessment cycle re-initiated (Fig. 2.1).

This assessment reveals that representing plasticity on a tree-to-tree scale is too restrictive, producing extreme changes in physiological characteristics year to year, even if the tree's local light environment does not change (as the height of the tallest trees determine the height class boundaries). To overcome this, the physiological parameters representing foliage properties, (E, D, K), were made a property of the smallest level of spatial resolution -- the cube (10 cm on a side) of varying foliage density, rather than of the whole tree. Foliage properties could now differ within and across branches in a tree. Each foliage parameter takes one of two settings, a shade setting or a sun setting, with a third parameter determining the light intensity where the setting switches. The branch maintenance cost parameter, Dead, was made to vary as a simple linear function of branch length with a minimum cost threshold (Fig. 2.5). The Sector and Height Increment parameters remain set at the level of the whole tree and remain fixed for the stand. These

changes in physiological plasticity representation were incorporated to form a new model, WHORL2.

The Pareto Optimal Model Assessment was applied to WHORL2 (Fig. 2.1). WHORL2's Pareto Optimal Set was generated using the simulated evolution optimization routine (Table 2.7). The Pareto Optimal parameter search ranges appear sufficient (Fig. 2.6). The model structure revision has resulted in a shift to slightly larger Sector Increment settings to compensate for the increased maintenance burden of the new branch maintenance cost mechanism (Fig.s 2.3 and 2.6). Nor do the optimal simulations reveal any Mathematical Structure deficiencies (Fig. 2.7).

The inability to satisfy all criteria simultaneously (Table 2.7) must arise from deficiencies in either the model structure or the criteria formulations. WHORL2 improves WHORL's performance, represented by Groups 1 - 3 in Table 2.6 once the effect of clustering is removed, as there are groups which achieve both the more general stand criteria (Median Live Height, Mortality) as well as the more specific dominant growth rate criteria (Groups 2 and 3, Table 2.7). However, achieving the general stand height and open grown criteria appears still to conflict with achieving the dominant tree growth rate criteria.

The inability to satisfy all criteria simultaneously arises from neither the parameter search nor WHORL2's Mathematical Structure. The assessment moves to investigating deficiencies in the assessment criteria formulations or the model's Process Structure (Fig. 2.1).

STAGE 5: ASSESSING THE CRITERIA FORMULATIONS

While Stage 4 is primarily concerned with detecting simulations which achieve criteria in unacceptable ways as a result of model structure deficiencies, Stage 5 is primarily concerned with detecting the alternative problem: simulations which achieve criteria in unacceptable ways as a result of poorly formulated assessment criteria. Is each assessment

criterion capturing the particular phenomenon intended? The formulation of the criterion may be too rigid, involving hidden assumptions that produce biased result when not met. Or the assessment criterion may just be formulated in such a way that it does not capture the intended phenomenon.

The inability of Pareto Groups 2 - 8 to achieve all of the general stand height and open grown criteria undermines their attaining of the more specific growth rate criteria. A criteria hierarchy exists to the extent that the general stand criteria and open grown criteria are less specific in their focus than the growth rate criteria. Achieving the more specific but not the more general criteria leads one to suspect that the simulation may satisfy the more specific criteria in an unacceptable manner; recall the clustering-induced bias in the growth rate (Stage 3).

The Pareto Optimal Set can be broken into three broad classes of stand simulations based on the results in Fig. 2.7: Groups 1 and 2 adequately simulate the assessed characteristics of the stand height distribution and trade-off achievement of the open grown and suppressed growth rate criteria for the dominant growth rate criteria (Fig. 2.7, b, c; Table 2.7). Note that in Group 2 the transition from suppressed to dominant tree height, as determined by growth rate, occurs over a narrower height range than in the observed stand (Fig. 2.7 a, c Height Increment plots).

Groups 3, 4, and 5 generate stands with very short, skewed stand height distributions (Fig. 2.7 d, e, f). This is partially due to the shift to slightly larger Sector Increment settings to compensate for the new branch maintenance cost mechanism (Fig. 2.6). Combined with their relatively low Height Increment setting, these groups produce relatively shorter, squatter trees, emphasizing horizontal over vertical growth and resulting in relatively stunted height distributions. As with Group 2, the shorter Height Increment setting (Fig. 2.6) results in a stand with a height transition zone from

suppressed to dominant growth rates that is more narrow than observed in the field (Fig. 2.7 a, d, e, f, Height Increment plots).

In contrast, Groups 6, 7, and 8 generate stands of trees with very tall height distributions dominated by large modes of 4 and 5 m tall trees (Fig. 2.7 g, h, i). These stands' enhanced height and low mortality are the result of their extreme Height Increment settings and physiological parameter settings (E, D, K in Fig. 2.6). The almost identical parameterizations produce tall, slender trees (Crown Angle ~ 6 degrees), minimizing horizontal overlap and maximizing height growth. The reduced overlap results in stands comprised almost wholly of trees with dominant growth rates (Fig. 2.7 g, h, i Height Increment plots).

The growth rate criteria assume that all simulations will generate stands with the transition between suppressed and dominant trees occurring around 3 m in height (Table 2.2). As only trees with heights > 3.2 m are used in the regression, only a subset of the dominant trees in each simulation in Groups 2-7 are used in the regression, biasing the dominant growth rate results (Fig. 2.7). Fitting growth rate regressions to each Pareto Optimal simulations' apparent suppressed and dominant trees (the height ranges defined by the simulation results rather than fixed based on the observed data), reveal that no parameterizations satisfy the dominant R^2 criterion and only Group 8 satisfies the dominant tree growth rate criterion (slope = 0.081 m increment / m height) (Reynolds, 1996). With that exception, all groups generate dominant tree growth rate slopes that are too large (0.089 - 0.149 m increment / m height versus the observed 0.04 m incr. / m ht.). Similarly with the dominant R^2 criterion (0.79 - 0.93 vrs. 0.07).

The presence of Groups 2 - 7 in WHORL2's Pareto Optimal Set (Table 2.7) is an artifact of using a fixed height defining the transition between suppressed and dominant trees used in dominant tree growth rate criteria. In a further development of the model, the breakpoint between suppressed and dominant trees should itself be an assessment

criterion, though automating its calculation may prove difficult. The growth rate regressions should be defined relative to this predicted breakpoint.

The current criteria formulation also assumes a linear growth rate response to tree height. While simplifying the criteria calculations, it poorly captures the observed plateau (Fig. 2.7, (a) Height Increment plot) which appears to a lesser degree even in the simulation results (Fig. 7, b - i).

The Dominant R^2 criterion is also inadequate as an acceptable model isn't really expected to achieve it. The variability in dominant tree growth rates results from a variety of factors, most ignored by the model structure (genotypic variability, microclimate effects, spatial heterogeneity and temporal heterogeneity in soil properties, water, etc.). Achievement of the criteria should generate skepticism regarding the simulation's underlying manner of achievement (it has only been satisfied as a result of WHORL's clustering or WHORL2's biased regressions).

Binary Error Intervals

The binary error measures dictate what satisfies a criterion. Investigating their effect on the Pareto Optimal Set requires multiple replications of the evolutionary optimization routine, and hence was not undertaken. Though it is otherwise impossible to predict the effect on the Pareto Set of expanding any criterion's acceptance interval, the changes from decreasing the intervals can be explored from a plot of the optimal simulation results against each criterion's acceptance interval (Fig. 2.8).

Reducing the intervals of any of five criteria would eliminate at least two groups (Table 2.8), highlighting the tenuous nature of the membership of many of the groups in WHORL2's Pareto Set. In each case, the eliminated groups' membership rests mainly on their achievement of the dominant slope criterion, (in most cases due solely to the poor criterion formulation discovered above). Minor reduction in the dominant slope

criterion's interval would eliminate 6 of the Pareto Set's 8 groups (Table 2.8). This reveals the central role it plays in determining WHORL2's Pareto Optimal and illustrates that in addition to understanding how a model functions it is equally important to learn how it can best be assessed.

This investigation of the criteria formulations and the error measure intervals focuses attention on the data requirements for further model development. More growth data from the stand being simulated would allow refinement and reformulation of the growth rate criteria, provide more informative summary statistics, and extend the database supporting interval limit selection.

Revising the collection of criteria or their error measures is a refinement of the Stage 1 decisions and the Pareto Optimal Model Assessment cycle should be repeated (Fig. 2.1). Aware of the inadequacies in the criteria formulations, the researcher may still proceed to Stage 6 to investigate Process Structure. Such investigations, however, are approached in light of the criteria biases and deficiencies demonstrated in Stage 5.

STAGE 6: DETECTING PROCESS STRUCTURE DEFICIENCIES BY COMPARING CRITERIA ACHIEVEMENTS OF THE PARETO OPTIMAL GROUPS

Deficiencies untraceable to the parameter search (Stage 3), the model's Mathematical Structure (Stage 4), or the criteria selection and formulation (Stage 5), arise from the model's Process Structure (Fig. 2.1). These deficiencies take two forms in the Pareto Optimal Set: (i) assessment criteria that the model is unable to achieve in an acceptable manner with any investigated parameterization, or (ii) sets of assessment criteria that cannot be simultaneously achieved in an acceptable manner. A third possible deficient outcome, satisfying all criteria by an unrealistic parameterization, would result from a deficiency in either the Mathematical Structure or assessment criteria formulations and would have already been detected in Stage 4 or 5.

Investigating the Pareto Optimal simulation results should provide insight into the deficiencies. Special focus should be placed on understanding how the Pareto Optimal Groups fail to satisfy the missing criteria. WHORL2 is unable to achieve simultaneously the general stand height distribution and open grown criteria and the dominant tree growth rate (slope) criterion (Table 2.7). It's Process Structure is missing something to limit or control the growth rate of the tallest dominant trees to the range actually observed (Fig. 2.7 b Height Increment plot). This could result from not incorporating foliage age, nainstem maintenance, or other factors which diminish the resource capture and utilization capacities of the foliage. Model revision to explore this should be undertaken in conjunction with further fieldwork to characterize the functional nature, and variability, of the growth rate of this stand of trees.

The model assessment identified areas of focus for revising both WHORL2's Process Structure and the assessment criteria formulations. Both investigations require further data collection. The assessment cycle then reiterates, employing new criteria to assess the further revised model structure.

The only other possible scenario at this stage of assessment would be complete satisfaction of all criteria by a realistic parameterization -- i.e., an adequate model structure. In this case, the criteria are no longer informative tools for assessing the model structure. A different application of the model as a heuristic may require more precisely focused criteria and therefore a revised assessment.

OTHER APPLICATIONS

Though WHORL(2) is moderately sized and has a well documented Process Structure (Sorrensen-Cothern et al, 1993), these are not inherent requirements of the Pareto Optimal Model Assessment technique. Nor is the technique only applicable to individual based process models, though the example illustrates that these lend themselves to

assessment. The only requirements are the selection of informative criteria, which will be assessed, and the computational facilities to generate the Pareto Optimal Set.

For larger models composed of fairly independently functioning components, it would be best to first use the Pareto Optimal technique to assess the individual components. Each assessment should utilize criteria specifically focusing on the adequacy of the component's dynamics. Once each component is deemed 'adequate', a more comprehensive criteria set should then be chosen for the Pareto Optimal assessment of the complete model structure.

Smaller models should be assessed in the manner applied to WHORL and WHORL2 -- a single Pareto Optimal Set. The advantage of smaller models lies in their generally reduced computational demands, allowing for more thorough assessment and greater understanding of their workings.

The Pareto Optimal technique can also be used as a tool for comparing model structures. In model construction, especially with large models, one is often faced with choosing from competing mathematical representations: e.g., should a simpler representation of photosynthesis be used or a more complex one, containing more parameters for estimation and having greater computational demand? This decision of adequate dynamic representation tends to be overshadowed by a focus on sensitivity analysis and whether the parameters of a given formulation can be inferred from the available data. A more direct assessment of the representations is to construct models with each formulation, generate their Pareto Optimal Sets with respect to the same criteria set, and then compare the Pareto Optimal Sets. Are both structures able to satisfy the same criteria in the same ways?

Similarly, the Pareto Optimal Set can be used as a metric to compare the capabilities of wholly different model structures in satisfying a common set of criteria. For example, in model aggregation, (constructing a simplified model that adequately captures a more

complex model's dynamics), the essential characteristics can be selected as the assessment criteria and the simplified model revised until it simultaneously satisfied the complete set.

CONCLUSION

While a model assessment is never complete (Oreskes et al, 1994), the Pareto Optimal assessment technique described here explicitly demarcates the assumptions behind acceptance of the model structure, defining the context and acceptable parameter ranges for prediction. As the assessment progresses, insight into the model's capabilities and limitations increases, increasing its use as a heuristic. By pinpointing the failures of the model structure, assessment pinpoints the failures of the component hypotheses, and hence of our understanding of the phenomenon. Likewise, use of a model as a heuristic requires defining what constitutes an 'adequate' model, and hence will determine what criteria should be used to assess the model structure. Assessment and use of the model as a heuristic are integral partners, determining and being determined by each other.

By utilizing multiple criteria in a vector, Pareto Optimization provides a powerful and informative technique for assessing ecological process models. Deficiencies are more easily detected, and their sources located in the model structure or the assessment decisions. It requires effort in selecting informative and well formulated criteria, and in undertaking the simulations required to generate the Pareto Optimal Set. But it provides a flexible learning process in which poor decisions can be detected and revised in later iterations.

Table 2.1: Initial parameter space search ranges for generating WHORL's Pareto Optimal Set.

Parameter	*Description*[1]	*Settings*		
Sector Increment	Branch Increment rate, m per unit of relative production	a - 0.015	b - 0.01	c - 0.005
		d - 0.001	e - 0.0005	
Height Increment	Tree Height Increment rate, m per unit of relative production.	a - 0.00016	b - 0.00008	c - 0.00004
Dead	Minimum production required to sustain a sector, in units of relative production.	a[2] - 10 10 10	b - 10 20 30	c - 10 25 40
		d - 1 10 30	e - 1 20 30	f*[3]- 1 20 40
		g - 1 10 40		
E	Efficiency of converting intercepted irradiation to production, relative scale of units of production per irradiation flux.	a - 1.5 1.5 1.5	b - 1.5 1 1	c - 2 1 1
		d - 3 1 1	e* - 3 1.5 1	
D	Minimum relative irradiation level required to sustain living foliage, in units of relative irradiation.	a - 0 2 2	b* - 4 4 4	c - 8 8 8
		d - 0 4 12	e - 4 8 12	f - 0 8 15
K	Coefficient of decline controlling convexity of the foliage response to irradiation decay. Unitless.	a - 0.2 0.2 0.2	b[4] - 0.2 0.3 0.5	

[1] See Sorrensen-Cothern et al (1993) for a more detailed description of the parameters.

[2] Plasticity in physiological parameters is simulated by assigning values to trees as a function of their height class relative to the fifth tallest live tree: short - height <= .55* reference tree height; medium - .55*reference height < height <= .75*reference height; tall - height > .75*reference height. For example, Dead setting g assigns 1 to short trees, 10 to medium height trees, and 40 to tall trees.

[3] * - denotes the baseline parameter settings from Sorrensen-Cothern et al (1993).

[4] K setting b was investigated only in conjunction with D settings a, d, e, and f.

Table 2.2: WHORL's Assessment Criteria.

	Criterion	*Description*	*Binary Error Interval (Observed)*
Stand Criteria	Mortality	Cumulative Mortality, year 30.	77, 137 (107)
	Stand Height Frequency Distribution	Predicted and observed live height frequency distributions compared with the Kolmogorov-Smirnov two-sample test.	p-value > .01
	Median Live Tree Height	A robust measure of tree height central tendency.	2.82, 3.35[5] (3.08 m)
Open Grown Criteria	Number of Live Whorls	The number of living whorls on a 30 year old open grown tree.	9, 17 (13)
	Crown Angle	Angle formed by the tips of live branches relative to trunk.	10, 15 degrees
	Crown Length Ratio	Ratio of live crown length to tree height.	accept $\geq$ 90%
Growth Rate Criteria	Suppressed Tree Growth Rate	Slope estimate from linearly regressing two-year height increment on height, at year 28, of trees $\leq$ 2.8 m (Fig. 2.2).	0.014, 0.046[6] (0.03 m/m)
	Suppressed Tree R^2	The variability in suppressed tree height increment rates, measured by the coefficient of determination from the regression above (Fig. 2).	0.04, 0.44 (0.24)
	Dominant Tree Slope	Same as Suppressed Tree Growth Rate, for trees $\geq$ 3.2 m.	-0.005, 0.085 (0.04 m/m)
	Dominant Tree R^2	Same as Suppressed Tree R^2, for trees $\geq$ 3.2 m.	0.0, 0.27 (0.07)

[5] 95% Confidence Interval for observed median live tree height.

[6] Observed regression estimate $\pm$ 3*sigma-of-estimate.

Table 2.3: WHORL's Initial Pareto Optimal Set. A shaded cell denotes that the group of parameterizations adequately simulates the criterion.

Group	# Parameterizations	Mortality	Ht. Distribution	Median Ht.	# Live Whorls	Crown Angle	Crown Ratio	Sup. Slope	Sup. R^2	Dom. Slope	Dom. R^2
1	1										
2	1										
3	14										
4	9										

Table 2.4: Parameter settings of each of WHORL's four Initial Pareto Optimal Groups, relative to the investigated settings. A shaded cell marks the parameter settings of the elements of the Pareto Optimal Group depicted on that row. Some groups contain more than one element, hence more than one setting per parameter may be marked. Parameter labels correspond to those given in Table 2.1. Note: increment parameters are listed from largest to smallest setting; physiological parameters are listed from least plastic to most plastic.

Group	Sector Increment					Height Incr.			Dead						
	a	b	c	d	e	a	b	c	a	b	c	d	e	f	g
1					■		■				■				
2					■		■								■
3					■			■		■		■	■		
4					■			■			■			■	■

Table 2.4: continued. Parameter settings of each of WHORL's four Initial Pareto Optimal Groups, relative to the investigated settings. Note: physiological parameters are listed from least plastic to most plastic.

Group	E						D						K	
	a	b	c	d	e	f	a	b	c	d	e	f	a	b
1														
2														
3														
4														

Table 2. 5: Extended Parameter Search Ranges. For model WHORL, parameters K, D, and E require three settings, one for each height class. E. g., K1=0.1, K2=0.2, K3=0.3. For the model WHORL2, parameters K, D, and E require two settings, one for sun and one for shade foliage. Each of these parameters has an associated Switch setting dictating the irradiation level at which the foliage characteristic changes from sun to shade.

Parameter	*K*	*D*	*E*	*Switch*	*Height Increment*	*Sector Increment*
Maximum	0.5	12	3.0	1.0	0.00010	0.0007
Minimum	0.1	0	1.0	0.0	0.00002	0.0001
Scale	0.1	1	0.5	0.05	0.00001	0.0001

Table 2. 6: WHORL's Pareto Optimal Set. A shaded cell denotes that the group of parameterizations adequately simulates the criterion.

Group	# Parameterizations	Mortality	Ht. Distribution	Median Ht.	# Live Whorls	Crown Angle	Crown Ratio	Sup. Slope	Sup. R^2	Dom. Slope	Dom. R^2
1	65										
2	18										
3	1										
4	39										
5	1										
6	53										
7	2										
8	2										

Table 2. 7: WHORL2's Pareto Optimal Set. A shaded cell denotes that the group of parameterizations adequately simulates the criterion.

Group	# Parameterizations	Mortality	Ht. Distribution	Median Ht.	# Live Whorls	Crown Angle	Crown Ratio	Sup. Slope	Sup. R^2	Dom. Slope	Dom. R^2
1	260										
2	1										
3	2										
4	4										
5	2										
6	1										
7	3										
8	1										

Table 2. 8: Effect on WHORL2's Pareto Optimal Set of reductions in criteria error measure intervals. `Group A >p Group B' means Group A dominates Group B.

Criterion Error Interval Reduction	Change in Pareto Set
Increase Mortality Lower Bound	Group 8 >p Groups 6 and 7
Increase Crown Ratio Lower Bound	Group 6 >p Groups 7 and 8
Decrease # Live Whorls Upper Bound	Group 7 >p Groups 4 and 8
Decrease Suppressed Slope Upper Bound	Group 5 >p Groups 4 and 6
Decrease Dominant Slope Upper Bound	Group 1 >p Groups 2, 3, 4, 6, 7, 8

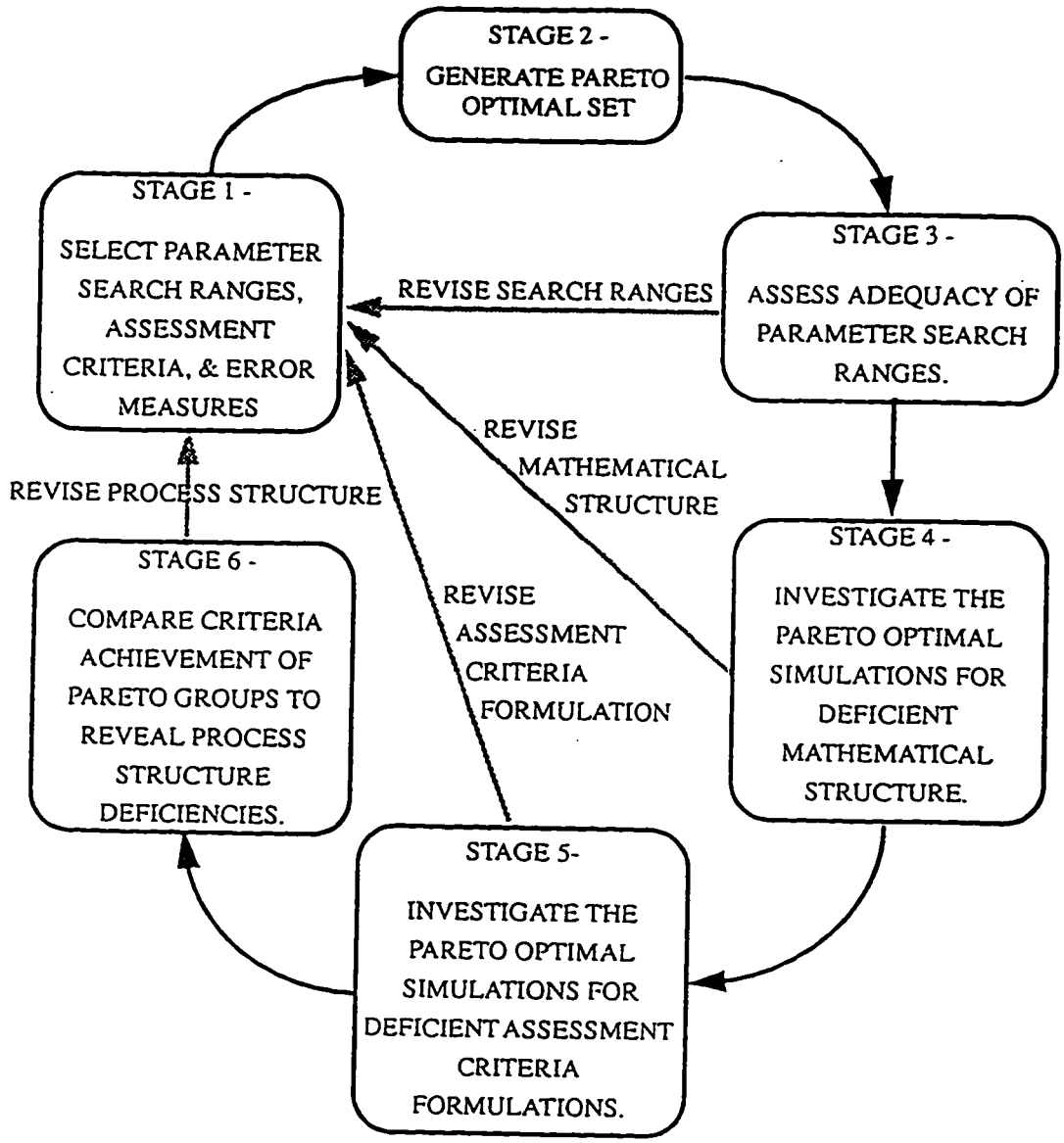


Fig. 2.1: Pareto Optimal Model Assessment Cycle. Locating a deficiency in Stages 3 - 5 requires reinitiation of the cycle after the correction (gray arrows). Otherwise, assessment proceeds to the next stage (black arrows). If no deficiencies are located by the end of Stage 6, either the error measures should be made more restrictive or more refined criteria employed.

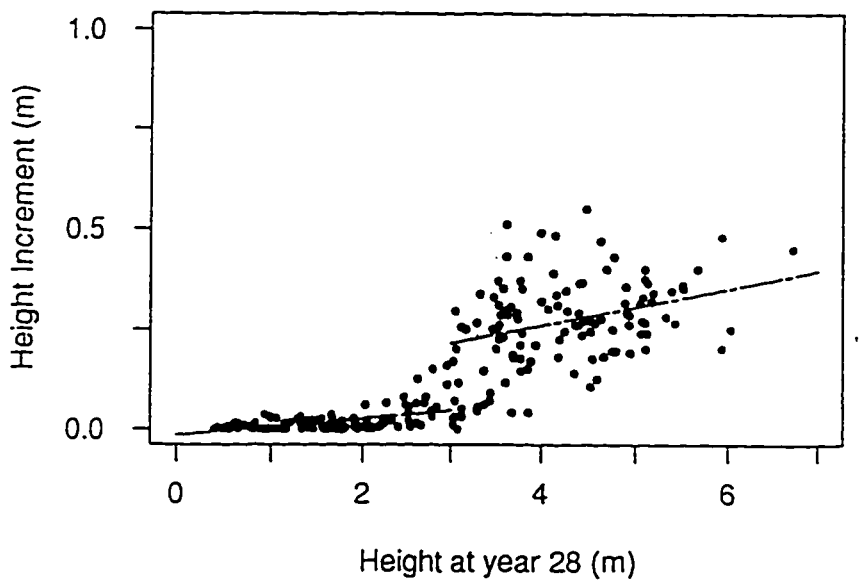


Fig. 2.2: Observed 2 year height increment by initial tree height at year 28. Regression lines are based on trees < 2.8 m tall for suppresseds, and trees > 3.2 m tall for dominants. The slope estimates and coefficients of determination are assessment criteria.

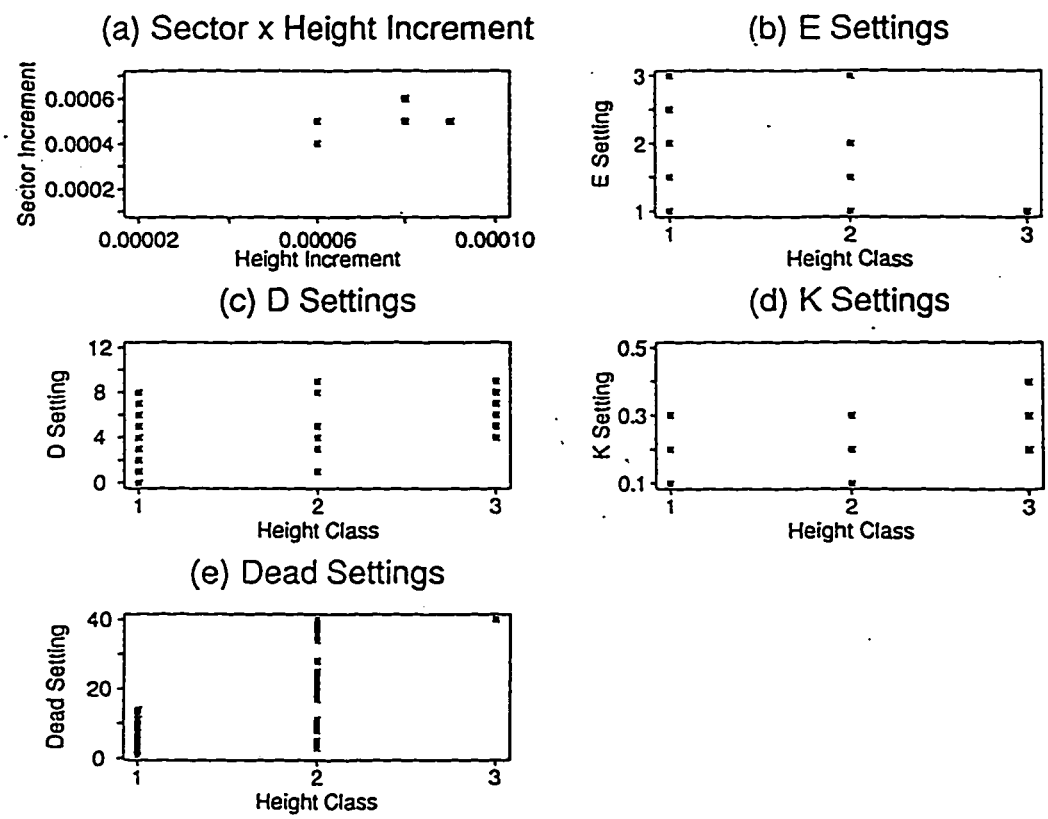


Fig. 2.3: Parameter settings of the elements in WHORL's Pareto Optimal Set, plotted relative to each parameter's search range. Plots (b) - (e) show the parameterization at each height class. No parameter is uniformly selected at an extreme of it's search range, (E is forced to have a lower bound of 1 by definition, see Sorrensen-Cothern et al (1993)).

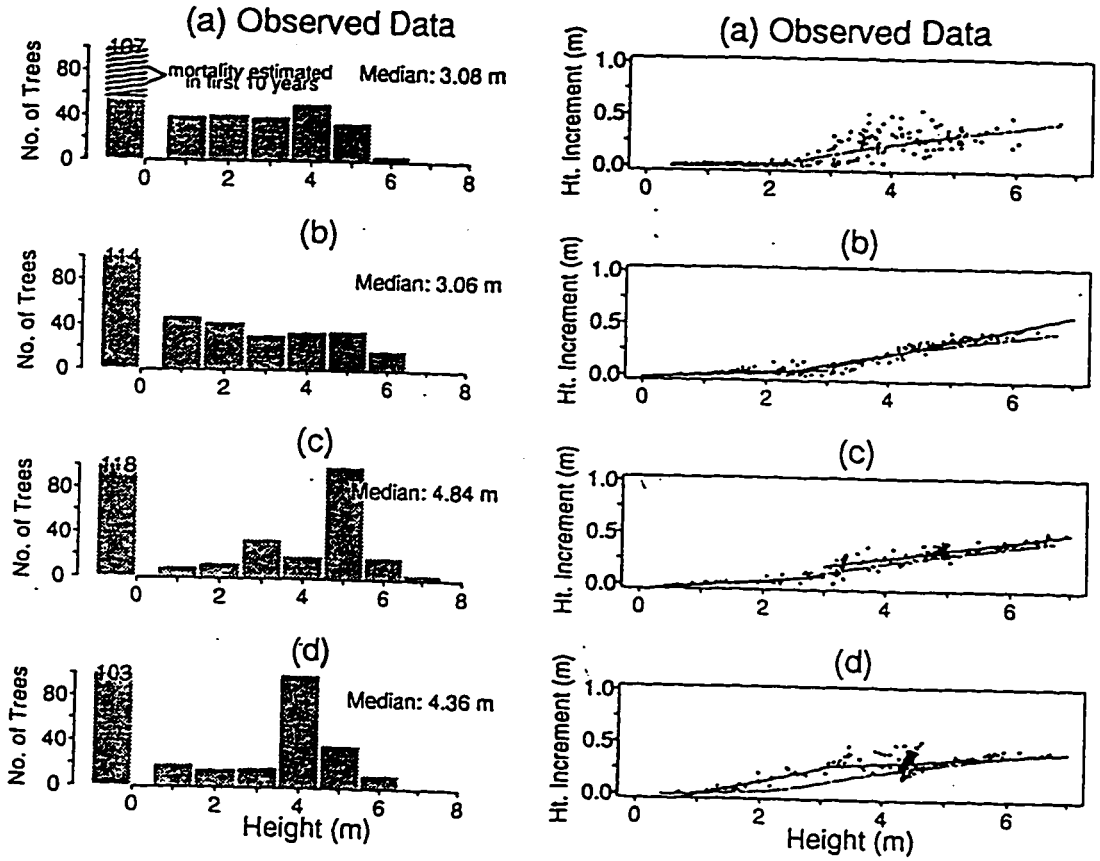


Fig. 2.4: Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations for each of WHORL's Pareto Optimal Groups: (a) Observed data, (b) Group 1, (c) Group 2, (d) Group 3, [continued next page] (e) Group 4, (f) Group 5, (g) Group 6, (h) Group 7, (i) Group 8. The leftmost column of each histogram represents mortality. Two year height increment plots display the suppressed and dominant tree growth rate regressions for the simulation (solid lines) and, for reference, a smooth of the observed data (broken line, also shown in (a)).

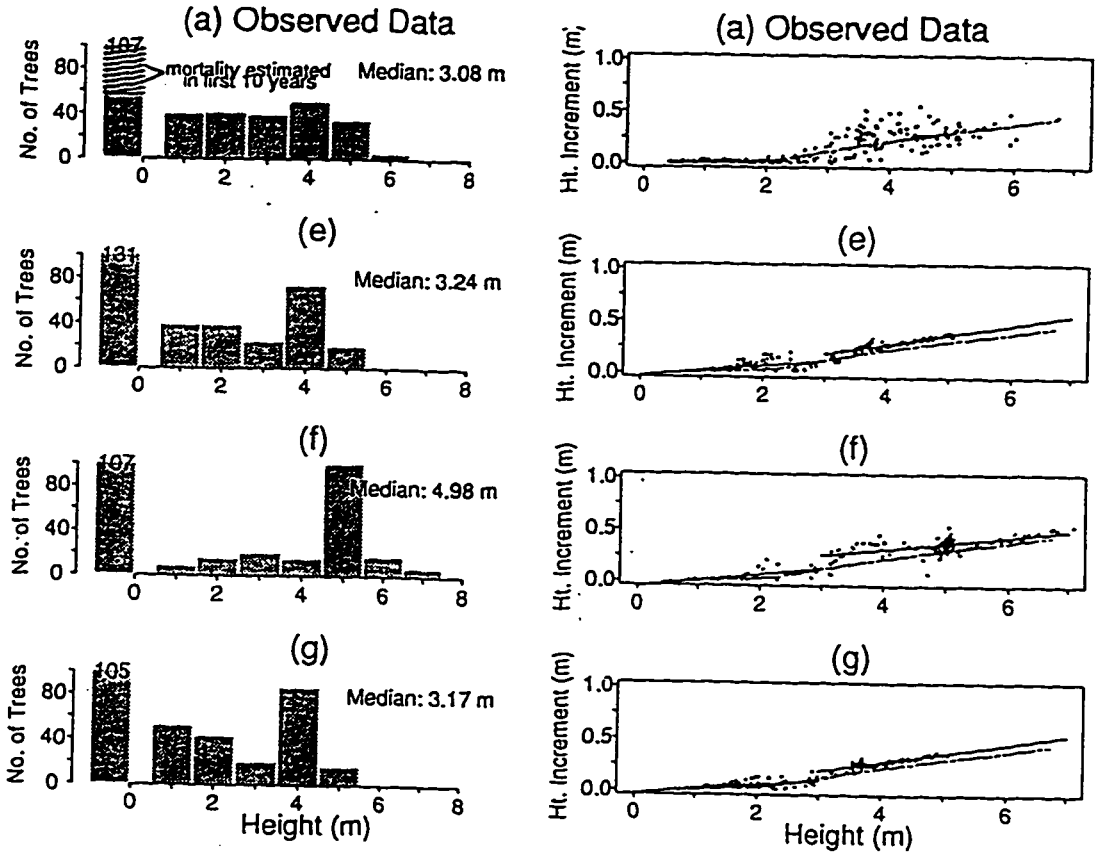


Fig. 2.4: continued. Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations for each of WHORL's Pareto Optimal Groups: (a) Observed data, (e) Group 4, (f) Group 5, (g) Group 6.

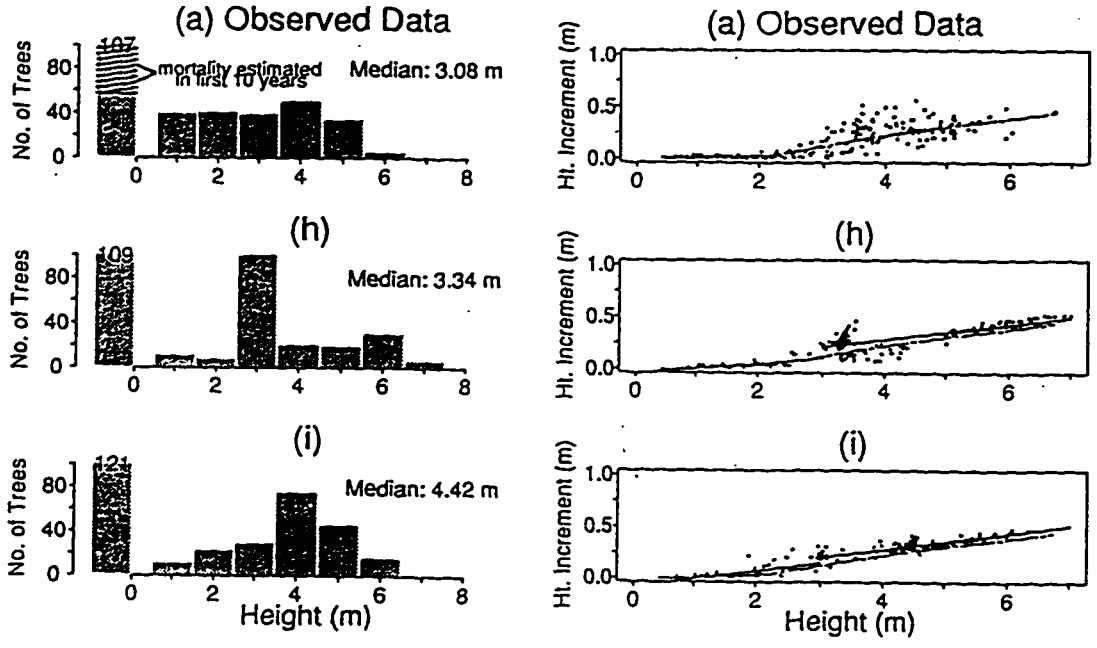


Fig. 2.4: continued. Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations for each of WHORL's Pareto Optimal Groups: (a) Observed data, (h) Group 7, (i) Group 8.

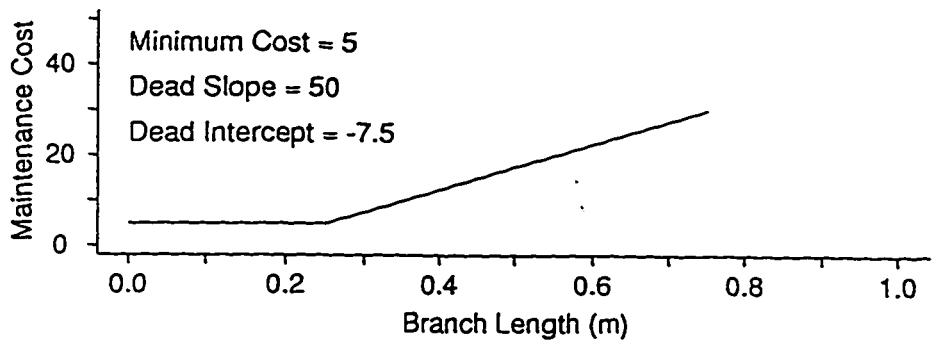


Fig. 2.5: Revised branch maintenance cost function with example Dead parameters -- minimum maintenance cost, slope, and intercept.

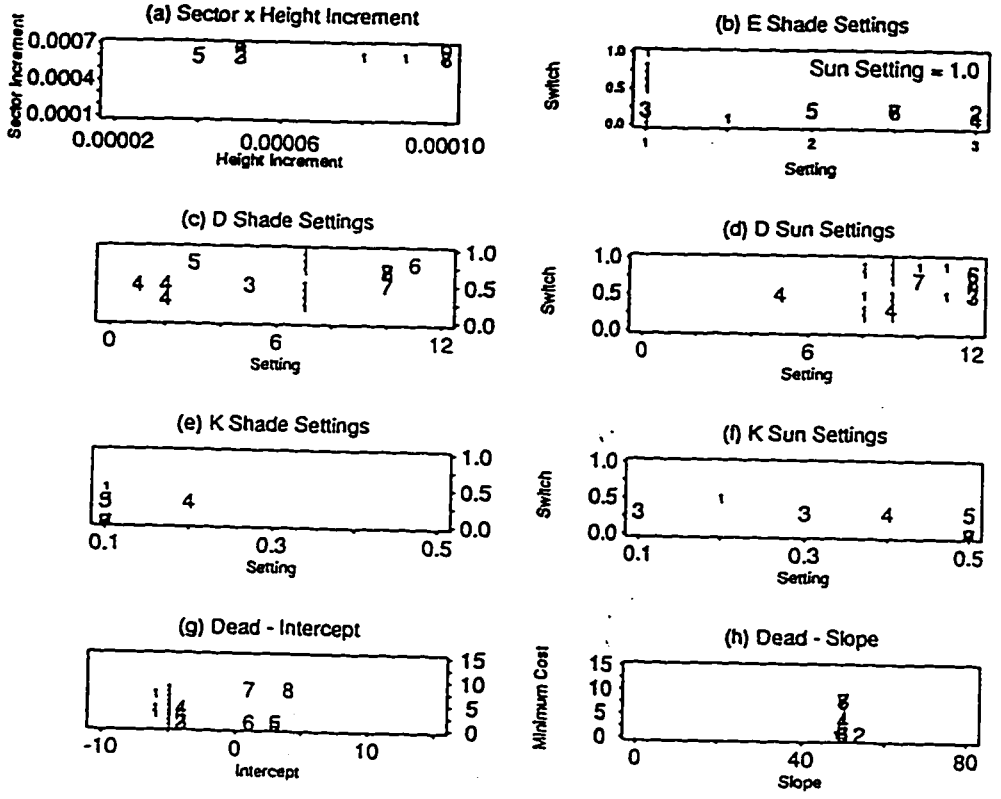


Fig. 2.6: Parameter settings of WHORL2's Pareto Optimal Set elements, plotted relative to each parameter's search range and identified by Pareto Optimal Group. Plots (b), (c), and (e) display shade foliage settings; plots (d) and (f) display sun foliage settings. Switch axis on plots (b) - (f) records the irradiation level, as % of above canopy irradiation, below which foliage characteristics change from sun to shade. The switch setting can differ for each physiological parameter (E, D, K). For example, K settings for Group 4 are 0.2 for shade foliage (receiving less than 25% full irradiation) and 0.4 for sun foliage.

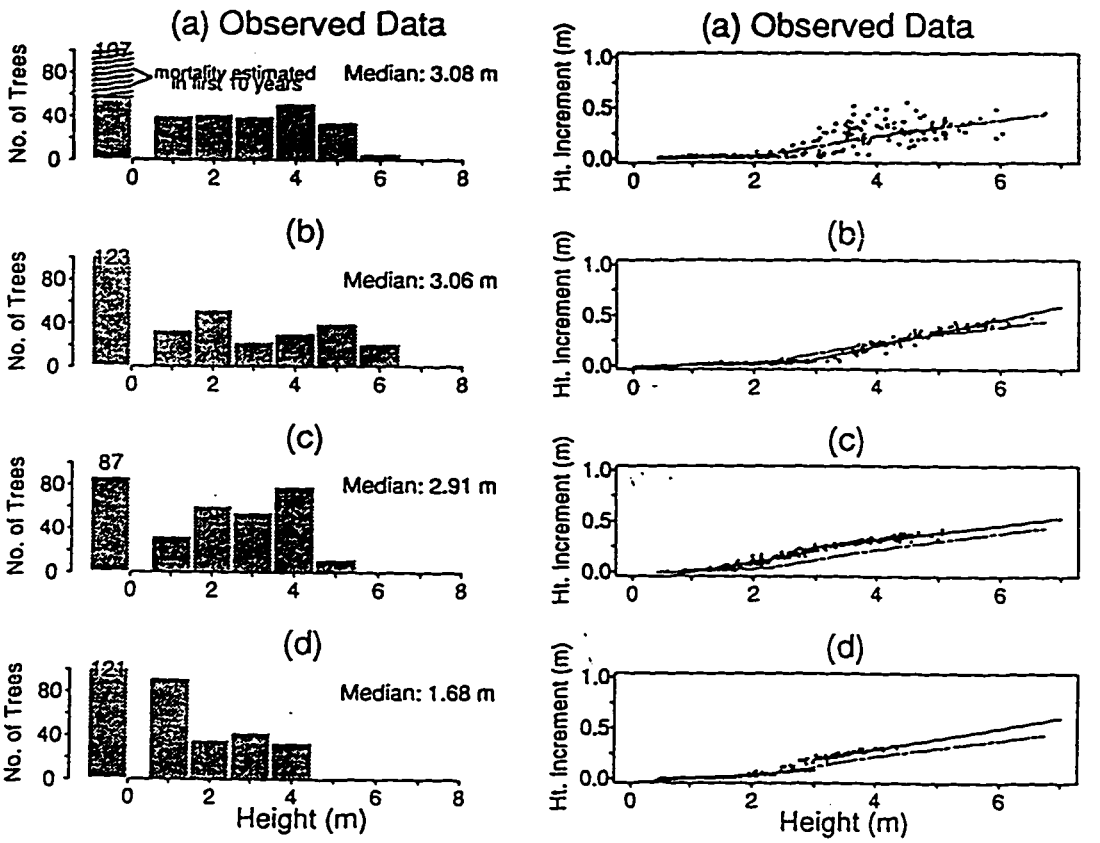


Fig. 2.7: Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations of WHORL2's Pareto Optimal Groups: (a) Observed data, (b) Group 1, (c) Group 2, (d) Group 3, [continued on next page] (e) Group 4, (f), Group5, (g) Group6, (h) Group 7, (i) Group 8. The leftmost column of each histogram represents dead trees. The two year height increment plots also display the suppressed and dominant tree growth rate regressions for the simulation (solid lines) and, for reference, a smooth of the observed data (broken line, also shown in (a)).

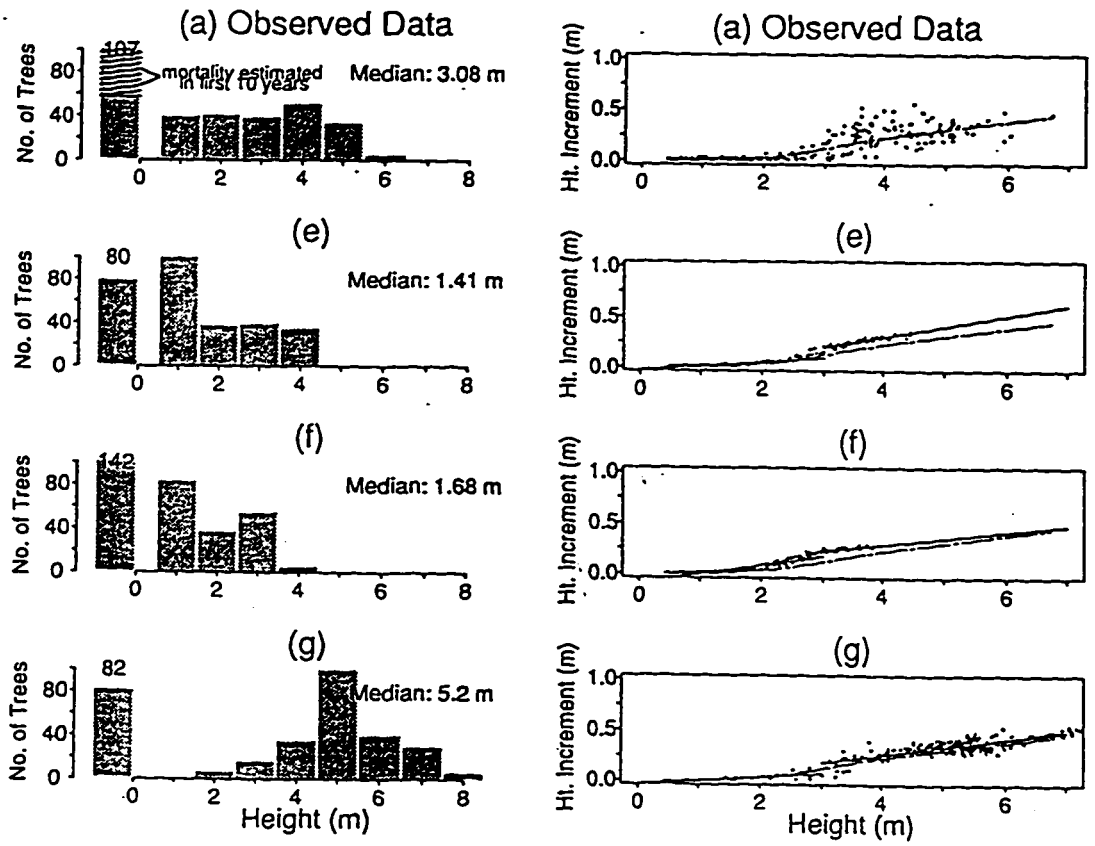


Fig. 2.7: continued. Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations of WHORL2's Pareto Optimal Groups: (a) Observed data, (e) Group 4, (f), Group5, (g) Group6.

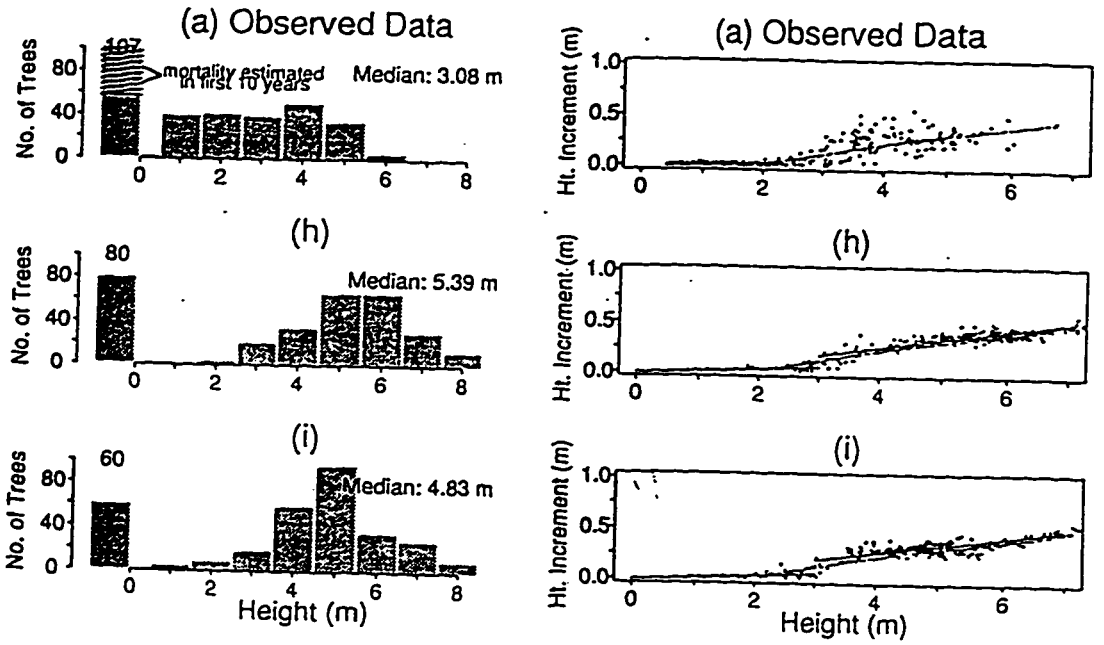


Fig. 2.7: continued. Tree height frequency distribution at year 30 and relationship of tree height at year 28 to subsequent two year height increment, for representative parameterizations of WHORL2's Pareto Optimal Groups: (a) Observed data, (h) Group-7, (i), Group8.

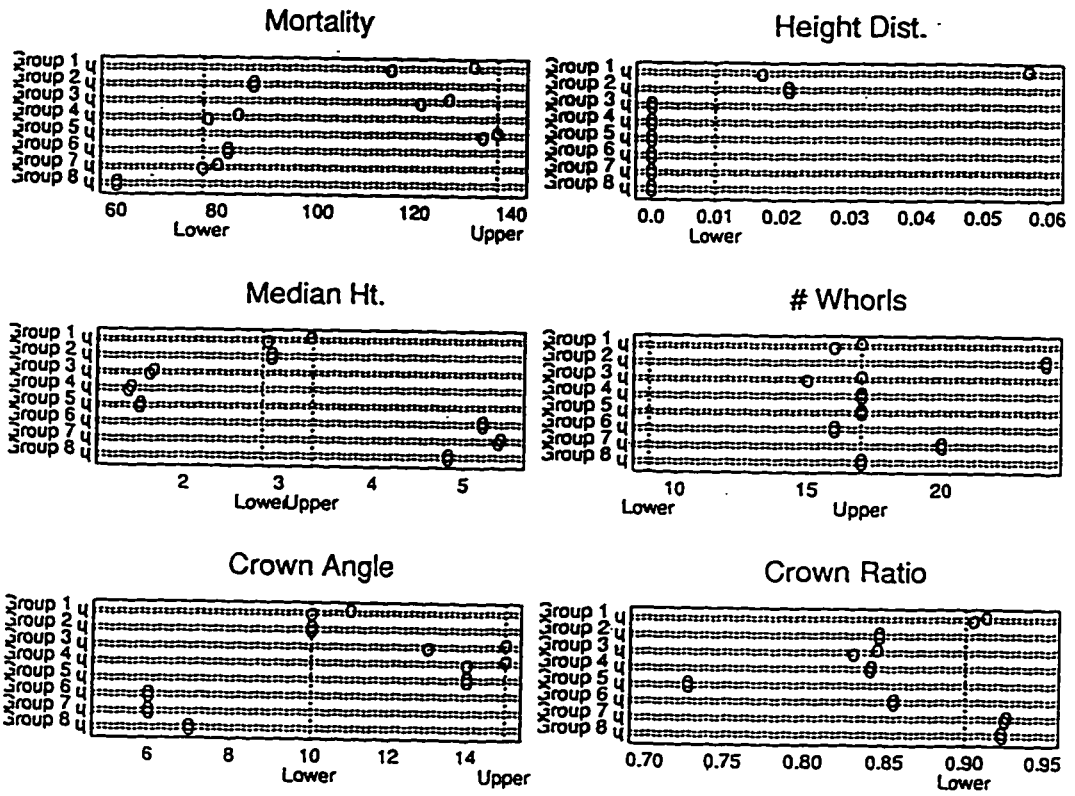


Fig. 2.8: Lower and Upper limits of WHORL2's Pareto Optimal simulations for each criterion. Lower limit of simulation results marked by 'l', upper limit marked by 'u' on left-hand axis. Error interval limits (Table 2) are marked by vertical lines labeled "Lower" and "Upper" on the x-axis of each plot. Limits beyond the graph range are not shown. Example: Group 8's simulation under-predicts Mortality and over-predicts Median Live Height.

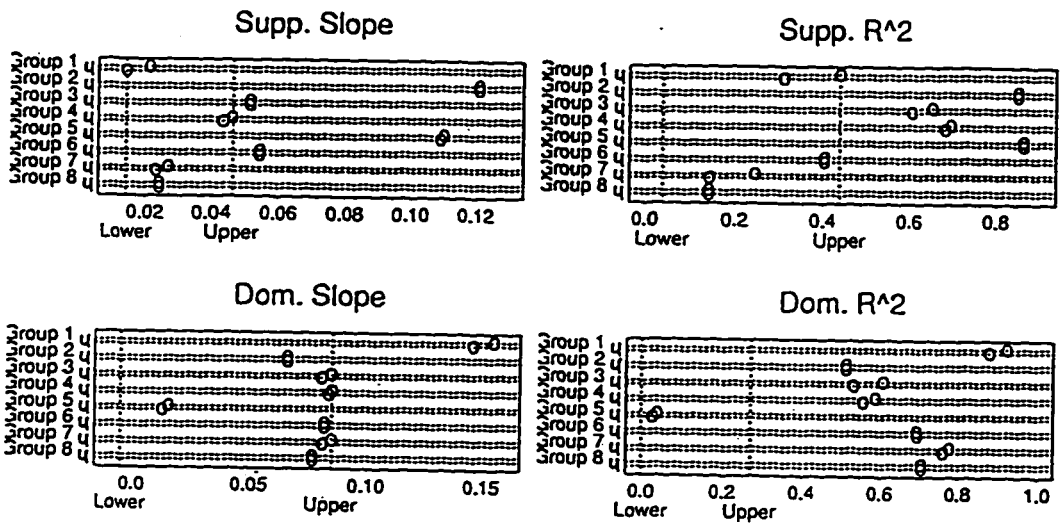


Fig. 2.8: continued. Lower and Upper limits of WHORL2's Pareto Optimal simulations for each criterion.

CHAPTER 3: USING EVOLUTIONARY OPTIMIZATION TO GENERATE A PARETO OPTIMAL SET

INTRODUCTION

Generating the Pareto Optimal Set involves optimizing a vector of incommensurable criteria and therefore poses difficulties for traditional gradient based optimization methods. This chapter presents a simulated evolution optimization algorithm for generating the Pareto Optimal Set when binary interval-type error measures are used for each criterion. The computer code is presented in Appendix B. The Pareto Optimal Set of the canopy competition model WHORL (S-C) generated by the evolutionary optimization algorithm is compared to that from a forward search on a lattice.

SIMULATED EVOLUTION OPTIMIZATION

Simulated evolution is a suite of techniques to evolve solutions to optimization problems by simulating natural selection (Michalewicz, 1992; Fogel, 1994). These include Genetic Algorithms (Goldberg, 1989; Holland, 1992), Evolutionary Strategies (Bäck et al, 1991), Evolutionary Programming (Fogel, 1994), and their extensions (Michalewicz et al 1992). These techniques share a basic procedure but differ in their finer details of operation. The terms used are borrowed from genetics. A population of parameterizations is selected and a simulation run for each. Each parameterization is assigned a fitness based on how well it's simulation satisfies the selection criterion, the optimization problem. Parameterizations are then selected from the population for breeding the next generation's population, the probability of being selected to breed being a function of a parameterization's fitness. Different techniques employ different breeding strategies to generate the next population. The two basic strategies are: random parameter mutation, in

which a component parameter is selected from a parameterization and its setting randomly shifted, and crossover recombination in which two randomly selected parameterizations exchange portions of their parameter settings. The process is stopped either after a fixed number of generations or upon the evolution of a parameterization that sufficiently optimizes the selection criterion. Each generation searches multiple locations in the parameter space, the more 'fit' locations producing more 'offspring' and therefore a locally more intensive search.

Simulated evolution places few mathematical constraints on the form of the optimization criteria, requiring only that a procedure be defined for selecting the next generation's parents using a measure of achievement of the selection criteria. Membership in the Pareto Optimal Set can therefore be incorporated into the selection procedure.

THE ALGORITHM: OVERVIEW

The simulated evolution routine ran for 50 generations, each generation producing 75 offspring (parameterizations) selected from the current parent population. The parent population consists of all parameterizations in the current Pareto Optimal Set, supplemented with dominated parameterizations from the last generation's offspring for genetic variety. The parent population's fluctuating size is always larger than 75. A parameterization's fitness, used in selecting the next generation's offspring, is a function of both the number of criteria satisfied and membership in the Pareto Optimal Set. Offspring are randomly assigned to undergo either single parameter mutations (within a given search range) or crossover recombination with another parameterization. The initial population consisted of the Pareto Set from the earlier latticed-based optimization search (Chapter 2, Stage 2) augmented with randomly selected parameterizations for genetic diversity. The final Pareto Set (Table 3.1) was judged stable, having existed without change in the group Assessment Vectors for 13 generations.

OFFSPRING SELECTION

Increasing genetic diversity allows a broad exploration of the parameter space by reducing the resemblance of an offspring parameterization to either of its parents (assuming a crossover operator is employed). It is therefore more desired in the early generations than in later ones when locally refined searches become more important (refs). This shifting role of diversity is incorporated into each generation's offspring selection.

The current parent population consists of the current Pareto Optimal Set and the current dominated parameterizations. The parameterizations of the Pareto Optimal Set are partitioned into groups by their assessment vectors as in the example above. The dominated parameterizations come from two sources: parameterizations from the last generation that are dominated by the current Pareto Optimal Set, and parameterizations from last generation's Pareto Optimal Set that were dominated by parameterizations in the current Pareto Optimal Set.

75 offspring are selected as follows:

1. Two parameterizations randomly from each Pareto Optimal group.

2. Randomly from the Pareto Optimal Set, for a combined total with those from step 1 of 35 + Generation #, or 65, whichever number is smaller. For example, if it is the 20^{th} generation, a total of 35 + 20 = 55 offspring will be selected from the Pareto Optimal Set. If the Pareto Optimal Set contains 5 groups, these 55 will consist of 2 randomly chosen from each group (10 total) and 45 randomly chosen from the complete Set based on each group's relative fitness (see below).

3. The remainder are randomly selected from the list of recently dominated parameterizations based on relative fitness (see below).

FITNESS

Each parameterization's fitness is calculated as the sum, across all criteria, of the following score function:

S(criterion i) = 1	if the simulation result is within the binary error interval $[a_i, b_i]$.
S(criterion i) = .5	if the simulation result is outside the interval, but within the interval $[a_i - 0.1*(b_i - a_i), bi + 0.1*(b_i - a_i)]$. I.e., the simulation result is relatively near the interval limits.
S(criterion i) = 0	Otherwise.

Any score function could be employed, including criterion specific continuous score functions. However, each criterion's score function must be normalized to maintain the essential equality of criteria which underlies the Pareto Optimal Set. If employing continuous error measures rather than binary error measures, the offspring selection scheme must be adjusted to compensate for the fact that the Pareto Optimal Set will no longer partition into groups of parameterizations producing common assessment vectors.

All parameterizations in a particular Pareto Optimal group will have the same fitness, since it is a function of the assessment vector. Selecting offspring from the Pareto Optimal Set as a whole is therefore a two-step process: (1) select the Pareto Optimal group, where $Pr(select\ group\ i) = \frac{fitness\ of\ group\ i's\ assessment\ vector}{\sum_{all\ groups} fitness\ of\ assessment\ vector\ j}$; (2) randomly select the offspring from the selected group, where $\Pr(select\ parameterization\ i) = \frac{1}{Number\ of\ parameterizations\ in\ the\ group}$.

Offspring are selected from the dominated parameterizations using $Pr(select\ parameterization\ i) = \frac{fitness\ of\ parameterization\ i}{total\ fitness\ of\ all\ dominated\ parameterizations}$.

GENETIC OPERATORS

Each offspring is assigned a genetic operator when it is selected, either mutation or crossover. These probabilities of assignment were kept constant each generation; Pr(mutation) = 0.60, Pr(crossover)=0.40. They could be made to change with the generations, allowing, for example, a decreasing probability of crossover for greater local search refinement as the routine progressed.

NONUNIFORM MUTATION

Nonuniform mutation decreases the possible mutation size as the generation increases, producing a more refined local search in the later generations (Michalewicz, 1992). A parameter is randomly selected from the parameterization, and a coin toss simulated to decide if the current setting should be increased or decreased. The current setting is then changed by the amount: $Parameter\ Adjustment = Y*(1-r^{(1-k/N)^b})$, where Y is the search range available in the chosen direction, r is a random draw from a Uniform[0,1] distribution, k is the generation number, N is the maximum number of generations, and b is a system parameter controlling how quickly the possible mutation size reduces. Following Michalewicz (1992), b=5 was utilized. The adjustment was then rounded to the nearest whole multiple of a parameter-specific increment size, controlling the regularity of the settings explored.

CROSSOVER

A list of parameterizations assigned to the crossover operator is randomly sorted, then split into sequential pairs. For each pair of offspring, a distance is randomly chosen along

the sequence of parameters that define the parameterizations. Each parameterization is split at this point and the two offspring swap sections. For example, assume the model has five parameters, (p_1, p_2, p_3, p_4, p_5), and the two selected parameterizations are (1,1,1,1,1) and (2, 2, 2, 2, 2). A split distance of 3 produces two new parameterizations from the crossover: (1,1,1, 2, 2) and (2, 2, 2, 1, 1).

EXAMPLE

The Pareto Optimal Set of the canopy competition model WHORL (Sorrensen-Cothern et al, 1993) was generated both by simulated evolution (Table 3.1) and a forward search of a lattice of parameterizations (Table 3.2). The two optimization approaches were restricted to the same parameter ranges (the extended parameter ranges defined in Table 2.4), and were initiated using the Pareto Optimal Set calculated from an initial lattice search of 7560 simulations (Table 2.3). This was used to revise the parameter search ranges (Table 2.4) and as part of the simulated evolution program's founding population. The simulated evolution results are based on 50 generations of 75 parameterizations each, or 3750 total simulations beyond the initial lattice search. The forward search employed 7916 total simulations beyond the initial lattice search. Ten criteria were employed (described in Chapter 2).

The simulated evolution optimization routine, coded in C, took approximately 5.2 days running in the background on five heavily utilized Sparc Servers (2 Sparc 20's, 2 Sparc 10's, 1 Sparc +). 99% of this time was spent running the model simulations. Though the time wasn't tracked, the forward search took a minimum of 2.11 times as long (7916 / 3750 = 2.11 times as many simulations).

The simulated evolution Pareto Set (Table 3.1) extends that produced from the forward search (Table 3.2). For example, Group 7 of Table 1 dominates Groups 3,4, and 5 of

Table 3.2. Simulated evolution is a much more efficient technique for generating a model's Pareto Optimal Set.

Table 3.1 Pareto Optimal Set generated by Simulated Evolution. A shaded cell denotes the criterion is satisfied.

Group	Mortality	Ht. Distribution	Median Ht.	# Live Whorls	Crown Angle	Crown Ratio	Sup. Slope	Sup. R^2	Dom. Slope	Dom. R^2
1	■	■	■	■	■	■	■	■		
2	■			■	■	■	■	■	■	
3	■			■	■	■			■	■
4	■	■	■		■	■	■	■	■	
5	■			■		■		■	■	■
6	■	■	■	■	■		■	■	■	
7	■	■		■	■	■			■	
8	■			■	■	■		■	■	

Table 3.2 Pareto Optimal Set generated by Forward Search. A shaded cell denotes the criterion is satisfied.

Group	Mortality	Ht. Distribution	Median Ht.	# Live Whorls	Crown Angle	Crown Ratio	Sup. Slope	Sup. R^2	Dom. Slope	Dom. R^2
1	■	■	■	■	■	■	■			
2	■			■	■	■		■	■	
3	■				■	■			■	■
4	■			■	■				■	■
5				■		■			■	■
6	■	■	■		■		■		■	
7	■	■	■		■			■	■	
8		■			■		■	■	■	
9	■			■		■	■	■	■	
10		■		■	■			■	■	
11	■	■	■	■	■				■	

CHAPTER 4: USING WHORL2 TO CRITIQUE THE REPRESENTATION OF CANOPY COMPETITION IN SELF-THINNING MODELS

THE SELF-THINNING RELATIONSHIP

Self-thinning, expressed as the relationship between increasing mean plant biomass, ϖ, and decreasing population density, N, in crowded, even-aged monospecific stands, has been a focus of research since Yoda et al (1963) asserted the constancy for all species of the slope γ in:

$$\log(\varpi) = \log(\kappa) + \gamma * \log(N) \text{ (eqn. 4.1),}$$

where κ is a species-specific constant. Researcher's found wide empirical support for this linking of plant growth and competition-induced mortality via the slope γ = -3/2 (see reviews in White, 1981; Westoby, 1984; Weller, 1987), and a number of theoretical models derived the relationship from proposed rules of plant geometry and growth (Yoda et al, 1963; White, 1981; Pickard, 1983). Re-analysis of the empirical evidence has revised interpretation of the self-thinning relationship to a species and site specific phenomenon (Zeide, 1985, 1987; Weller, 1987a, 1987b, 1988; but see Osawa and Sugita, 1989; Weller, 1990; Lonsdale, 1990; Weller, 1991), shifting the fundamental question from 'why -3/2?' to 'what controls the competition process during stand development?'. Though many theoretical models have been proposed to explain self-thinning (Yoda et al, 1963; White, 1981; Zeide,1987; Pickard, 1983; Weller, 1987b; Norberg, 1988; Clark, 1992b), insight into the controlling processes is still limited (Zeide, 1985; Westoby, 1984; Pickard, 1983; Clark, 1992b; Osawa & Allen, 1993). I suggest that this is partly due to the fundamental limitations of the way these models represent competition.

Theoretical models proposed to explain the self-thinning relationship contain two components: a model relating the dimensions of an individual tree's crown and perhaps mainstem to a measure of tree size, and a relationship uniting crown size and stand mortality. For example, Norberg (1988) uses structural support requirements to derive a relationship between trunk diameter and stemwood volume, from which he then derives crown width and stand mortality; Clark (1992) presents a temporal model of projected crown area growth from which he derives aboveground biomass and stand mortality. Other models utilize static relations among the crown dimensions based on empirical allometric investigations (White 1981; Yoda et al 1963; Pickard 1983; Weller 1987b; Zeide 1987).

Though tree size is represented by biomass in equation 4.1, crown volume has been proposed as a more appropriate measure (Westoby, 1984; Weller, 1987a; Zeide, 1985; Norberg, 1988). In fact, the model of Yoda et al (1963) was developed in terms of crown volume and appeal then made to the assumption that biomass is proportional to volume to arrive at eqn 1 from:

$$\log(\overline{Crown\ Volume}) = \log(\kappa') + \gamma \log(N). \text{ (eqn. 4.2).}$$

The literature contains no clear definition of which volume measurement should be used in measuring self-thinning: Yoda et al (1963) discuss crown spatial volume, or hull; Norberg (1988), stemwood volume. In terms of growth and competition, crown foliage volume, the volume from which the tree acquires resources, better represents a tree's current status than integrative measures like biomass, trunk volume, or even crown volume sensu Yoda et al (1963).

The essential link uniting individual plant growth and density dependent (DD) mortality in all of the models mentioned is the assumption that mean projected crown area is inversely proportional to the stand density:

$$\overline{Crown\ Area} \propto N^{-1} \text{ (eqn. 4.3).}$$

This summary of the canopy competition process is the core of these models' explanations of self-thinning. But this amounts to a priori assuming a formula for the very process whose dynamics are to be investigated (Ford & Sorrensen, 1992; Yokozawa & Hara, 1992). To the extent that this relationship is a poor summary of canopy competition, the models will be poor expositions of the underlying processes.

Adopting the $\overline{Crown\ Area} \propto N^{-1}$ relationship as a summary of the canopy competition process has a number of implications:

1. The self-thinning model can apply only when the sum of the projected crown areas is constant. That a stand should have reached canopy closure is usually assumed sufficient to justify this, though equation (4.3) doesn't require full closure. For a constant sum of individual projected crowns to follow from constant canopy projection (closure), crown overlap among trees must remain constant, or at least represent only a negligible fraction of the projected crowns' total area.
2. Modeling focuses on the average tree, ignoring the variability among tree sizes and resource acquisition which drives the competition process. The stand is perceived as consisting of N trees identical to the average tree (in height, crown volume), and the trajectory of the mean tree is assumed to capture the competition process (Norberg, 1988; others, implicit; see Osawa & Allen, (1993) for a critique of the allometric errors inherent in this misconception).
3. Canopy competition is modeled as a two-dimensional packing process among identical trees. When one dies the remaining trees' crowns can grow to reoccupy the vacated horizontal space, making growth a response to mortality. Horizontal crown

expansion is the only relevant aspect of growth as area, not volume occupancy, is the essential driving variable. Consequentially, crown growth models tend to be constructed from more convenient observables (stemwood volume, dbh, or biomass for smaller plants) and allometric relationships (empirical or theoretical) used to derive crown width. This replaces direct investigation of the relationship between crown volume and DD mortality with two approximations (Fig. 4.0). Modeling canopy competition as a packing process entails that all competition is 2-sided and symmetric.

4. Though changes are expected within the population's spatial distribution, by focusing on the mean tree the proposed models do not explicitly consider the effect of spatial aggregation on the self-thinning process.

Investigating these assumptions requires detailed understanding of tree size, crown volume, crown interactions, and mortality throughout the course of stand development. The difficulty of acquiring observations of this may be one reason self-thinning models are often assessed simply by comparing the predicted self-thinning slopes with empirical results (Pickard, 1983). More detailed scrutiny must be applied to self-thinning models if they are to increase our understanding of the underlying processes.

SIMULATION INVESTIGATION OF THE SELF-THINNING PROCESS

The proposed mortality relationship (eqn. 4.3), and each of the entailed assumptions listed above, were investigated using WHORL2 which simulates branch and foliage interaction within and between crowns. WHORL2's spatial detail enables tracking individual crown volumes and the vertical and horizontal overlap of neighboring crowns throughout the course of stand development (see Materials). The influence of both the initial pattern of tree spatial distribution and initial variability in tree size (height) on stand dynamics and mortality, and the ability of the self-thinning relationship to reveal these effects, were also investigated.

In order to remove the statistical bias of having N appear in both sides of the self-thinning relationship (Westoby, 1984; Weller, 1987a), the investigation uses the self-thinning relationship (eqn 4.2) rewritten in terms of stand crown volume:

$$\log(\sum_{trees} CrownVolume) = \log(\kappa') + (1+\gamma)\log(N) \text{ (eqn. 4.4).}$$

THE CANOPY COMPETITION MODEL WHORL2

See Chapter 2 for a description of WHORL2. Briefly, each tree is represented modularly as a vertical central axis with horizontal whorls stacked along its length. Each whorl consists of 4 autonomous sectors representing branches. The volume within which stand growth is simulated is partitioned into cubes 10 cm on a side; a branch can support foliage in every cube it intersects. A yearly time-step in the model consists of (1) an acquisition phase when resource capture is determined in each cube of foliage, followed by (2) an allocation phase when the growth of each tree's branches (horizontally) and mainstem (vertically) are determined.

Foliage density within a cube changes as a function of the local light environment. The local light environment changes due to the growth or death of vertically overlapping branches and their foliage, regardless of their tree of origin. If more than one branch intersects a particular cube, the foliage supported in that cube, and the resources it acquires, are evenly divided among the occupying branches. A branch's horizontal growth is a function of the total resource acquired by the foliage it supports, in excess of the maintenance costs of both the individual foliage cubes and the branch itself. A tree's vertical growth is a function of the total resource acquired by the foliage on its branches, in excess of the foliage and branch maintenance costs.

The foliage in a cube dies if it cannot acquire enough resource to meet its maintenance costs. A branch dies if the foliage it supports cannot acquire enough resource to meet the branch's maintenance costs. A tree dies when all of its branches die.

INVESTIGATED PARAMETERIZATIONS

The investigation employed parameterizations from Group 1 of WHORL2's Pareto Optimal Set (Table 2.7). The parameterizations in this group differed only slightly in their settings and generated almost identical stands, as judged by the ten assessment criteria chosen for the assessment (Chapter 2). Six representative parameterizations of these slight variants were selected for use in this investigation (Table 4.1). Varying the settings selected for the Dead Minimum parameter had no detectable effect, therefore only simulations using Dead Minimum set to 4 are shown (Table 4.1).

CANOPY COMPETITION SCENARIOS

Each parameterization was run under 4 different scenarios for the stand's initial spatial and height distribution in a 2x2 combination of: an aggregated (denoted AS) and a uniform (denoted US) spatial distribution (Fig.s 4.1 and 4.2), and a random (RH) and a identical (IH) initial height distribution (Table 4.2). The densities and plot size were chosen for similarity to the permanent plot used in the model assessment (see Sorrensen-Cothern et al, 1993).

Each simulation ran until the tallest tree exceeded 23 m, the stand ceiling defined by the memory restrictions of the program (Table 4.2). All mortality in the simulations results from the competition process. Foliage units can be classified into one of two groups: sun or shade. Foliage units in the same group have the same physiological and morphological parameter settings.

MEASUREMENTS

Crown foliage volume can be measured in WHORL2 as the number of cubes in which the tree supports foliage. As a measure of resource acquisition, this is better than both crown spatial volume and stemwood volume, though it ignores such differences between cubes as foliage density, foliage physiology (sun or shade type), and local resource availability (the quantity of radiation).

The following variables are annually recorded for each live tree: projected crown area, crown foliage volume, height, and the number of cubes jointly occupied with branches of other trees (a measure of crown overlap). Each tree's projected crown area is measured by locating the longest branch, across all whorls, in each of the four sectors making up each whorl. The radii are used to calculate the combined area of these four projected ¼ circles. The stand canopy foliage volume, the total number of .001 m^3 cubes with live foliage, is also recorded.

To avoid double-counting foliage cubes occupied by more than one tree, stand crown volume is estimated as stand canopy foliage volume, the total volume of foliage in the canopy, rather than the sum of the individual tree's foliage volumes.

INVESTIGATION RESULTS

TOTAL PROJECTED CROWN AREA

The assumption that $\overline{Crown\ Area} \propto N^{-1}$ can be equivalently formulated as

$$\sum_{trees} Crown\ Area = constant \text{ (eqn. 4.5).}$$

This relationship was not found for any of the simulations, the sum of the projected crown areas declines in all four scenarios after the onset of DD mortality (Fig. 4.3). The

only exception occurs during years 10 through 40 for the USIH scenario (Fig. 4.3; see Table 4.2 for scenario definition), prior to the onset of DD mortality.

JOINT OCCUPANCY

Joint occupancy of the canopy foliage volume declines from over 60% at year 10 to 5% by year 80 (Fig. 4.4). All scenarios display a monotonic decrease in joint occupancy during this interval, with the exception of the USIH scenario which displays a relatively constant percentage volume (years 10 to 40). While this total percentage remains constant, the distribution of the number of occupants per cell rapidly alters in a contrasting manner compared to the other scenarios (Fig. 4.5). With the exception of USIH, all scenarios show a regular decline in the maximum number of branches jointly occupying a space (Fig. 4.5).

PROJECTED CROWN AREA

The distribution of projected crown areas becomes increasingly right-skewed with time in all simulations, at a rate that decreases with decreasing spatial aggregation in the initial stem distribution (Fig. 4.6). All of the simulations develop bimodal crown area distributions, making the mean a very poor characterization of the stand's crown area distribution (Fig. 4.6).

CROWN FOLIAGE VOLUME

The distribution of crown foliage volume becomes increasingly right-skewed and bimodal with time in all simulations (Fig. 4.7). The USIH scenario has the slowest rate of skewness development (Fig. 4.7).

HEIGHT

The distribution of tree heights becomes increasingly right-skewed and bimodal with time in all simulations (Fig. 4.8). A third mode in the later ASIH and ASRH scenarios reveals a developing intermediate canopy class (Fig. 4.8, year 89). The USIH scenario has the slowest rate of skewness development (Fig. 4.8).

COMPETITION SYMMETRY

Crown foliage volume, projected crown area, and tree height are all three highly associated (Table 4.3). Their increasingly skewed distributions lead to stands in which roughly 80% of the total canopy foliage volume consists of the crowns of the tallest 25% of the stand (Fig. 4.9). The height of the lowest living branches also becomes highly correlated with tree height, the crowns of the taller trees eventually being completely above the tops of the shorter trees (Fig. 4.10, Year 70 and 89).

INFLUENCE OF INITIAL SPATIAL DISTRIBUTION ON SELF-THINNING PROCESS

Development of Stand Canopy

Increasing the initial spatial aggregation leads to the development of a larger canopy foliage volume, (AS = 119% of US), and slightly higher canopy foliage volume growth rates (Fig. 4.11). A uniform initial height distribution slows canopy development or, in conjunction with a uniform spatial distribution, temporarily stops it entirely when the when the crowns come to share space with their neighbors (Fig. 4.11, Years 10 to 40).

Mortality Rate Dynamics

Decreasing the initial spatial aggregation delays the onset of density dependent mortality and drastically alters its temporal dynamics (Fig.s 4.12, 4.13 note for fig. text: Annual mortality rates generally alternate between higher and lower rates; a smoothed, nonparametric regression of this year-to-year variability is displayed, revealing the

general trends). The annual tree mortality rate of the spatially aggregated stands rapidly peak, then decline to cycle around a fairly constant annual rate, in contrast to the spatially uniform stands which cycle around a roughly linearly increasing trend (Fig. 4.13). The assumption of uniform initial heights delays the onset of DD mortality in both spatial scenarios (Fig. 4.13).

Self-Thinning

Decreasing the initial spatial aggregation decreases the steepness of the self-thinning relationship (Fig 4.14, Table 4.4). The change from a more variable to a more uniform initial height distribution tends to delay the onset of mortality, hence the greater stand canopy foliage volume at the onset of self-thinning. This does not affect the asymptotic slope of the final crown foliage volume self-thinning relationship (Fig 4.14).

ADEQUACY OF THE GROWTH ~ MORTALITY RELATIONSHIP

These stand development results contradict each of the assumptions following from the growth ~ mortality relationship $\overline{Crown\ Area} \propto N^{-1}$. However, the relationship's failure extends beyond the limitations of it's mathematical assumptions: it provides no insight into the relationship between stand growth and mortality as it fails to capture the developing inequality in tree resource acquisition which underlies the development of dominance and the transition to a predominantly 1-sided competition process.

CONSTANT SUM OF PROJECTED CROWN AREAS AND CROWN OVERLAP

The sum of the projected crown areas declines rather than remain constant (Fig. 4.3), a result also found by Kajihara (1976). The decline is initially strictly due to the death of lower, longer branches as a result of shading (reduced radiation levels) and horizontal crowding (reduced foliage per branch) (Figs. 4.3, 4.5, 4.6), with tree mortality eventually resulting as branch loss increases (Figs. 4.3, 4.12). Zeide (1985, 1987) and Clark (1990,

1992) incorporate this decline into their models, arguing that increasing gap formation causes canopy closure to decline as the stand develops. However, in the dense stands simulated here, the branch death in Fig. 4.3 results from shading and crowding, and hence produces very few gaps.

The distribution of the number of branches jointly occupying a volume of space (Fig. 4.5) and the percentage of the total canopy foliage volume occupied by more than one branch (Fig. 4.4) both decline with stand development, contradicting the assumption that canopy overlap remains constant (Norberg, 1988; Zeide,1987). Mortality among the lower, longer branches (Fig. 4.5, declining maximum joint occupancy) produces horizontal partitioning of the stand at any given height, as described by Ford and Newbould (1970). The assumption of limited canopy overlap (Clark, 1990, 1992; Zeide, 1987) holds only after competition has caused the reduction in joint occupancy by branch death and the onset of tree mortality (Fig. 4.4, 4.5).

Representativeness of Mean Tree

As frequently stated previously, (Westoby, 1984; Weller, 1987a; Weiner, 1988) the mean tree is not representative of the skewed distributions of projected crown area, crown foliage volume, and height developed in self-thinning stands (Figs. 4.6, 4.7, 4.8). This is especially true as secondary modes form in the distributions (Figs. 4.6, 4.7, 4.8), revealing the development of dominant and suppressed tree classes. These classes result from the continual decline in foliage volume (Figs. 4.7, 4.9) and light environment (Fig. 4.10) of the suppressed trees, reducing their growth rates (Figs. 4.6, 4.7, 4.8, 4.15) (Ford and Newbould, 1970). The decline in foliage volume is a result of both horizontal crowding (Figs. 4.4, 4.5) and increased shading from taller, larger crowned neighboring trees (Table 4.3, Fig. 4.10). Note that bimodalities develop even in the USIH scenario, which was initiated with identical trees (Fig. 4.6, 4.7, 4.8). The bimodalities reveal the decline in

status, and in foliage volume, that a tree undergoes prior to mortality and have been reported for a range of single species stands grown at high density (Ford, 1975).

COMPETITION AS A 2-DIMENSIONAL PACKING PROBLEM

The growth ~ mortality relationship implies that self-thinning is a 2-dimensional packing problem (see Norberg, 1988, pg 221, 223). Zeide (1987, pg. 533) states "the process of self-thinning (in even-aged stands) consists of two components: the creating of gaps due to mortality and the filling of some of these by surviving trees." It follows that competition among branches is predominantly 2-sided and symmetric, that there are no dominants trees shading suppresseds and hence that stands are composed of trees nearly identical in height and crown size, and hence evenly spaced (see Norberg, 1988, pg. 253). If there were dominant and suppressed trees, the mortality would occur among the suppressed trees due to overshading. But this would not free space for crown expansion as the light environment of the vacated volume would not be able to support foliage, hence the death of the suppressed trees. Therefore mortality would not result in further growth. The packing analogy describes the growth of an identical stand of trees as a response to mortality, rather than its cause. Further, in the horizontal packing model, this mortality, by definition, is not density dependent . The USIH scenario revealed that even when conditions might be expected to develop a stand of identical trees, minor variability in resource acquisition produces dominant and suppressed tree classes (Figs. 4.6 - 4.8).

This 2-dimensional perception of competition ignores the vertical dimension whose exploitation marks the development of a dominant class and a transition from predominantly 2-sided competition among branches to 1-sided competition among crowns. An initial horizontal partitioning of the stand (Figs. 4.4, 4.5) is followed by a vertical partitioning as trees become suppressed (Figs. 4.9, 4.10, 4.15). The development of vertical dominance among branches provides a mechanism for growth to induce DD mortality: shading. The timing of this transition, and its rate, depend on the initial

variability in the stand, both spatially and in terms of initial crown size (analogous here to initial height) (Fig. 4.9, 4.10, 4.15).

The transition from 2-sided to 1-sided competition is a conceptual simplification, as these processes can occur simultaneously in the stand, the dominating process differing between local neighbors as a function of the local tree size distribution and spatial distribution. 2-sided and 1-sided competition occur at slightly different scales: 2-sided competition involves branch -to-branch horizontal crowding, though the cumulative effect could be crown-to-crown crowding; 1-sided competition involves shading of a branch by a higher branch or crown, though the cumulative effect could be vertical dominance between crowns. The clearest measure of branch-to-branch interaction is the complement of the percentage of canopy foliage volume jointly occupied (Fig. 4.4): the percentage of canopy foliage volume occupied by a single branch. This reveals the increasing horizontal partitioning produced by 2-sided competition. The clearest measure of vertical dominance is mortality, which results predominantly from shading (Fig. 4.12). Plotting these two measures against each other clearly reveals the rather abrupt transition from 2- to 1- sided competition characteristic of the uniformly spaced stands in contrast to the more gradual shift of the two competition processes due to the variation in local density of the spatially aggregated stands (Fig. 4.16).

The 2-dimensional packing analogy implicitly assumes that crown readjustment and mortality are synchronous and simultaneous. This is directly contradicted by the decline in a tree's foliage volume preceding mortality, (Fig. 4.7), demonstrating the release of volume for reoccupancy prior to death, and more directly by the fact that it is the suppressed trees in the understory which die, and the dominant trees in the overstory which grow: the volume occupied by new growth is not the volume vacated by mortality.

SPATIAL INFLUENCES ON DD MORTALITY AND THE TRANSITION TO 1-SIDED COMPETITION

Increasing the variability in the initial spatial distribution produces lower density neighborhoods in which trees have a larger growing space available before horizontal crown expansion brings them to share space with their neighbors. Their relatively larger crowns (Fig. 4.6, Year 10) provide both greater resource acquisition and larger growth rates (Fig. 4.15) than both their more crowded neighbors and trees in stands with less initial spatial variability. This variability in growth rates hastens the development of skewed stand-wide distributions of height, crown area, and crown volume relative to the uniform stands (Figs. 4.6 - 4.8), resulting in taller trees and greater stand biological volume at any given time (Fig. 4.11). The variability in growth rates lead to a quicker transition from 2-sided to 1-sided competition (Fig. 4.16, ASRH, ASIH scenarios).

Trees in the higher density neighborhoods experience increased horizontal crowding of branches relative to the more uniform stands (Fig. 4.5, Years 1, 5, 10). The intensified 2-sided competition among branches for foliage volume, and increased shading from branch overlap, cause extensive mortality among the lower, longer branches of the clustered trees (Fig. 4.5, Year 5 , 10, 20, decline in maximum joint occupancy in aggregated stands versus uniform stands as longer branches die off). Branch mortality this early in stand development represents the loss of a significant proportion of a tree's foliage volume. A slightly taller or less crowded tree with the advantage of exposure to relatively higher light levels and/or increased foliage volume is less impaired by the loss of a lower shaded branch relative to a shorter, more shaded and crowded neighboring tree. This produces relatively greater height increment growth rates for the taller or less crowded trees (Fig. 4.15), increasing both the shading of suppressed trees and the relative gap in height growth rates. The shortest or most crowded trees come to live in a "less favourable but not greatly changing light environment" (Ford and Newbould, 1970). They either die from the inability of their remaining foliage to capture enough resource in the rapidly reduced light environment, or are able to maintain enough foliage on less shaded

branches to survive but not continue growing (Figs. 4.7, 4.15). The most variable simulations (ASRH, Table 3.2) have both the highest initial mortality rates (Fig. 4.13) and the highest initial growth rates of Total Stand Foliage Volume (Fig. 4.11).

Initial spatial variability allows some trees to exploit the vertical dimension of the stand volume more quickly. Once a tree is taller than its immediate neighbors, an increasing proportion of its crown will shade, but not suffer shading from, its neighbors (Fig. 4.10). The overall effect on stand development is a faster transition from predominantly 2-sided to1-sided competition (Fig. 4.16), achievement of larger crown volumes (due to the reduced local densities) (Figs. 4.6 and 4.7), and greater stand canopy volume (Fig. 4.11).

Conversely, the complete lack of variability in spatial distribution and height, in effect the scenario assumed by the 2-dimensional packing model, prolongs the transition to 1-sided competition (USIH scenario, Figs. 4.3 - 4.12, 4.14, Years 10 - 40, Fig. 4.16). The initially unhindered growth allowed by uniform spacing becomes stagnation as crowns simultaneously begin crowding each other (Figs. 4.4 and 4.5), mutually limiting each others' resource acquisition and vertical growth (Figs. 4.7 and 4.8) by horizontal crowding and shading. Eventually minor differences in the model's branch orientation generate slight differences in shading, resulting in nonuniform foliage and branch death. This slight differential in resource acquisition produces a slight differential in vertical growth rates, which cascades until the stand breaks out of this stagnation phase and stand development continues (Figs. 4.6 - 4.8, 4.11). Variability in initial heights introduces enough inequality into the uniformly spaced simulation to avoid this prolonged stagnation (Fig. 4.11), though the course of stand development from 2- to 1-sided competition is just as abrupt (Fig. 4.16).

SELF-THINNING RELATIONSHIP

The self-thinning relationship mainly reveals differences in stand development after the onset of DD mortality. Yet mortality occurs after the initial transition from predominantly

2- to 1-sided competition and the determination of the suppressed and dominant trees. The relationship gives no insight into the earlier growth processes underlying this determination and transition.

This is especially apparent in the crown foliage volume self-thinning slopes (Table 4.4), which are of similar magnitude to empirical results of the foliage weight ~ density self-thinning slopes reported for *Prunus pensylvanica* and *Abies balsamea* (Mohler et al, 1978). Crown foliage volume declines as a tree becomes suppressed and loses vigor. By the time of death, a tree's contribution to the stand's foliage volume has been declining as a result of both it's own decreasing volume and the increasing volume of the dominant trees (Figs. 4.7, 4.9). The spatially uniform stands, USRH and USIH, have achieved over 60% of their final canopy foliage volume by the onset of mortality, years 15 and 50 respectively (Figs. 4.12, 4.13). The suppressed trees destined to die represent only a very small portion of this volume (Figs. 4.7, 4.9). The change in canopy volume after the onset of DD mortality represents the continued, though limited, expansion of the dominant crowns slightly offset by the continued decline of the already small suppressed crowns.

CONCLUSION

Reanalysis has eroded empirical support for the assumption that all stands of plants 'self-thin' similarly (Zeide, 1985; Weller, 1987a, 1987b), and has suggested that self-thinning may be a species specific phenomenon (Zeide, 1985, 1987; Weller, 1987a; models of Norberg, 1988, and Clark, 1992b). This investigation suggests differences in initial size and spatial distribution affect the competition process as well. Further, the investigation shows that the self-thinning relationship, and the models proposed to explain it, inadequately represent the competition process. A more detailed approach to investigation is required.

The self-thinning relationship (eqn.s. 4.1 or 4.3) is an intriguing empirical phenomenon and perhaps a useful measure of Self-Tolerance (Zeide, 1985), but provides little insight

into stand development processes, and is inadequate for generating theory to explain the self-thinning phenomenon. Whether phrased in terms of a hypothetical mean tree or total canopy foliage volume, the relationship (Fig. 4.14) masks the developing inequality in growth and status among individual trees in the stand (Figs. 4.6 - 4.8), capturing only the aggregate behavior. Yet it is exactly the local variability in resource acquisition that drives stand development through the different phases of competition, generating mortality and self-thinning. It also obscures spatial and temporal differences in stand development (Figs. 4.11, 4.13, 4.14, 4.16).

An investigation of the processes underlying self-thinning should observe the two fundamental temporal dynamics, canopy development and mortality, at the level of the individual tree. Stand canopy development consists of a predominantly 2-sided competition phase followed by a predominantly 1-sided competition phase, though the two competition processes can operate simultaneously (Fig. 4.16). The relative length of each phase, the nature of the transition, and the resulting canopy structure are influenced by the initial variability in resource acquisition among individual trees. This variability is a function of many factors: spatial variability in the initial stand distribution, variability in initial height distribution, microsite heterogeneity, genetic heterogeneity (plasticity), etc. More importantly, the rate of transition between phases depends upon the initial stand structure and reveals the differing dynamics of stand development.

Table 4.1: Investigated Parameterizations. Both settings of D Switch were investigated in combination with the other parameters. See Sorrensen-Cothern et al (1993) for model and original parameter definitions. See Chapter 2 for definition of K1, K2, D1, D2, E1, E2, Switch, Dead Minimum, Dead Slope, and Dead Intercept parameters.

D1	7	D2	9	D Switch	0.5 / 0.9	R	0.20
K1	0.1	K2	0.2	K Switch	0.5	Height Incr.	0.00008
E1	1	K2	1	E Switch	0.5	Sector Incr.	0.0006
Dead Minimum		1/ 4 / 7		Dead Intercept	5	Dead Slope	50

Table 4.2: Simulation Scenarios' Initial Spatial and Height Distributions.

Simulation	Spatial Distribution	Height Distribution	Stand Size (m x m)	Initial Trees	Initial Density (trees/m^2)	Years of Simulation[7]
ASRH	Aggregated[8]	Random[9]	6.0 x 6.0	327	9.08	89
ASIH	Aggregated	Identical[10]	6.0 x 6.0	327	9.08	89
USRH	Uniform[11]	Random	6.12 x 6.24	340	8.90	98
USIH	Uniform	Identical	6.12 x 6.24	340	8.90	110

[7] Length of stand development simulation.

[8] The stem map of the permanent plot used in the model assessment, with the addition of fifty trees added to represent the estimated mortality between stand release and the first measurements 20 years later (Fig. 2). See Sorrensen-Cothern et al (1993) for complete details.

[9] Initial tree heights are randomly assigned from a truncated normal distribution with mean 0.5 m, standard deviation 0.5 m, and limits of 0.35 and 0.65 m.

[10] All trees have an initial height of 0.5 m.

[11] A hexagonal grid of radius 0.36 m, 20 rows x 17 columns (Fig. 1).

Table 4.3: Spearman's Rank Correlation of (Tree Height, Crown Foliage Volume) and (Tree Height, Projected Crown Area).

	ASRH		ASUH		USRH		USUH	
Year	Foliage	Crown	Foliage	Crown	Foliage	Crown	Foliage	Crown
10	0.97	0.88	0.99	0.90	0.99	0.95	0.41	0.67
20	0.97	0.95	0.98	0.96	0.98	0.96	0.73	0.91
40	0.96	0.95	0.95	0.93	0.95	0.92	0.99	0.98
87	0.96	0.96	0.97	0.95	0.96	0.95	0.93	0.92

Table 4.4: Stand Canopy Foliage Volume Self-Thinning Slope using Equation 4.4.

Simulation	Slope of 'Asymptotic' Section[12] $\log(m^3) / \log(trees/m^2)$
ASRH, ASUH	-0.20
USRH, USUH	-0.07

[12] Slopes calculated using Principle Components Analysis as errors occur in both variables (Weller; 1987).

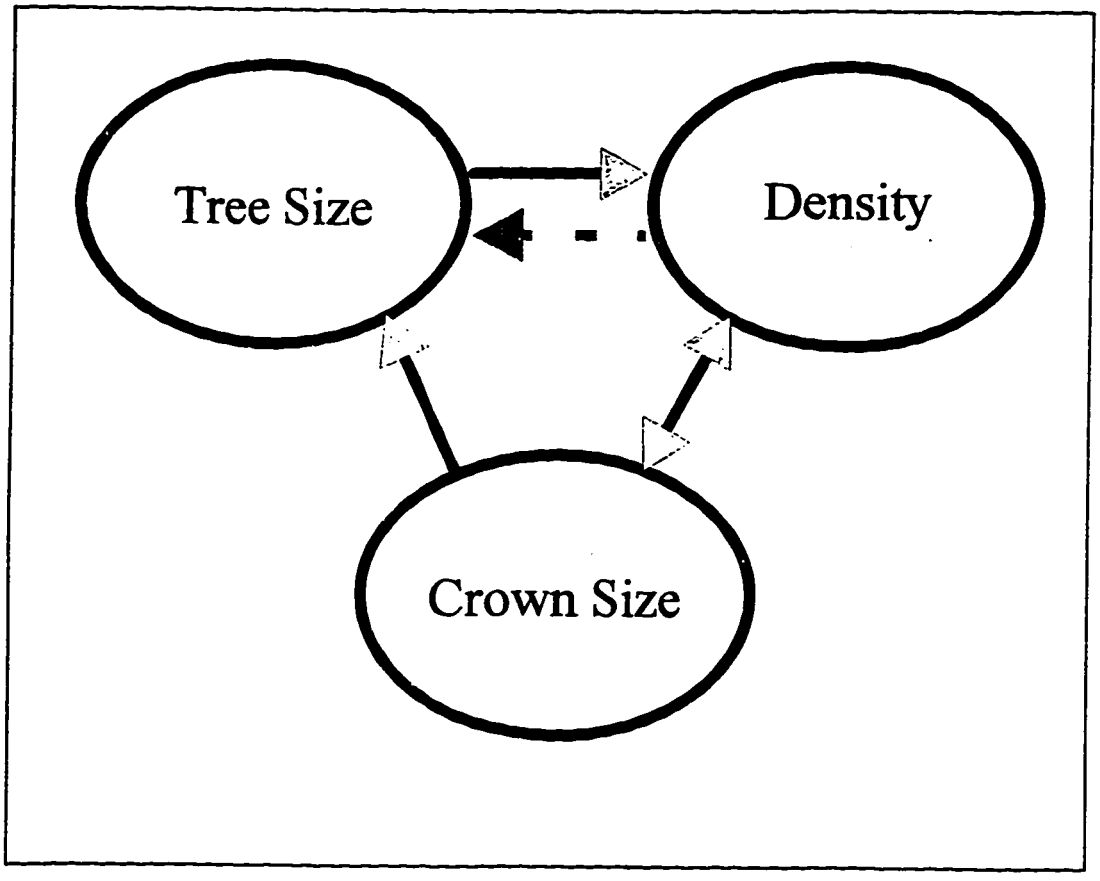


Figure 4. 1: The Canopy Competition Process and it's indirect representation in Self-Thinning Models.

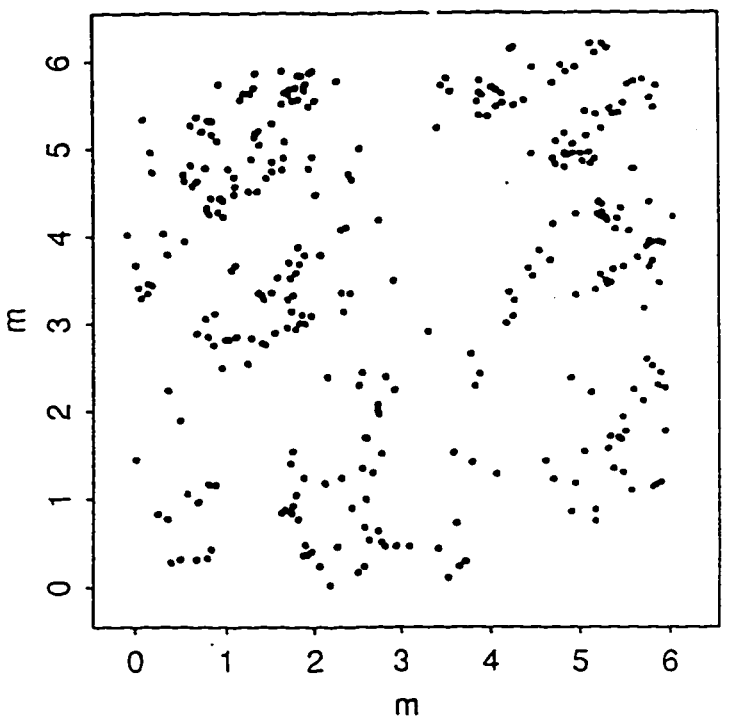


Figure 4. 2: Stem map of Aggregated initial spatial scenario, from permanent plot.

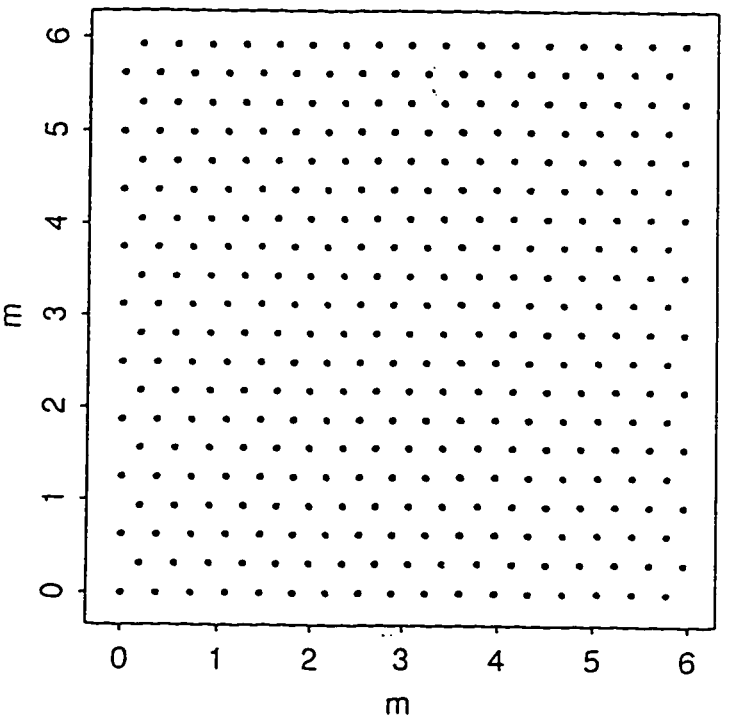


Figure 4. 3: Stem map of Uniform initial spatial scenario. Hexagonal grid of radius 0.36 m.

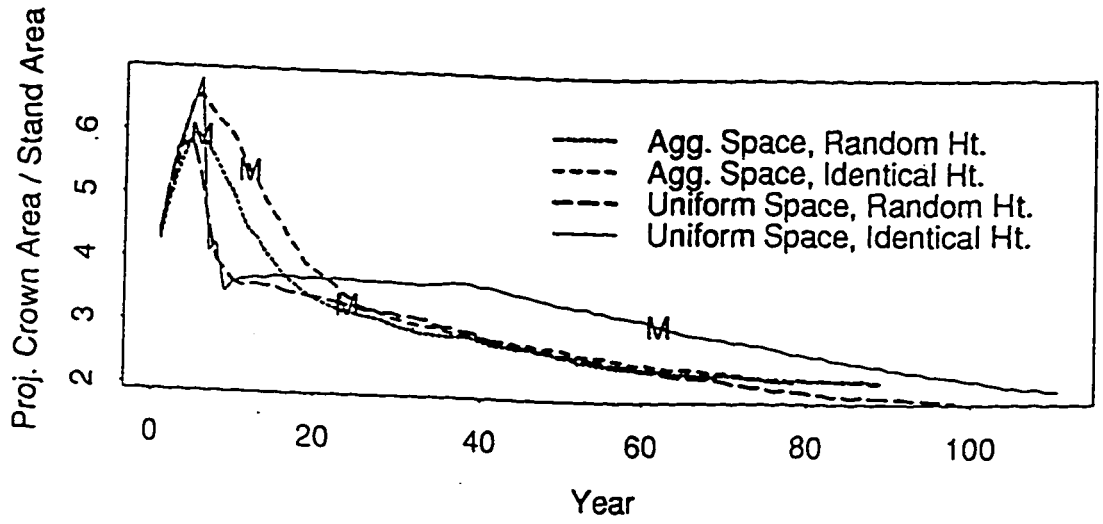


Figure 4. 4: Sum of Projected Crown Areas relative to Stand Area. The line coding scheme described in the legend is maintained throughout the rest of the chapter's figures. M denotes onset of density dependent mortality.

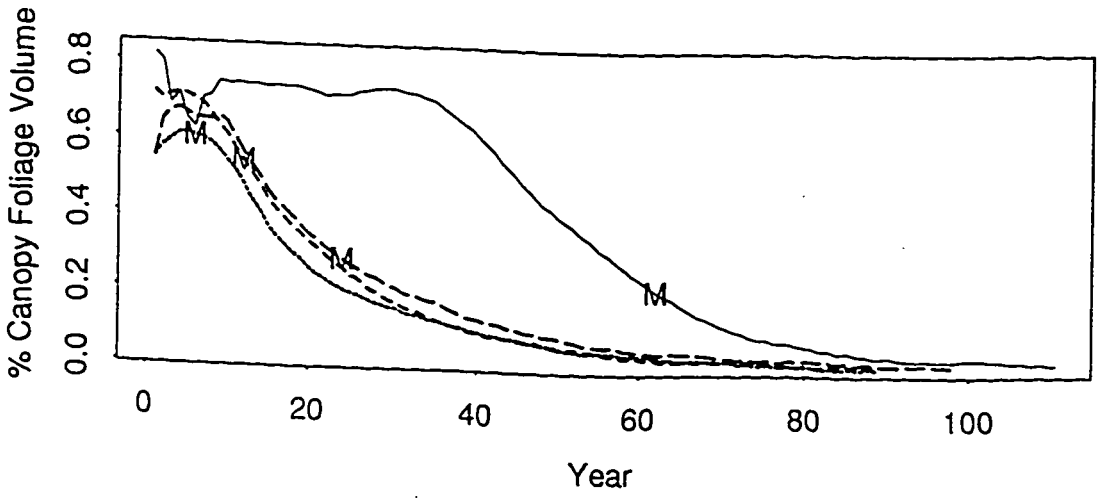


Figure 4. 5: Percentage of canopy foliage volume consisting of cells occupied by more than one branch. M denotes onset of density-dependent mortality.

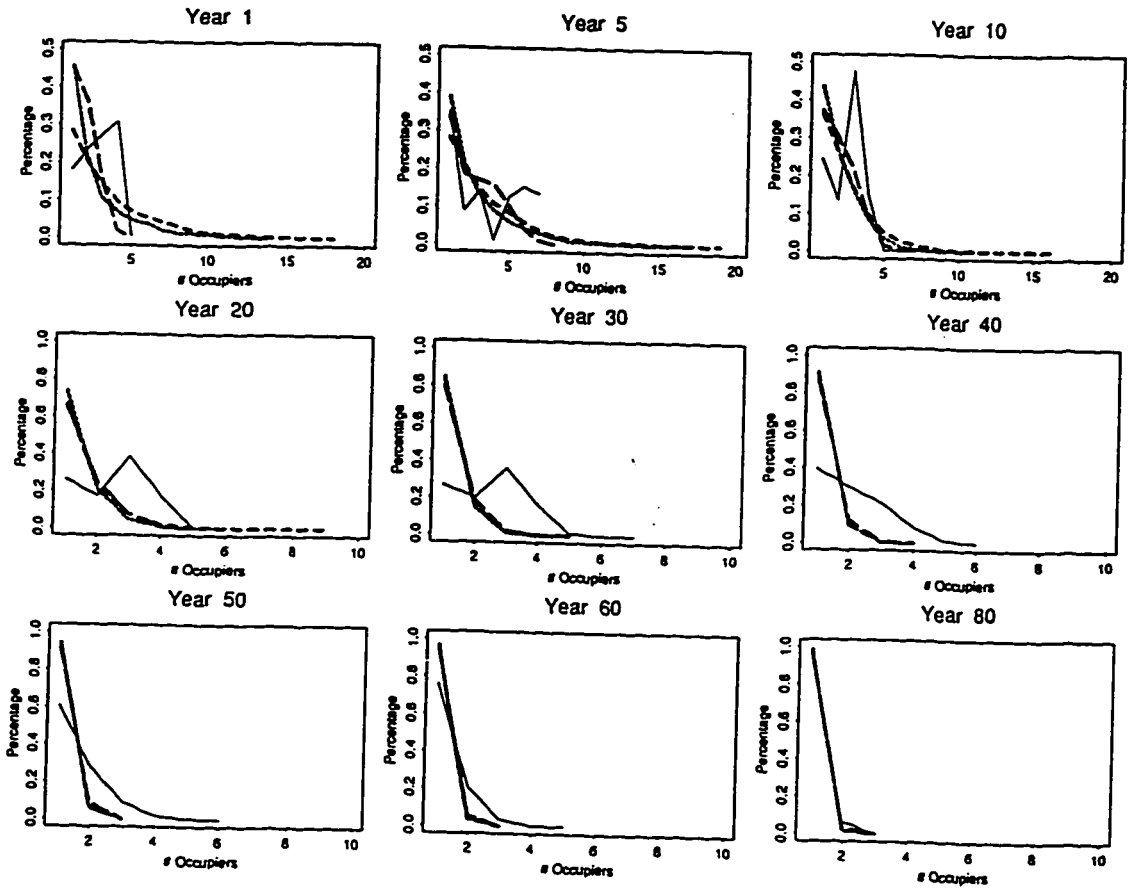


Figure 4. 6: Change in distribution of the joint occupancy of crown volume cells.

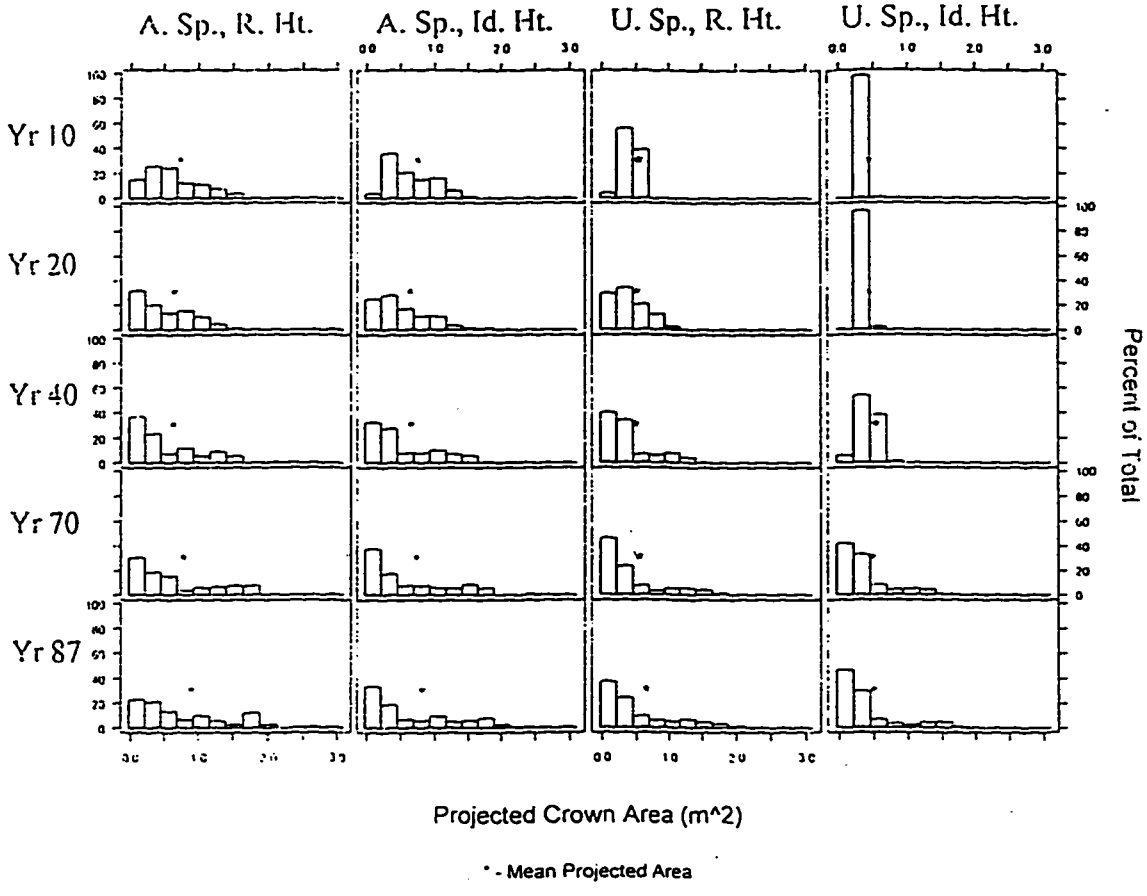


Figure 4. 7: Development of Projected Crown Area distribution through time. * denotes the mean projected crown area.

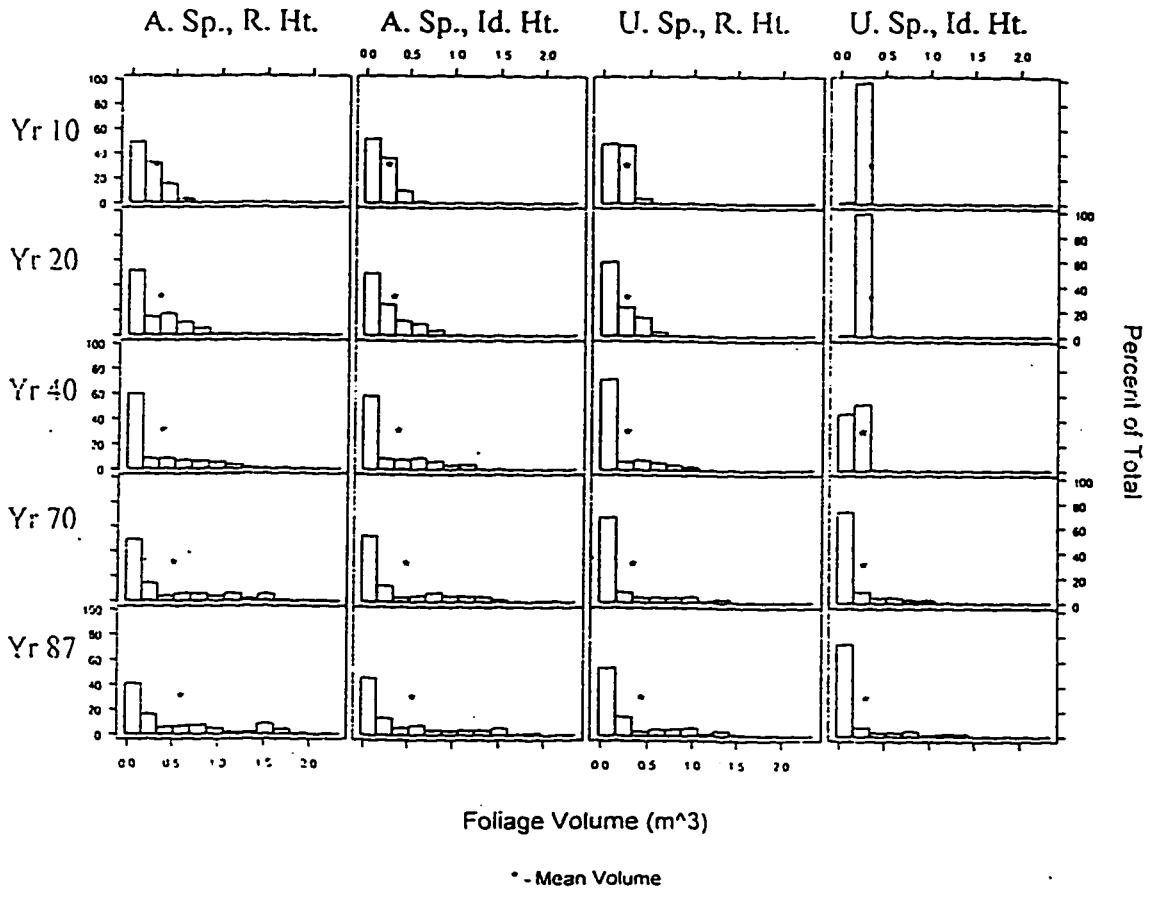


Figure 4. 8: Development of Crown Foliage Volume distribution through time. * denotes mean crown foliage volume. Only live trees are included.

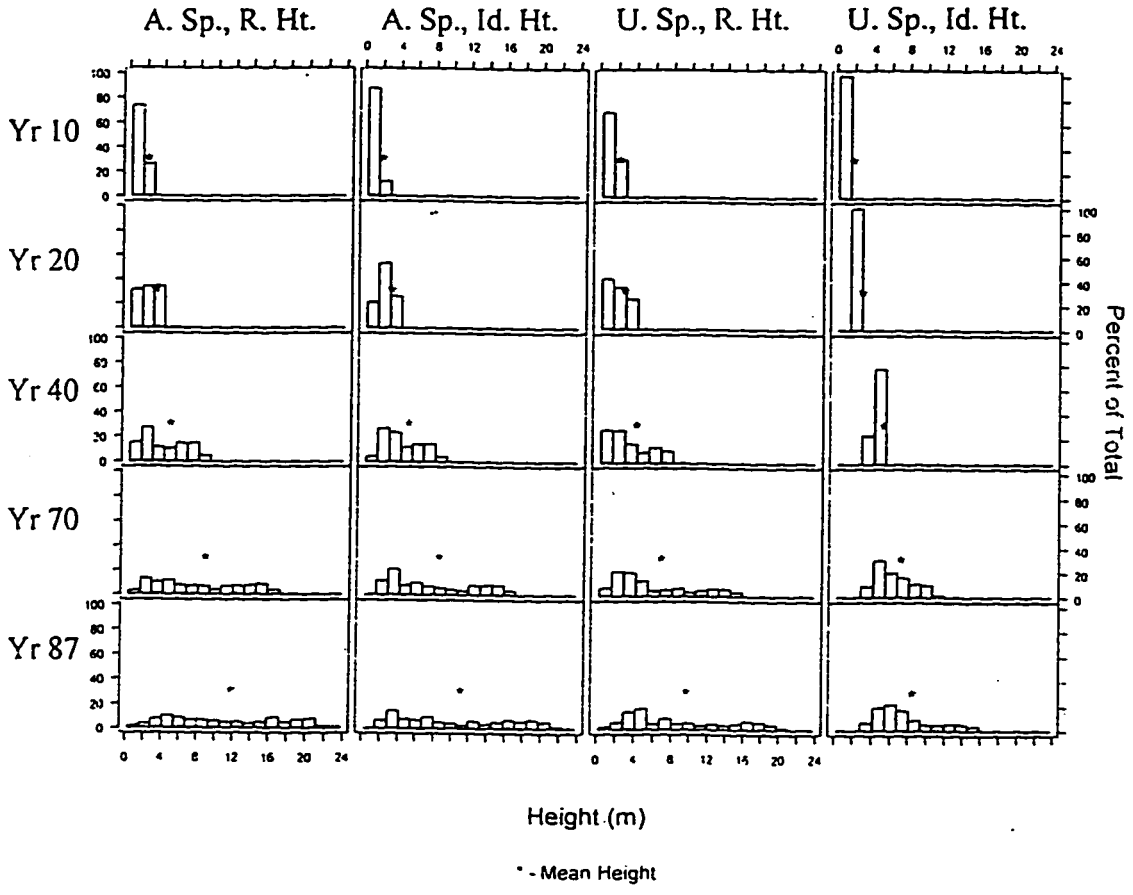


Figure 4. 9: Development of Height distribution through time. * denotes mean live tree height.

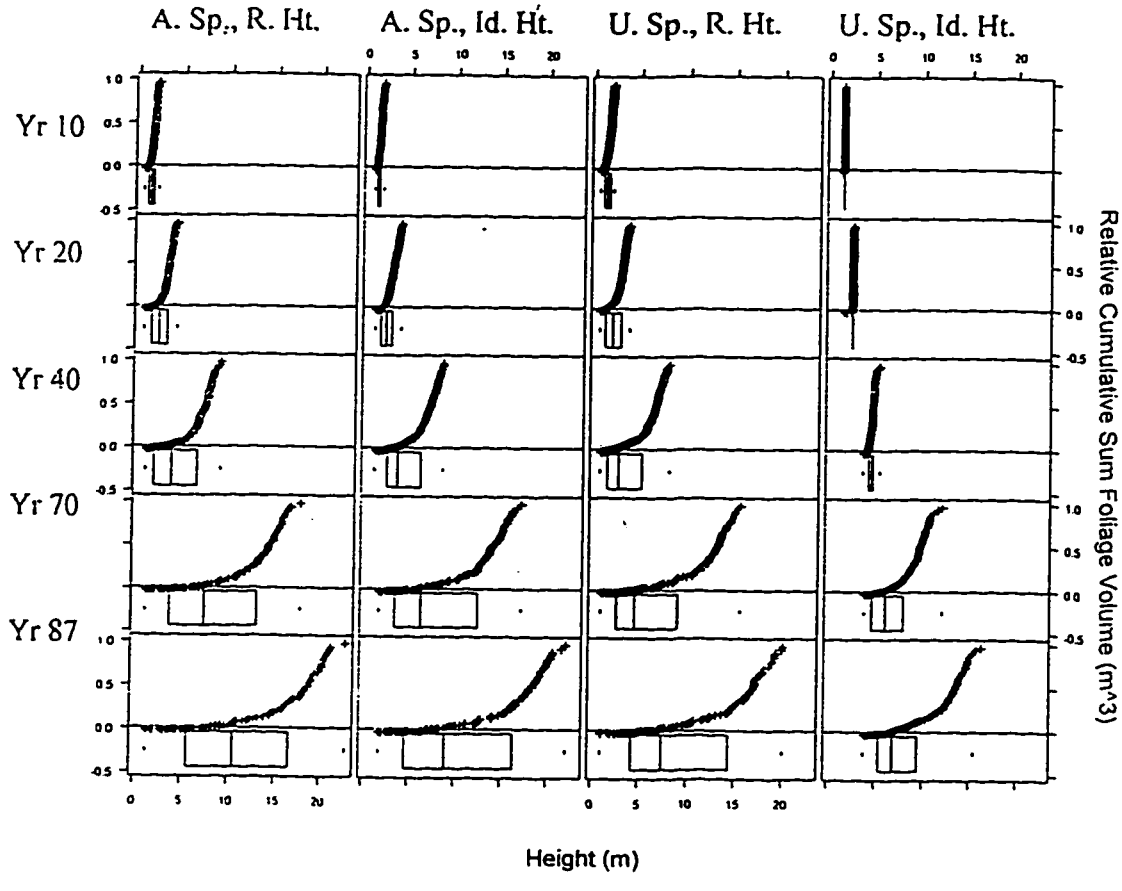


Figure 4. 10: Relative proportion of Total Stand Foliage Volume by Tree Height through time. Bottom horizon of each graph displays a boxplot of the Tree Height distribution. Dotted horizontal lines are 20% increments.

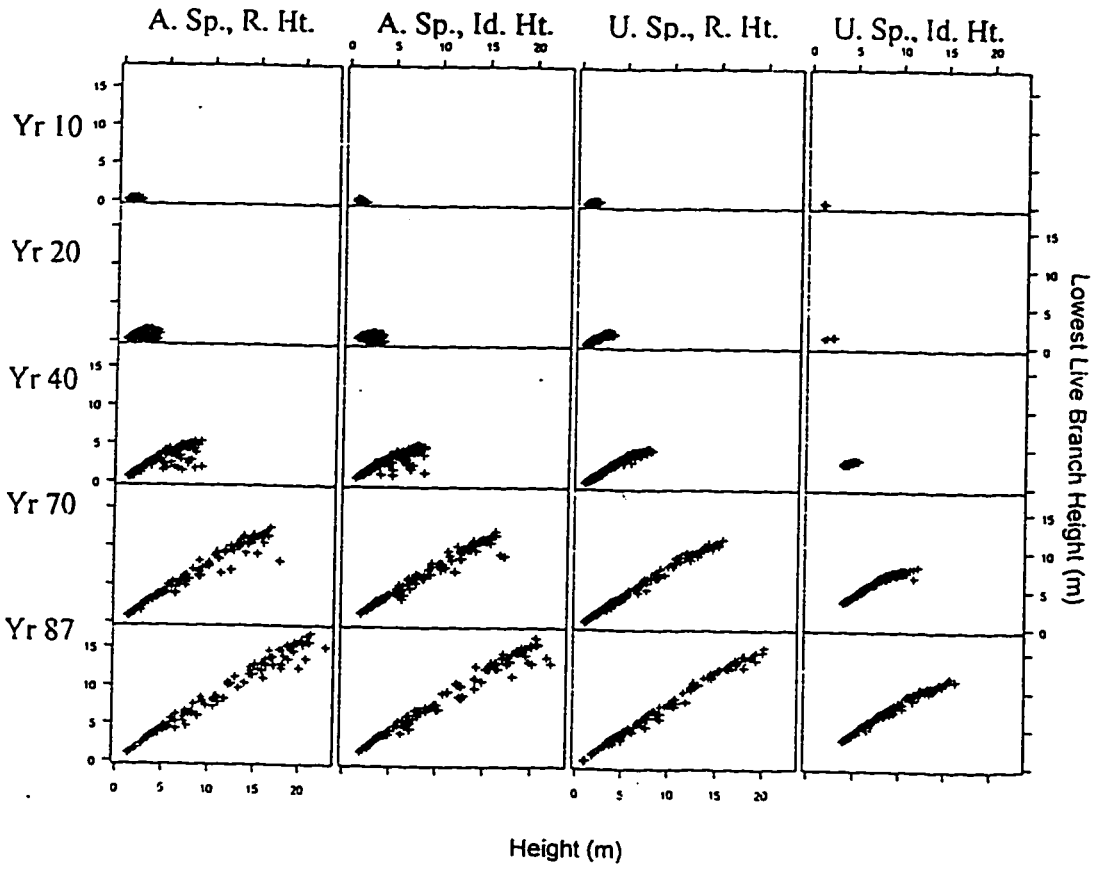


Figure 4. 11: Relationship of height of lowest living branch to tree height, through time.

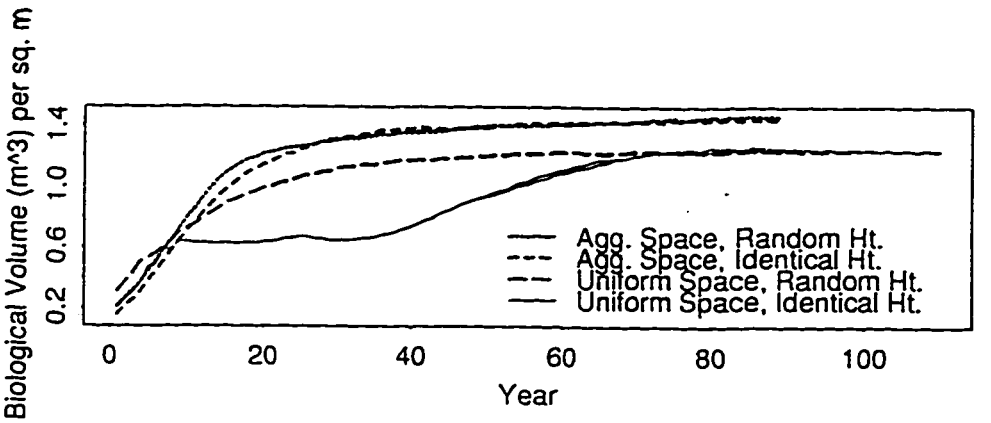


Figure 4. 12: Development of Stand Foliage Volume per unit area.

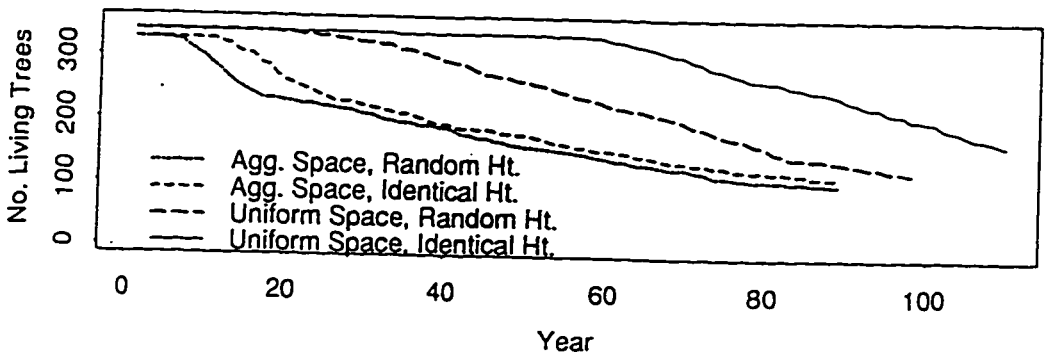


Figure 4. 13: Cumulative Mortality dynamics.

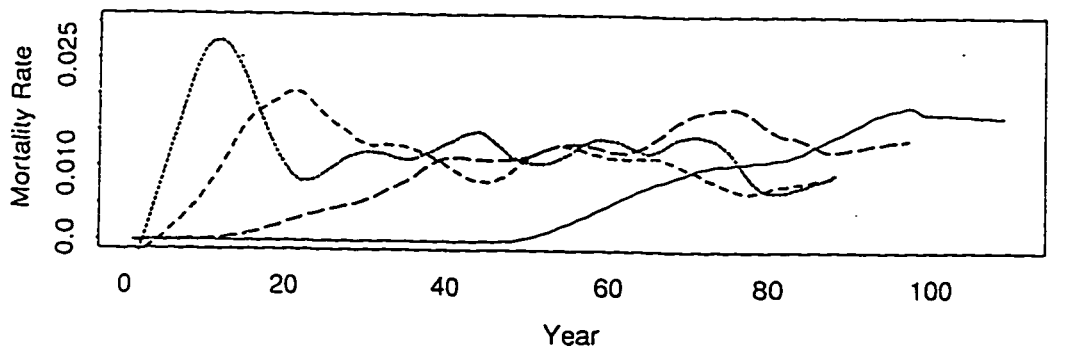


Figure 4. 14: Annual Relative Mortality Rate.

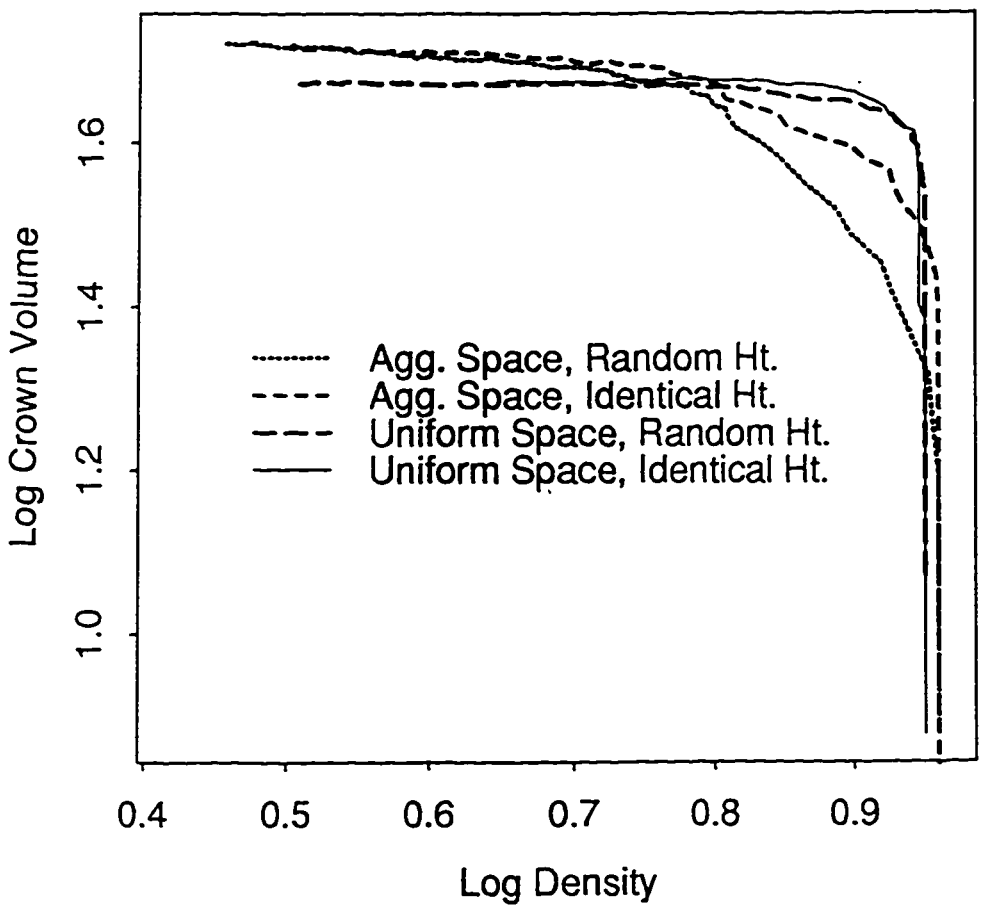


Figure 4. 15: Self-Thinning Relationship of Crown Foliage Volume to Density. See Table 4.4 for estimated self-thinning slopes.

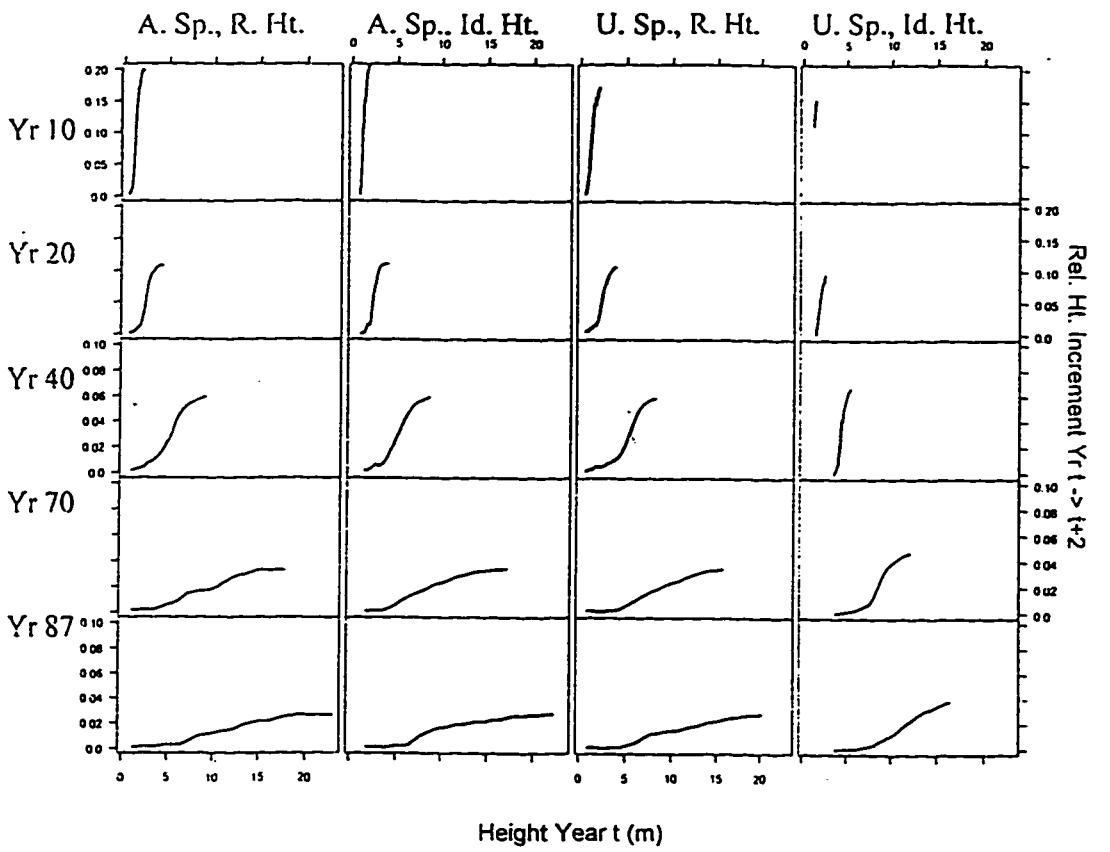


Figure 4. 16: Relationship of Height at year t to Two-year relative Height Increment (Height Increment yr t -> yr t+2 / Height yr t). A smooth of the observations is displayed.

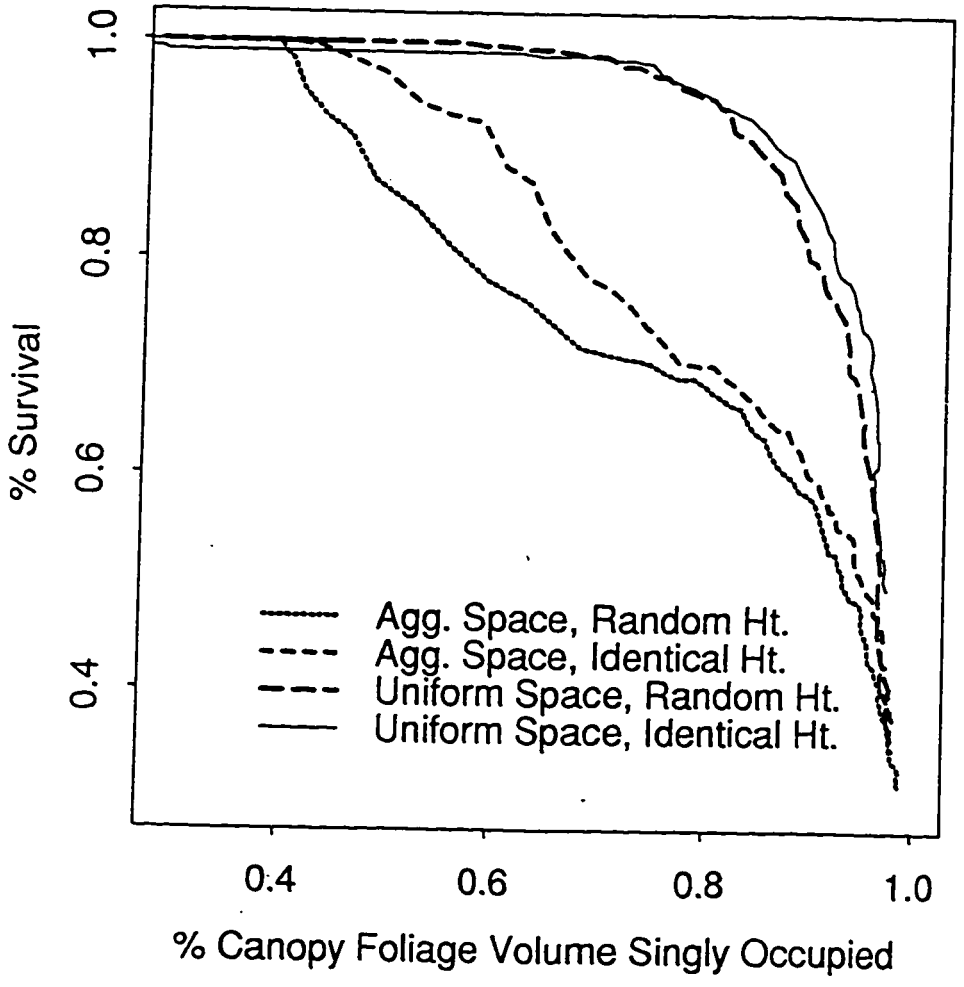


Figure 4. 17: Competition Process Transition. Moving to the right on the horizontal axis represents increasing horizontal partitioning of crowns via 2-sided competition. Moving down the vertical axis represents increasing vertical partitioning of the stand via 1-sided competition.

CHAPTER 5: A NOTE ON DEMONSTRATING THE EFFECTIVENESS OF THE PARETO OPTIMIZATION MODEL ASSESSMENT TECHNIQUE

One suggested test of the proposed Pareto Optimization Model Assessment Technique (Chapter 2) would be to take a model, use it to generate output data sets, then alter the model in both structure and parameter values. The POMA technique would then be applied to the altered model to see if the assessment leads the researcher back to the original model. This should be repeated with a number of data sets to ensure that it is not any particular data set that controls the assessment performance.

While this would certainly provide insight into applying the procedure, it would not be a test of the technique. Rather, it would be a test of the researcher's selection of informative assessment criteria and error bounds.

The POMA technique does not guarantee that known alterations will be detected within a certain number of assessment cycles. Detecting specific deteriorations is a function of both the assessment criteria set and their error bounds. Any given assessment cycle will present one of two results: 1) the current assessment criteria cannot be simultaneously satisfied. Either the assessment criteria or error bounds or model structure need revision, and investigation of the Pareto Optimal Set should reveal which (Stages 3, 5, 5, and 6 in Fig. 2.1). Or 2) All assessment criteria can be simultaneously satisfied. The technique then dictates that the error bounds and assessment criteria should be assessed, as well as model performance (Fig. 2.1). If no errors are revealed on any of these levels, then the alteration has no effect on performance *as captured* by the current selection of assessment criteria. The technique dictates that both the error ranges and the assessment criteria then should be reassessed (Stage 5, Fig. 2.1), revised, and the assessment cycle repeated. The next round may, of course, initiate with further fieldwork as the error ranges and assessment criteria were presumably based on current understanding of the phenomenon

and available data prior to the last assessment cycle. New insights gained during the assessment may suggest new criteria.

The ability to detect deficiencies will always be limited by the current data and knowledge of the system, for this is the basis upon which the criteria and error bounds, the 'gauges' of our assessment, are defined and selected. Poor criteria or error bounds can potentially mislead an assessment, but their effects will be detected in Stages 4 or 5 of the cycle (Fig. 2.1). If a model structure revision increases the deficiencies in the model, these deficiencies will be detected in the next round of assessment if they affect performance as measured by the current assessment criteria. If the revisions create deficiencies undetected by the current assessment criteria, then the researcher is in the scenario mentioned above where all assessment criteria are adequately satisfied simultaneously. At this point the researcher can either select new assessment criteria or error bounds, refining their focus, or accept the provisional result that 'the model satisfies these assessment criteria with these error ranges' and go on to employ the model aware of the limited context within which it has been assessed. The same results would hold with a known target (the suggested test). The only difference is that one would know the limitation of the assessment criteria and error bounds if the "known deficiency" went undetected, and hence would be uncomfortable accepting the provisional result.

The POMA technique makes a co-evolving system of the model structure and the assessment criteria. The 'target test' simply provides a stopping criterion to an otherwise continually reiterating process.

Of course, the assessment process will be halted at some point because either: 1) the model simultaneously accomplishes all that is currently requested of it, and there is no desire to refine the assessment criteria selection and repeat the cycle; or 2) the assessment has led to a model whose revealed limitations are acceptable for the current application.

In either case, the assessment will have provided both an attempt to detect model deficiencies in order to correct them and a process of defining the model structure's 'domain of explanation'. This is sequentially defined during assessment through the construction of the set of assessment criteria and error bounds which the model structure can simultaneously satisfy, and those that cannot be satisfied simultaneously with members of the first set. Refining the domain of explanation, demarcating the context within which the model adequately represents the phenomenon, provides an overarching purpose and guidance to the repeated cycles of model assessment, each of which is internally guided by the search for deficiencies.

BIBLIOGRAPHY

Bäck, T., F. Hoffmeister, and H.P. Schwefel. 1991. A survey of evolution strategies. In Proceedings of the Fourth International Conference on Genetic Algorithms. eds. Belew, R. and L. Booker, pp. 2 - 9. Morgan Kaufmann, Los Altos, CA.

Beck, M. B. 1985. Lake Eutrophication: Identification of Tributary Nutrient Loading and Sediment Resuspension Dynamics. Applied Mathematics and Computation 17: 433 - 58.

Beck, M. B. 1987. Water Quality Modeling: A review of the Analysis of Uncertainty. Water Resources Research 23: 1393-1442.

Beck, M. B., and E. Halfon. 1991. Uncertainty, Identifiability and the Propagation of Prediction Errors: a case study of Lake Ontario. Journal of Forecasting 10: 135-161.

Botsford, L. W. 1992. Individual state structure in population models. In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 213- 236. Chapman, and Hall, New York.

Clark, J. S. 1992a. Relationships among individual plant growth and the dynamics of populations and ecosystems. In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 421 - 454. Chapman, and Hall, New York.

Clark, J. S. 1992b. Density-Independent Mortality, Density Compensation, Gap Formation, and Self-Thinning in Plant Populations. Theor. Pop. Biol. 42: 172 - 198.

Cubasch, U., and R. D. Cess. 1990. Processes and Modelling, Chapter 3 in Climate Change, The IPCC Scientific Assessment. eds. J. T. Houghton, G. J. Jenkins, and J. J. Ephraums, pp 69 - 92. Cambridge Univ. Press, Cambridge.

DeAngelis, D. L. and K. A. Rose. 1992. Which individual-based approach is most appropriate for a given problem? In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 67 - 87. Chapman and Hall, New York.

DeAngelis, D. L. and L. J. Gross, editors. Individual-Based Models and Approaches in Ecology. Chapman and Hall, New York. 1992.

Dingman, J. S., and K. W. Bedford. 1986. Skill tests and parametric statistics for model evaluation. Journal of Hydraulic Engineering 112: 124-141.

Fogel, D. B. 1994. Applying evolutionary programming to selected control problems. Computers Math. Applic. 27(11): 89 - 104.

Ford, E. D. 1975. Competition and stand structure in some even-aged plant monocultures. J. of Ecol. 63: 311 - 333.

Ford, E. D., and P. J. Newbould. 1970. Stand structure and dry weight production through the sweet chestnut (*Castanea sativa* Mill.) coppice cycle. J. of Ecol. 58: 275 - 296.

Ford, E. D., and K. A. Sorrensen. 1992. Theory and models of inter-plant competition as a spatial process. In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 363 - 407. Chapman, and Hall, New York.

Gates, W. L., P. R. Rowntree, and Q-C. Zeng. 1990. Validation of Climate Models. Chapter 4 in Climate Change, The IPCC Scientific Assessment. eds. J. T. Houghton, G. J. Jenkins, and J. J. Ephraums, pp 93 - 130. Cambridge Univ. Press, Cambridge.

Gentil, S., and G. Blake. 1981. Validation of complex ecosystem models. Ecological Modelling 14: 21-38.

Goldberg, D. E. 1989. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, New York.

Gross, L. J., K. A. Rose, E. Rykiel, W. Van Winkle, and E. E. Werner. 1992. Individual-Based Modeling: Summary of a Workshop. In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 511 - 522. Chapman, and Hall, New York.

Halfon, E. 1979. On the parameter structure of a large-scale ecological model. In Contemporary Quantitative Ecology and Related Econometrics. Patil, G. P., and M. L. Rosenzweig, eds, pp. 279 - 93. International Co-operative Publishing House, Fairland, Maryland.

Holland, J. H. 1992. Genetic algorithms. Scientific American, July: 66 -72.

Hornberger, G. M., and R. C. Spear. 1981. An Approach to the preliminary analysis of environmental systems. Jnl. of Environmental Management 12:7 -18.

Hornberger, G.M., and B. J. Cosby. 1985. Selection of parameter values in environmental models using sparse data: a case study. Applied Math. and Computation 17: 335 - 355.

Jaffe, P. R., C. Paniconi, and E. F. Wood. 1987. Model calibration based on random environmental fluctuations. Jnl. of Environmental Engineering 114(5): 1136 - 1145.

Kajihara, M. 1976. Studies on the morphology and dimensions of tree crowns in even-aged stand of Sugi (IV) Crown basal area. J. Jap. For. Soc. 58: 433-440.

Keesman, K., and G. Van Straten. 1989. Identification and prediction propagation of uncertainty in models with bounded noise. Inter. Journal of Control 49(6): 2259 - 69.

Kursawe, F. 1991. A variant of Evolution Strategies for Vector Optimization. in Lecture Notes in Computer Science 496: Parallel Problem Solving from Nature. eds. G. Goos and J. Hartmanis, pp193 - 197. Springer-Verlag, Berlin.

Lonsdale, W. M. 1990. The Self-thinning Rule: Dead or Alive? Ecology 71(4): 1373 - 1388.

Mäkelä, A. 1988. Performance analysis of a process-based stand growth model using Monte Carlo techniques. Scand. J. For. Res. 3: 315 - 331.

Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, Berlin.

Michalewicz, Z., C. Z. Janikow, and J. B. Krawczyk. 1992. A modified genetic algorithm for optimal control problems. Computers Math. Applic. 23(12): 83 - 94.

Mohler, C. L., P. L. Marks, and D. G. Sprugel. 1978. Stand structure and allometry of trees during self-thinning of pure stands. J. of Ecol. 66: 599 - 614.

Mooney, H. A. 1991. Biological response to climate change: an agenda for research. Ecological Applications 1(2): 112 - 117.

Murdoch, W. W., E. McCauley, R. M. Nisbet, S. C. Gurney, and A. M. de Roos. 1992. Individual-based models: combining testability and generality. In Individual-based models and approaches in ecology. eds. D. L. DeAngelis and L.J. Gross, pp. 18 - 35. Chapman, and Hall, New York.

Norberg, R. A. 1988. Theory of growth geometry of plants and self-thinning of plant populations: geometric similarity, elastic similarity, and different growth modes of plant parts. Am. Nat 131: 220 - 256.

Ogino, T. 1977. In Primary Productivity of Japanese Forests, Vol 16. eds. T. Shidei and T. Kira. pp. 169 - 186. University of Tokyo Press, Tokyo.

Olenik, S., and Y. Haimes. 1979. A hierarchical multiobjective framework for water resources planning. IEEE Transactions on Systems, Man, and Cybernetics 9: 534-544.

Oreskes, N., K. Shrader-Frechette, K. Belitz. 1994. Verification, validation, and confirmation of numerical models in the earth sciences. Science 263(Feb. 4): 641 - 646.

Osawa, A. and R. Allen. 1993. Allometric theory explains self-thinning relationships of mountain beech and red pine. Ecology 74(4): 1020 - 1032.

Osawa, A. and S. Sugita. 1989. The Self-thinning Rule: another interpretation of Weller's results. Ecology 70(1): 279 - 283.

Perry, D. A. 1984. A Model of Physiological and Allometric Factors in the Self-thinning Curve. J. Theor. Biol. 106: 383 - 401.

Pickard, W. F. 1983. Three Interpretations of the Self-thinning Rule. Ann. Bot. 51: 749 - 757.

Rinnooy Kan, A. H., G. T. Timmer. 1989. Global Optimization: a survey. in New Methods in Optimization and Their Industrial Uses, ed. Jean-Paul Penot. Birkhauser Verlag, Basel, Germany. pp. 133 - 155.

Rose, K.A., E.P. Smith, R. H. Gardner, A. L. Brenkert, and S. M. Bartell. 1991. Parameter sensitivities, Monte Carlo filtering, and model forecasting under uncertainty. Jrnl. of Forecasting 10: 117 - 134.

Sievanen, R., T. E. Burk, A. R. Ek. 1988. Parameter estimation in a photosynthesis-based growth model. Pages 345 - 352 in A. R. Ek, S. R. Shifley, T. E. Burk, editors. Forest Growth Modelling and Prediction, Vol. 1. USDA Forest Service General Technical Report NC-120.

Sorrensen-Cothern, K. A., E. D. Ford, and D. Sprugel. 1993. A process based model of competition for light incorporating plasticity through modular foliage and crown development. Ecological Monographs 63(3): 277 - 304.

Steuer, T. 1986. Multiple Criteria optimization: theory, computation, and application. Wiley & Sons, New York.

Taylor, Bernard, K. Roscoe Davis, Ronald North. 1975. Approaches to Multiobjective Planning in Water Resources Projects. Water Resources Bulletin. 11(5): 999-1008.

Uhry, Jean-Pierre. 1989. Applications of Simulated Annealing in Operation Research. in New Methods in Optimization and Their Industrial Uses, ed. Jean-Paul Penot. Birkhauser Verlag, Basel, Germany. pp. 191 - 203.

van Straten, G., K. Keesman. 1991. Uncertainty Propagation and Speculation in Projective Forecasts of environmental change: a lake-eutrophication example. Jrnl. of Forecasting 10: 163 - 190.

Vincent, T. L. 1987. Renewable Resource Management. In Applications of Multicriteria Optimization in Engineering and Science. Edited by W. Stadler, pp. 161 - 86. Plenum Press,

Weiner, J. 1988. Variation in the Performance of Individuals in Plant Populations. In Plant Population Ecology. eds. A. J. Dary, M. J. Hutchings, A. R. Watkinson. Oxford University Press, Boston, MA. pg. 59 - 81.

Vincent, T., and W. Grantham. 1981. Optimality in Parametric Systems. Wiley & Sons.

Weller, D. E. 1987a. A re-evaluation of the -3/2 power rule of plant self-thinning. Eco. Mono. 57(1): 23 - 43.

Weller, D. E. 1987b. Self-thinning exponent correlated with allometric measures of plant geometry. Ecology 68(4): 813 - 821.

Weller, D. E. 1989. The Interspecific size-density relationship among crowded plant stands and its implications for the -3/2 power rule of self-thinning. Am. Nat. 133: 20 - 41.

Weller, D. E. 1990. Will the real self-thinning rule please stand up? - A reply to Osawa and Sugita. Ecology 71(3): 2004 - 2007.

Weller, D. E. 1991. The Self-thinning Rule: dead or unsupported? - A reply to Lonsdale. Ecology 72(2): 747 - 750.

Westoby, M. 1984. The Self-thinning Rule. Adv. Eco. Res. 14: 167 - 225.

White, J. 1981. The Allometric Interpretation of the Self-thinning Rule. J. Theor. Biol 89: 475 - 500.

Wigley, T. M., and B. D. Santer. 1988. Validation of general circulation climate models. In Physically-Based Modelling and Simulation of Climate and Climatic Change, pt. 2. M. E. Schlesinger, editor, pp 841 - 882. NATO ASI Series, series C, vol. 243. Kluwer, The Netherlands.

Yoda, K., T. Kira, H. Ogawa, K Hozumi. 1963. Self-thinning in Overcrowded Pure Stands under Cultivated and Natural Conditions. Jrnl of Bio., Osaka City University 14: 107 - 129.

Yokozawa, M. and T. Hara. 1992. A Canopy Photosynthesis Model for the Dynamics of Size Structure and Self-Thinning in Plant Populations. Ann. of Bot. 70: 305 - 316.

Yu, P.L. 1985. Multiple-Criteria Decision Making. Plenum Press, New York.

Zeide, B. 1985. Tolerance and Self-tolerance of trees. For. Eco. and Man. 13: 149 - 166.

Zeide, B. 1987. Analysis of the 3/2 Power Law of Self-thinning. For. Sci. 33(2): 517 - 537.

APPENDIX A: C CODE FOR THE CANOPY COMPETITION MODEL WHORL2

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

WHORL.LOOP.SILENT.C                                             *
*                                                               *
*     INPUT:  simulation parameterizations are read from par.input.test     *
*          Stand spatial data in inCna3565.w                    *
*          Open grown spatial data in onetree.w                 *
*          Observed Tree Heights (yr 30) in obs.ht.stand        *
*               OUTPUT: crit.out - labels & listing of parameterization & criteria    *
*          results for each simulation.
*
*
                                                                *
* SUMMARY OF PROGRAM:                                           *
*                                                               *
*  Read in simulation parameterization, for each parameterization:        *
*   LOOP through Stand & OpenGrown simulations          (clear var, set flags)       *
*    Create whorl & tree  arrays  (based on input tree hts.)             *
*    Define x,y grid by  calculating min & max coordinates               *
*    (Note, offsets allow trees to grow up (-z) out of the plot,         *
*    and to the left (-x) and out the bottom (-y).)                      *
*    WHILE LOOP over years if total_dead not too high                    *
*     call acquisition():                                               *
*      LOOP over each cell                                              *
*       Find potential neighbors of the cell (by each trees maxmum radius)  *
*       LOOP over vertical column of cells (top to bottom)              *
*        Find actual neighbors of a cell and count them.                *
*                  Calculate foliage density of cell via irrad. level & foliage      *
*                   characteristics from last year's irrad. environment           *
*                  Calculate foliage density for each neighbor.                   *
*       LOOP over neighbors: Calculate uptake for each.                *
*       Calculate radiation passing through cell                       *
*     call utilization():                                              *
*     GROW TREES and extend cell boundaries                            *
*     update crown class of trees                                      *
```

```
*  End year LOOP, Stand / Open Grown loop                              *
* CALCULATE simulation results re: chosen criteria (crit_stand, crit_open)    *
* Write out simulation results - crit_out() -> FILE: crit.out          *
* Read in next Parameterization                                       *
**********************************************************************
*  annotated and revised (irradiation error) 5/29/92 by JHR & AA          *
**********************************************************************
*                input process and program structure revise by JHR 4/94
*
**********************************************************************
*                functions placed in separate files, Opengrown & Stand          *
*                simulations made one loop   by JHR 5,6/94                    *
**********************************************************************
*     criteria functions created & added to program, 6-7/94 JHR          *
**********************************************************************
*                Looping errors corrected, whorl.loop finalized  by JHR 8/94        *
**********************************************************************
* Sink output (updating print statements) commented out 12/5/94 by JHR       *
**********************************************************************
* Change physiological plasticity mechanism to foliage level (D, K, E), and  *
* XXXXXXXX level (Dead). Do away with Height Classes.    5/8/95  JHR       *
**********************************************************************/


#include "whorl.auto.h"
#include <string.h>



/* ================== Global variables / parameters ================== */
/*  Parameters that are currently fixed via results of Sorrensen-Cothern et
                al investigations are defined in  whorl.auto.h;
                Decision to fix for current simulations: JHR & EDF 4/94.
                Initiate input dataset & output for Stand Simulation, later change to
                Open Grown Simulation.
                                */


/* Run parameters read from  "par.input.test" */
```

```
char  Dataset[12]="inCna3565.w";                    /* dataset filename variable
*/

double Crlim[2]={0.0,0.0};     /* height limits for crown classes */


              /* Physiological Parameters: 0 - shade setting, 1 - sun setting
                                  2 - break point */
double K[3], D[3];          /* foliage density parameters, */
                    /* see irrad function for definitions */
double E[3];               /* uptake efficiency for each crown class */
double Dead[3];             /* thresholds for whorl relative uptake, by
                          crown class */

double H[5]={0.0,0.0,0.0,0.0,0.0}; /* height growth coefficients */
double W[2]={0.0,0.0};                    /* whorl growth  coefficients */


int   AUTO_TR=0;          /* automatically select sample trees if ==1 */
int   NSAMP=1;                                  /* number of trees to be sampled */
int   Sam_tr[1]={0};           /* sample tree id's */
double Sam_ht[1]={0.5};                   /* sampled heights, filled if  AUTO_TR==1
*/

int   Pol[2]={0,0};                       /* Pollution effects switch */
                                                /*    Pol[0]=1    =>    Shift
increased by 25% */
                    /* Pol[1]=1 => hinc = hinc/4 */

     /* raw output file switches  0==Nope, 1==Yep */
int   FOLIAGE=0;             /* sample tree - each cell - each year */
                             /* need to rewrite FOlIAGE - changes in cell_mid, etc.
                             5/8/95*/
int   TOTFOL=0;             /* all trees - every year */
int   TREE=0;              /* sample tree - uptake, foliage, area */
int   SAMP=0;              /* mean for each z level */
int   HEIGHT=1;             /* each tree */
int   HINC=1;              /* each tree - last two years */
int   DEATH_T=0;             /* # trees - every year */
int   RAD=0;               /* sample tree - sector radii */
int   GRID=0;              /* grid for plot - initialization */
                             /* need to rewrite GRID - changes in cell_mid, etc. 5/95*/
```

```
int             i, j;                                   /* for sort_struct */
struct trees2  temp;



/*************************************************************************
********
*                                                 *
*                       MAIN                          *
*                                                 *
*************************************************************************
*******/
void main()
{
 FILE *ip,*fp,*ffp,*tffp,*tfp,*sfp,*afp;
 FILE *htfp,*hifp,*dtfp,*rfp,*gfp,*obp;

 char htfile[12], hincfile[14];
                        /* ---------------------------- */
 int isub;                 /* loop indices: isub-subject trees */
 int  iyear;                 /*  iyear - year index */
 int once = 1;               /* for AutoSam */
                        /* ---------------------------- */
 int   Nsub;                /* number of trees (input) */
                                                 /* ------------------------
- */
 NBRPTR  nbr;
 SHAREPTR share;

 struct whorls whorl[NT];
 struct trees tree[NT];

 double maxrad[NT];            /* maximum whorl radius of each tree */
 double hmax;                /* max. ht. attained by any tree    */


 double foliage[NZ][NT][NQUAD];        /* foliage per qd. per wh. per tree */
 double uptake[NZ][NT][NQUAD];                  /* uptake  per qd. per wh. per
tree */
```

```
 int   no_cells[NZ][NT][NQUAD];          /* no cells occupied per qd. per wh. per
tree*/
 double reallen[NT];
                          /* ------------------------------ */
 int   NCx, NCy, nslice;          /* grid ranges,  x,y,z dimensions */
 int   NCxmax, NCymax;                    /* maximum dimensions encountered,
                                                    does     cell_irrad   memory
needs reallocating? */
 double xmin, ymin;                                 /* minimum coordinates */

 int             cell_array_set=0;
 double ***cell_irrad;                              /* starting ptr for cell_irrad
array*/
                              /* so large must be dynamically allocated */

                          /* ------------------------------ */
 struct cell_sample sample1[NZ];     /* see cell_sample definition */
 int             i1= -1;
 int   tag;
 int             stand;
                                                    /* ------------------------------ */
 double relup[NT];                          /* relative uptake*/
                          /* ------------------------------ */
 double total_laf= 0.0;                     /* total foliage of all trees */
 double total_ht=0.0;          /* total height of stand */
 int   total_dead = 0;         /* total number dead */

 struct results sim_crit;                   /* structure of criteria results */

 double obs_hts[NT];                                /* array of observed heights,
used
                                                                    in   criteria
calculations */
 int   Osub;
 int             loop_flag=0, count;    /*count- # simulations calculated */


/*****************************************************************
*                    Initialization                 *
*****************************************************************/
/* ------------------------------------------------------------ */
```

```
  setbuf(stdout,0);
/*  puts("START");
  system("date"); */

  nbr   = (NBRPTR) malloc ( sizeof (NBR));
  share = (SHAREPTR) malloc ( sizeof (SHARE));
  NCxmax = NCymax = 0;


/* ------------------------- scan parameters ---------------------- */

  /* Input (K, D, E, Dead, H, W); */ /*Defunct if reading (multiple)
                                                    parameterizations from
'par.input.test' */

  ip=fopen("par.input.test","r"); /* open pointer to input file */
  count=0;                                  /* initialize simulation counter */

  fscanf(ip,  "%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s");
   /* throw out headers */


  while (fscanf(ip,"%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf \
                              %lf %lf", &K[0], &K[1], &K[2], &D[0], &D[1], &D[2],
                              &E[0], &E[1], &E[2], &Dead[0], &Dead[1], &Dead[2],
                              &H[1], &W[1]) == 14)
  { /* while there is input for a new simulation */
   ++count;                       /* update simulation count */

/*--------- Loop through body, first simulating Stand, then the Opengrown----*/
/*--------- simulation. Set 'stand' flag, triggers change of data set, etc.--*/
/*--------- after stand simulation.-------------------------------------*/


for (stand = 1; stand <= 2; stand++)
  { /* loop through model simulation for stand */

   /* clear  structures for each loop */
   bzero((char *) nbr, sizeof (NBR));
   bzero((char *) share, sizeof (SHARE));
   bzero((char *) whorl, sizeof (whorl));
   bzero((char *) tree, sizeof (tree));
```

```
bzero((char *) foliage, NZ*NT*NQUAD*sizeof (double));
bzero((char *) uptake, NZ*NT*NQUAD*sizeof (double));
bzero((char *) no_cells, NZ*NT*NQUAD*sizeof (int));

/* reset counters & flags */
once=1;
total_dead=0;
hmax = 0;

            /* if you want the raw data files written out, set the commented
              flags below to 1 / 0 as noted */
if (stand==1) {
            strcpy(Dataset,"inCna3565.w");
            HEIGHT = 0; /* if want raw, set = 1 */
            sprintf(htfile,"%s%d","stand.ht.", count);
            HINC = 0;   /* if want raw, set = 1 */
            sprintf(hincfile,"%s%d","stand.hinc.",count);
            TOTFOL=0;
  RAD=0;
            }

if (stand==2) {
            /* beginning OpenGrown loop, change variables */
            strcpy(Dataset,"onetree.w");
            sprintf(htfile,"%s%d","open.ht.", count);
            AUTO_TR=0;
            TOTFOL=0; /* if want raw output file - make '1' */
            HINC=0;
            RAD=0;    /* if want raw output file - make '1' */
 } /* end if stand==2 section */


/* ---------------------- open raw output files ---------------------- */
/* Can ignore this section if no raw output files are going to be written */
/* NOTE: if raw output files are to be written out - need to add appropriate
  flags and lines to print DIFFERENT output files for each simulation */

 if (FOLIAGE && stand ==1)  ffp = fopen ("out.fol.stand", "w");
            else ffp = 0;
 if (FOLIAGE && stand ==2)  ffp = fopen ("out.fol.open", "w");
            else ffp=0;
 if (TOTFOL && loop_flag==0)  tffp = fopen ("out.tfl.1", "w");
```

```
                else tffp = 0;
  if (TOTFOL && loop_flag==1)  tffp = fopen ("out.tfl.2", "w");
                else tffp = 0;
  if (TREE)    tfp = fopen ("out.tr", "w");
                else tfp = 0;
  if (SAMP)    sfp = fopen ("out.samp", "w");
                else sfp = 0;
  if (HEIGHT)  htfp = fopen (htfile, "w");
                else htfp = 0;
  if (HINC )   hifp = fopen (hincfile, "w");
                else hifp = 0;
  if (DEATH_T) dtfp = fopen ("out.dt", "w");
                else dtfp =0;
  if (RAD && stand ==1 && loop_flag==0)    rfp = fopen ("out.rad.stand.1", "w");
                else rfp = 0;
  if (RAD && stand ==1 && loop_flag==1)    rfp = fopen ("out.rad.stand.2", "w");
                else rfp = 0;
  if (RAD && stand ==2 && loop_flag==0)    rfp = fopen ("out.rad.open.1", "w");
                else rfp = 0;
  if (RAD && stand ==2 && loop_flag==1)    rfp = fopen ("out.rad.open.2", "w");
                else rfp =0;
  if (AUTO_TR && stand ==1)  afp = fopen ("out.auto.stand", "w");
                else afp = 0;
  if (AUTO_TR && stand ==2)  afp = fopen ("out.auto.open", "w");
                else afp =0;
  if (GRID && stand ==1)     gfp = fopen ("out.grid.stand", "w");
                else gfp = 0;
  if (GRID && stand ==2)     gfp = fopen ("out.grid.open", "w");
                else gfp = 0;

  /* puts("opened output files"); */


/* ---------------------------- scan input ---------------------------- */
/* read in input data re: spatial location of trees in stand, and initial hts */
  fp = fopen(Dataset, "r");
  get_trees (fp, tree, &Nsub, &hmax, reallen); /* read in initial tree data */
  fclose(fp);
  /* puts("scanned input, got trees"); */

/* ------------------------- initialize crowns ------------------------- */
```

```
  nslice = round(hmax/DZ);                    /* number of slices */

  make_crowns (whorl,  tree, Nsub, maxrad, reallen);
  /*puts("crowns initialized"); */

 /* ---------------------- assign x-y grid ---------------------- */


                /* calculate grid dimensions from orig. tree coords. */
  grid (&xmin,&ymin,tree,&NCx,&NCy,Nsub,stand);

  if ((cell_array_set == 0) || (NCxmax < NCx) || (NCymax < NCy))
                {
      if (cell_array_set != 0) /* need more memory */
        {
         free_cell_irrad(cell_irrad, NCxmax, NCymax);
        }
                NCxmax = NCx;
      NCymax = NCy;
      /* calloc cell_irrad array */
                init_cell_irrad(&cell_irrad, NCx, NCy);
                cell_array_set =1;
               }

  /* if (GRID)   NEEDS TO BE REWRITTEN - no cellmid 5-95*/
  /*  OutGrid (gfp, cellmid, NCx, NCy); */

  /*puts("grid assigned");*/


/* ------------------------------------------------------------ */
/* commented out of sink file, remove comments if debugging */

/*  printf("\nNsub=%d (NT=%d), nslice=%d (NZ=%d)\n", Nsub, NT, nslice, NZ);
  puts("----------");
  printf("x:  %lf %lf\n", xmin, xmin + CW/2 + (NCx -1)*CW);/*x-range XX
  printf("y:  %lf %lf\n", ymin, ymin + CW/2 + (NCy-1)*CW);/*y-range XX
  printf("NCx= %d   NCy= %d  NC= %d\n", NCx, NCy, NC);
  */


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

```
*                        main calculation                        *
**********************************************************************/

/****************************** resource acquisition *****************/

/* ---------------------------- year loop ------------------------- */


 iyear = 1;
 while ((iyear <= NYEAR) && (total_dead < 180))
 {
  /*printf ("\n\nyear %d:\n\n", iyear);*/


/* ------------------------- acquisition  loop ------------------------ */

   acquisition (xmin, ymin, cell_irrad, NCx, NCy, &tag, &i1, tree, Nsub,
                                nbr, maxrad, nslice, whorl, share, &total_laf,
                                iyear, foliage, uptake, no_cells, sample1, SAMP,
                                FOLIAGE, Sam_tr, ffp);

/*------------------------- resource utilization loop ----------------*/

   utilization (Nsub, iyear, &nslice, tree, whorl,  foliage, uptake, no_cells,
                                relup, maxrad, &total_dead, &total_ht, TREE, Sam_tr,
                                 NSAMP, HINC,  tfp, hifp);


/* ---------------------------- output ---------------------------- */

                        /* Year before last year select sample trees */

  if ((AUTO_TR == 1) && ((iyear + 1) == NYEAR) && (once == 1))
  {
   /*printf("\nAutomatic sample tree selection at year %d \n",iyear);*/
   AutoSam(afp, Nsub, tree, &once);
  }

 /* RAW DATA OUTPUT FILES WRITTEN */

  if ((AUTO_TR == 1) && (iyear == NYEAR))  /* Last year redefine any sample*/
    CheckLive(afp, Nsub, tree);            /* replace dead trees in sample */
```

```
  if ((HEIGHT) && (iyear % SAM_INT == 0))          /* every SAM_INT years */
   OutHt (htfp, tree, Nsub);

  if (TOTFOL)
   OutTfl (tffp, iyear, total_laf, total_dead, total_ht, nslice, Nsub);

  if (DEATH_T)
   OutDead (dtfp, relup, iyear, Nsub);

  if ((SAMP) && (iyear == NYEAR))
   OutSamp (sfp, i1, sample1, nslice);

      /* for out.rad only @ last year: if Rad && iyear==NYEAR */
  if ((RAD) && (iyear % SAM_INT ==0))          /* Every SAM_INT years */
   OutRad (rfp, whorl, Sam_tr, iyear, nslice, NSAMP);



/* ---------- reset variables at end of each year ---------- */

  total_laf = total_ht = 0.0;

  i1   = -1;
  tag  = 0;
  iyear += 1;

/*printf("mort: %d \n",total_dead);*/
 } /* END YEAR LOOP ---------------------------------------- */


 /* function to close raw data files */

 if (GRID)   fclose (gfp);
 if (RAD)    fclose (rfp);
 if (DEATH_T) fclose (dtfp);
 if (HEIGHT)  fclose (htfp);
 if (HINC)   fclose (hifp);
 if (SAMP)    fclose (sfp);
 if (TREE)   fclose (tfp);
 if (TOTFOL)  fclose (tffp);
```

```
if (FOLIAGE) fclose (ffp);
if (AUTO_TR) fclose (afp);

if (total_dead < 180) { /* completed simulation successfully*/
 if ((stand == 1) && (loop_flag == 0))  /* first simulation */
   {
                obp = fopen("obs.ht.stand","r");  /* read in nonzero obs. hts */
                fscanf(obp, "%d", &Osub);

                for (isub=0; isub < Osub; isub++)
                        fscanf(obp, "%lf", &obs_hts[isub+1]);
     fclose(obp);

                qsort(obs_hts+1, Osub+1, sizeof(double), compare);
   } /* end reading in observed hts */

 if (stand == 1) { /* calculate stand criteria results */
  crit_stand(total_dead, tree, Nsub, obs_hts, Osub, &sim_crit);
  }
 else          /* Open Grown simulation */
   { crit_open(whorl[0], tree[0], nslice, &sim_crit);
                /* write out simulation criteria results */
                criteria_out(sim_crit, loop_flag, K, D, E, Dead, H, W);
   }
}
else /* aborted simulation - mortality too high */
{  /* set sim_crit flags */
 sim_crit.mortality= total_dead;
 sim_crit.median = -1.0;
 sim_crit.ks_prob = -1.0;
 sim_crit.ks_stat = -1.0;
 sim_crit.sup_slope = -1.0;
 sim_crit.sup_r2 = -1.0;
 sim_crit.dom_slope = -1.0;
 sim_crit.dom_r2 = -1.0;
 sim_crit.whorlnum = -1;
 sim_crit.angle = -1.0;
 sim_crit.cr_ratio = -1.0;

 stand = 3; /* so don't do open grown simulation */
 }
```

```
 /* puts("END");
 system("date"); */
 } /* END of SIMULATION LOOP (Stand, OpenGrown) */


if (loop_flag == 0 ) loop_flag = 1; /* change after first sim. */


/* zero out criteria results for next sim. ~ parameterization  */
bzero ((char *) &sim_crit, sizeof(sim_crit));



} /* end of input (parameterization) loop */
fclose(ip);

}                                   /*                  END                  MAIN
********************************************************************** */


/* functions defined in setup.c, acquisition.c, utilization.c, misc.c, deque.c,
                output.c, input.c, crit_stand.c, crit_open.c crit_out.c */
```

```
/*************************************************************************
**************************************************************************
        • file:

WHORL.AUTO.H                                        *
*       last revision 5/8/95 by JHR                      *
**************************************************************************
**************************************************************************/

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



/* ------ define constants ---------------------------------------- */

#define NQUAD    4      /* Max number of sectors in a whorl */
#define CW    0.1       /* X-Y cell width (NOTE change NC accordingly) */
#define DZ    0.1       /* vertical increment */


/* ------------------- Array limits ------------------------------- */

                /* NT*NZ <= 82,500, else need to use dynamic memory alloc. */
#define NT 350         /* Max no. of subject trees (initializes uptake) */
#define NNBR 100       /* Max no. of potential nnbrs to a cell */
#define NNSR 100       /* Max no. of actual nnbrs to a cell */
#define NC 70         /* Max no. of cells per horizontal dimension */
#define NZ 235         /* Max no. of vertical slices */
#define NS 30                  /* Max no. of trees to sample */
#define Nc 10         /* No cells to average in a ray */



/* ------ define types ------------------------------------------- */

typedef struct nbors NBR, *NBRPTR;  /* trees which potentially overlap cell*/
typedef struct sharers SHARE, *SHAREPTR; /* trees which do overlap cell */


/* -------define Parameters currently fixed via results of Sorrensen-Cothern
```

```
et al investigations. Decision of JHR & EDF 4/94.---------- */

#define NYEAR 98            /* number of years simulated */
#define SAM_INT 2           /* sample interval during NYEAR years */
#define PLAST 1             /* 1 == plastic growth option */
#define SWITCH 2.0          /* set ht. classes when 5th tallest tree > this ht */

#define WGROW_TYPE 1        /* Type of whorl rad. / sector rad. growth */
#define CROWN_TYPE 6        /* Index for method of calc. crown length see
                                                    crown_type definitions */
#define LLB 1.0             /* lowest living branch height (m) */

#define NLH 0               /* 1 == nonlinear height increment */

#define R 0.20              /* max. possible irrad. interception rate */


/*----- Crown Type Aliases: Crown type determines crown length and shape---*/

#define REAL   1    /* real lengths of crowns are read in  */
#define FUNCT  2    /* call cl_len function to determine length */
#define LONG   3    /* cl = height - lowest living branch ht */
#define SIM1   4    /* cl = 3/4 height */
 /* Crown types for estimating simulated (small) trees */
   /* p - depth from top of crown */
#define METER  5    /* cl = .7 height; rad = a1*sqrt(p) */
#define HALFM  6    /* cl = .2; dw = .1; rad = a2*sqrt(p) */
#define SIM2   7    /* cl = .7 ht; dw = .132 + .049*ht; rad = a3(p) */
            /* whorl growth type aliases */
#define SECT   1 /* each quad. grows ind.; quadext=a + b*relup. of quad*/
#define HOMOG  2 /* uptake averaged over whorl; whorlext=a + b*hinc */
#define XMAS   3 /* " " ; whorlext=a + b*quad.relup*/
#define KK  0.5             /* foliage projection coefficient used
                                                    to calc. leaf
area density in cell*/
#define DT  0.6667                  /* width of treeless border
surrounding
                                                    plot */
#define A1  0.547                   /* crown angle for SIM1,
METER */
#define A2  0.8                     /*  crown
angle for HALFM */
```

```
#define A3  0.44                                        /* crown angle for SIM2 */

/*---------- Inline Functions & Numerical Constants ---------- */

#define distance(x1,y1,x2,y2)  sqrt( (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) )
#define min(a,b) ((a)>(b)?(b):(a))
#define max(a,b) ((a)>(b)?(a):(b))

                                /* sorts in ascending order */
#define sort_struct(n,c,a) \
                for (i = 0; i < n-1; i++) for (j = i+1; j < n; j++) \
        if (c[i].a > c[j].a) {temp=c[i]; c[i]=c[j]; c[j]=temp;}

/******** crown functions ********/          /* h= height, p=depth from top */
#define cr_len(h)   .007*h + .12*pow(h,0.2)
#define w_intv(h)    -.01526 + .0442*h            /* inter-whorl distance */

                                /******* irrad function ********/
#define fol_den(i,k,d)    R * pow((i - d)/100.0, k)
  /* i-irrad, k-irrad. attenuation param, d-min. req. maintenance irrad (Shift) */

#define PI     3.14159


/* ================================================================== */


/***** template structures and typedefs *****/

struct trees          /* coordinates, height, spatial occupancy */
{
 double x;
 double y;
 double h;
 double ht_inc;                 /*height increment */
 double theta;                  /* angle at which first quadrant of top whorl starts */
 int    pol;         /* will tree be effected by polution or not */
 int              cubes;                        /* spatial occupancy: # cubes occupied */
 int              biocubes;      /* biological spatial occupancy: live fol. cubes occupied */
 double folmass;                /* sum of foliage_den over all biocubes */
};
```

```
struct trees2          /* for temporary sort of AutoSam */
{
 int i;
 double h;
};

struct subject_tree    /* for crit_stand() calculations */
{
 double ht;
 double ht_inc;
};

struct whorls
{
 float rad[NZ][NQUAD];    /* radius of each whorl quadrant at each height */
 int   shift[NZ];          /* label even or odd */
};

struct occupy              /* registers of cube occupancy by number of trees */
                                  /* used as an array, each entry the frequency
count
                                     for that # cubes occupied jointly by (the
array index)
                                     trees. ex. occupancy[3].cubes = # cubes
jointly occupied
                                    by 3 trees.          */
{
 int cubes;                /* cubes occupied */
 int biocubes;             /* cubes with live foliage, occupied */
};

struct nbors          /* potential neighbors of a cell */
{
 int n;             /* number of potential neighbors of cell */
 int id[NNBR];        /* their ids */
 int q[NNBR][2];      /* their quadrants for even and odd whorls */
 double dist[NNBR];   /* their distances to cell midpoint */
};

struct sharers        /* neighbors which actually share a cell, given z */
```

```
{               /* see nbors explanation */
 int n;
 int id[NNSR];
 int q[NNSR];
};


struct cell_sample    /* characteristics of sampled trees */
{
 double irr[Nc];      /* % irradiation leaving cell */
 double fol[Nc];      /* foliage density of cell */
 double up[Nc];       /* uptake in the cell */
};

struct results                  /* simulation criteria results */
{
 int    mortality;              /* Mortality of stand by year NYEAR */
 double median;                 /* median live tree height at NYEAR */
 double ks_prob;                /* P-value of Kolmogorov-Smirnov two sample
                                        test statistic */
 double ks_stat;                /* K-S two sample test statistic */
 double sup_slope;              /* slope of regression of Ht. Inc. against
                                        height at NYEAR-2 for Suppressed trees,
                                        (ht <=2.8 meters).      */
 double sup_r2;                 /* R^2 from this regression */
 double dom_slope;              /* slope, etc., for Dominant trees (>=3.2m) */
 double dom_r2;                 /* R^2 for Dominant trees */

 int    whorlnum;               /* # of live whorls on open-grown tree */
 double angle;                  /* Crown angle (from top, looking down) of O.G.tree*/
 double cr_ratio;               /* Crown Ratio (foliage bearing length of crown
                                        versus total lenght(height) of crown */
};


/*****************************************************************
                    Declaration
*****************************************************************/

/*************************** declaration of functions ***************/

void  get_trees();
```

```
void   make_crowns();
double radius();

void grid();
void init_cell_irrad();
void free_cell_irrad();

double findnth();

void tag_sample();

void            acquisition();
void   findpotent();
double findist();
void   findshare();
void   calc_foliage();
double parameter_setting();

void utilization();
void collect_sums();
void grow_tree();
void grow_rad();
double param_linear();
void AutoSam();
void CheckLive();

int round();
double kmean();
double kstdev();

void crit_stand();
double med_live();
int compare();
int compare_ht();

void crit_open();

void height_incr();
void regress();

void kstwo_ht();
float probks();
```

```
void criteria_out();

void OutGrid();
void OutFol();
void OutTree();
void OutHinc();
void OutHt();
void OutSpace();
void OutTfl();
void OutDead();
void OutSamp();
void OutRad();
void OutRad2();
void OutRad3();
void OutJoint();
```

```
/*********************************************************************
**
                                **
**              file:

GLOBALS.H
**


** Declaration file for global variables in whorl.auto.funct.c                **
**              To be included in function files
**
*********************************************************************
***                             JHR 4/94
                **/

extern double Break[2];
extern double Crlim[2];

extern double K[3], D[3], E[3], Dead[3], H[5], W[2];

extern int AUTO_TR, NSAMP, Sam_tr[30], Pol[2];
extern double Sam_ht[1];

extern int FOLIAGE, TOTFOL, TREE, SAMP, HEIGHT, HINC, DEATH_T, RAD,
GRID, SPACE;

extern int i,j;

extern struct trees2 temp;
```

```
/*********************************************************************
*
* file:

 INPUT.C                                        *

*                                               *
*********************************************************************/
/* Revised 4/9/94 to change input approach:
                Certain parameters are fixed in the main file header,
                (need to change this file and recompile code to change parameter value);
                Read other parameter values from input file, 1 line per simulation.
                Paremeters read in for each simulation: K,D,E,DEAD,H[1],W[1]. */


#include <stdio.h>


/************************************ Input **********************/
Input ( K, D, E, Dead, H, W)

double *K;
double *D;
double *E;
double *Dead;
double *H;
double *W;

{
 FILE *fp;


/***** input *****/

 fp = fopen ("par.input.test", "r");    /* for parameter descriptions see whorl.c */

 /* PUT IN A LINE HERE TO READ & DISCARD THE HEADER LINE OF THE
INPUT FILE */
 /* this should read in the 14 label strings but not assign them to anything */
 fscanf (fp, "%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s");

 fscanf (fp, "%lf", &K[0]);                     /* light attenuation parameter */
```

```
  fscanf (fp, "%lf", &K[1]);
  fscanf (fp, "%lf", &K[2]);

  fscanf (fp, "%lf", &D[0]);                /* min. irr. required for maintenance*/
  fscanf (fp, "%lf", &D[1]);                /* 'Shift'
*/
  fscanf (fp, "%lf", &D[2]);

  fscanf (fp, "%lf", &E[0]);                /* Energy Conversion Efficiency */
  fscanf (fp, "%lf", &E[1]);
  fscanf (fp, "%lf", &E[2]);

  fscanf (fp, "%lf", &Dead[0]);                          /* 'Dead' */
  fscanf (fp, "%lf", &Dead[1]);
  fscanf (fp, "%lf", &Dead[2]);

  fscanf (fp, "%lf", &H[1]);                /* Height Increment */

  fscanf (fp, "%lf", &W[1]);                /* Sector Increment */

  fclose (fp);


/***** output to screen *****/

   puts("");
   printf ("K        %lf %lf %lf\n", K[0], K[1], K[2]);
   printf ("D        %lf %lf %lf\n", D[0], D[1], D[2]);
   printf ("E        %lf %lf %lf\n", E[0], E[1], E[2]);
   printf ("Dead      %lf %lf %lf\n", Dead[0], Dead[1], Dead[2]);
   printf ("H        %lf %lf   %lf %lf %lf\n", H[0], H[1],
                                  H[2], H[3], H[4]);
   printf ("W        %lf %lf\n", W[0], W[1]);

}
```

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

SETUP.C
*

* Setup functions: get_trees, make_crowns, set_class, radius, grid,        *
*                                          init_cell_irrad,                free_cell_irrad
*
*********************************************************************/

#include "whorl.auto.h"
#include "globals.h"


/*********************************************************************
*                                                  *
*                Setup  FUNCTIONS                        *
*
*********************************************************************/

/********************************** get_trees **********************/
/*****
  Reads subject trees coordinates and heights from the input dataset.
  Randomly assigns the angle (theta) of the first sector of the top whorl
  (0 - PI/2 radians) [to avoid all sectors lining up N -S for example]
  and a pollution flag.

*****/
void get_trees (fp, tree, Nsub, hmax, reallen)
FILE  *fp;
struct trees *tree;   /* array of tree records - coord, ht, theta, poll */
int   *Nsub;        /* number of subject trees */
double *hmax;        /* maximum tree height */
double *reallen;     /* real crown length if crown type = REAL */
{
 int  isub;

 double srand48(), drand48();
 double rand;                  /* random number */
 double seed = 54591;             /* random number seed */
```

```
  fscanf(fp,"%d ", Nsub);                    /* number of subject trees */
  for (isub = 0; isub < *Nsub; isub++)
   fscanf(fp,"%lf ",&tree[isub].x);          /* x coordinates */
  for (isub = 0; isub < *Nsub; isub++)
   fscanf(fp,"%lf ",&tree[isub].y);          /* y coordinates  */
  for (isub = 0; isub < *Nsub; isub++)
  {
   fscanf(fp,"%lf ",&tree[isub].h);          /* heights */
   if (tree[isub].h > *hmax) *hmax = tree[isub].h;
  }
  if (CROWN_TYPE == REAL)                    /* crown_type set in input parameters */
   for (isub = 0; isub < *Nsub; isub++)
    fscanf(fp,"%lf ",&reallen[isub]);        /* real lengths of crown */

  srand48(seed);
  for (isub = 0; isub < *Nsub; isub++)
  {
   rand = drand48();
   tree[isub].theta = PI/2*rand;  /* assign angle of first branch */

   if (rand > 0.5)
    tree[isub].pol = 1;                      /* assign pollution flag */

   tree[isub].ht_inc = 0.0;                  /* zero initial ht. increments */
  }
}


/********************************* make_crowns ******************/
/*****
  Called from main during initialization. For each tree it computes the
maximum whorl radius (based on crown class) and fills the whorl structure.
the shift (whether it is an even or odd whorl), and the radius of the
quadrants.  All quadrants of a whorl are equal at this point.
  NOTE: Whorl rad at this height is assigned to an approximate z level
round(wht/DZ). If dw is small (relative to DZ) two whorls will be assigned
to the same z level and the last radius (the largest) will be saved.
Unfortunately only the last shift gets saved too.  This is a bug that I'm not
going to fix. Only small trees are effected.
```

```
*****/
void make_crowns (whorl,  tree, Nsub, maxrad, reallen)
struct whorls *whorl;   /* whorl structure for this tree to be calculated */
struct trees *tree;     /* tree array */
int   Nsub;           /* number of trees */
double *maxrad;         /* maximum radius of each tree */
double *reallen;       /* real crown length, crown class = REAL */
{
 int   isub, iz, iq;  /* index for tree, height, and quad loops */
 double cl,           /* crown length */
      dw,           /* inter whorl distance */
      nw,           /* number of whorls */
      wht;          /* estimated whorl height (from ground ) */

 for (isub = 0; isub < Nsub; isub++)
 {

  dw = w_intv(tree[isub].h);                /* actual inter-whorl dist */
  switch(CROWN_TYPE)
  {                         /* get crown length, possibly alter dw */
    case(REAL)   : cl = reallen[isub];
                break;
    case(FUNCT)  : cl = cr_len(tree[isub].h);
                break;
    case(LONG)   : cl = max(.15,tree[isub].h - LLB);
                break;
    case(SIM1)   : cl = max(0.15, tree[isub].h *0.75);
                break;
    case(METER)  : cl = .7*tree[isub].h;
                                    break;
    case(HALFM)  : dw = 0.1;
                cl = 0.2;
                break;
    case(SIM2)   : dw = 0.1316525 + 0.0494915*tree[isub].h;
                                    cl = 0.7*tree[isub].h;
                break;
  }
  nw = round(cl / dw);                              /* no of whorls */

  maxrad[isub] = radius (((double) nw) * dw, tree[isub].h, CROWN_TYPE);

  for (iz = 0; iz < nw; iz++)
```

```
    {
     wht = tree[isub].h - (iz + 1) * dw;        /* calculate whorl height */
     whorl[isub].shift[NZ - round(wht/DZ)] = (iz % 2);/*assign shift to whorl*/

     for (iq = 0; iq < NQUAD; iq++)
      whorl[isub].rad[NZ - round(wht/DZ)][iq] =
          radius (((double) (iz + 1) * dw), tree[isub].h, CROWN_TYPE);
                                                        /* assigns radius to each
quad. of whorl */
    }
  }
}


/********************** radius ************************************/
/*****
 Computes the radius of the whorl as a function of the distance from
  the top of the tree and a whorl angle constant An.
*****/
double radius (p, h, key)
double p, h;   /* p - depth of whorl from top of tree; h - tree ht */
int   key;    /* crown_type, defines function used for calc. radius */
{
 switch (key)
 {
  case SIM1 : case METER : return (A1 * sqrt(p));
  case HALFM :          return (A2 * sqrt(p));
  case SIM2 :           return (A3 * p);
  case REAL : case FUNCT :
  case LONG :           return (exp(.366 - .2096*h)*sqrt(p));
  }
}


/*****************grid ************************************/
/*****
 Calculates minimum and maximum dimensions of grid based on coord.s of
 subject trees *****/

void grid (xmin, ymin, tree, NCx, NCy, Nsub, stand)

double *xmin, *ymin;  /* minimum dimensions of grid, starting corner*/
```

```
struct trees *tree;
int   *NCx, *NCy;    /* x, y dimensions of grid (and cell_irrad array) */
int   Nsub, stand;   /* stand - 1 : Stand, -2 : Open Grown */
{
 int isub;
 double xmax =  0,    ymax =  0;

 *xmin =  10000;
 *ymin =  10000;

 for (isub = 0; isub < Nsub; isub++)
 {
  if(tree[isub].x < *xmin) *xmin = tree[isub].x;
  if(tree[isub].x > xmax) xmax = tree[isub].x;
  if(tree[isub].y < *ymin) *ymin = tree[isub].y;
  if(tree[isub].y > ymax) ymax = tree[isub].y;
 }

 if ((stand == 1)&&(Nsub==327)) {
                /* add treeless border to Observed Stand simulations */
    *xmin -= DT/2;
    xmax += DT/2;
    *ymin -= DT/2;
    ymax += DT/2;}

 if ((stand == 1)&&(Nsub==340)) {
                /* Adjust max for uniform grid for Hexagonal Spatial sim.s */
    xmax += .18;
    ymax +=.312;}

 *NCx = round((xmax-*xmin)/CW);                  /* no. of cells each dim */
 *NCy = round((ymax-*ymin)/CW);

}


/****************************** init_cell_irrad ********************/

void init_cell_irrad(cell_irrad, NCx, NCy)

double ****cell_irrad;  /* cell_irrad array, to be dynamically allocated */
int              NCx, NCy;
```

```
{
int ix, iy;
double ***temp_array;

 /* DYNAMICALLY ALLOCATE cell_irrad ARRAY */

 temp_array = (double ***) calloc(NCx, sizeof(double **));
 if (temp_array == 0 ) {
                puts("No Memory for Cell_Irrad");
    exit (1);
                }

 for (ix = 0; ix < NCx; ix++){  /* step through X dimension */
   temp_array[ix]= (double **) calloc (NCy, sizeof(double *));
   if (temp_array[ix] == 0) {
                puts("No Memory for Cell_Irrad[ix]");
     exit (1);
     }

   for (iy = 0; iy < NCy; iy++) {
                temp_array[ix][iy]= (double *) calloc(NZ, sizeof(double));
                if (temp_array[ix][iy] == 0) {
                                puts("No Memory for Cell_Irrad[ix][iy]");
                                exit (1);
                                }
   } /* end of iy */
 }  /* end of ix */

 *cell_irrad = temp_array;

}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% */
void free_cell_irrad(cell_irrad, NCx, NCy)

double ***cell_irrad; /* cell_irrad array, to be dynamically allocated */
int             NCx, NCy;

{
int ix, iy;
```

```
 /* FREE cell_irrad ARRAY */

 for (ix = 0; ix < NCx; ix++){  /* step through X dimension */
    for (iy = 0; iy < NCy; iy++) {
                free((char *) cell_irrad[ix][iy]);
     }  /* end of iy */
    free((char *) cell_irrad[ix]);
  }  /* end of ix */

 /* free original pointer contents */
 free((char *) cell_irrad);

}
```

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

ACQUISITION.C
                                **

** FUNCTIONS: tag_sample, acquisition, findpotent, findist, findquad,      **
**                    findshare, calc_foliage, parameter_setting.
**
*************************************************************************/
#include "whorl.auto.h"
#include "globals.h"

/*extern void OutFol();*/



/************************ tag_sample ******************************/
/****
 Tags the cell if it is in the inner 3/5 of the plot area, in both
  the x and y directions.
****/

void tag_sample (xcell, ycell, NCx, NCy, tag, i1)
double xcell, ycell;
int   *tag, *i1;
{
 if ((xcell/CW < NCx*4/5) && (xcell/CW > NCx/5)
                &&
    (ycell/CW < NCy*4/5) && (ycell/CW > NCy/5))
                {
                                *tag = 1;
                                *i1 += 1;
                }
 else *tag = 0;
}


/************************ acquisition *******************************/
/*****
                Loops through cells in a column (top to bottom), columns on grid, to
```

calculate the irradiation interception of each sector of each whorl
of each tree.

CURRENTLY THE COMMENTED OUT CALL TO SAMPLE1 NEEDS TO BE DEBUGGED

****/

```
void acquisition (xmin, ymin, cell_irrad, NCx, NCy, tag, i1, tree, Nsub,
                              nbr, maxrad, nslice, whorl, share, total_laf, iyear,
                              foliage, uptake, no_cells, sample1, SAMP, FOLIAGE,
Sam_tr, ffp,
                              occupancy)

double xmin, ymin;
double ***cell_irrad;
struct trees *tree;
struct whorls *whorl;
struct cell_sample sample1[NZ];
struct occupy *occupancy;

NBRPTR nbr;
SHAREPTR share;

int NCx, NCy, nslice, Nsub, *tag, *i1, iyear, no_cells[NZ][NT][NQUAD];
int SAMP, FOLIAGE, *Sam_tr;

double *maxrad, *total_laf;
double foliage[NZ][NT][NQUAD];
double uptake[NZ][NT][NQUAD];
FILE *ffp;


{
 int ix, iy, iz, inbr;  /* iz - vertical (whorl), inbr - neighbor indices*/
 double irrad, irrad_removed; /* irradiation entering & removed from cell*/
 double foliage_den;   /* new fol.density (% irrad. captured) for each
                                                        cell */

 double xcell, ycell;              /* cell coordinate variables */
```

```
double prior_irrad;              /* cell's irrad. last year */
double Esetting;        /* foliage cell's current Conversion efficiency*/
int maxnbr=0, maxshare=0; /* cell sample parameters */



/* ---------------------------- cell loop ---------------------------- */

  for (ix=0; ix < NCx; ix++)
  {
   for (iy=0; iy<NCy; iy++)
   {                              /* loop for each column */
    irrad = 100.0;   /* percent light incoming at top of canopy */

                xcell = xmin + CW/2 + ix*CW;  /* Calculate cell center coordinates */
                ycell = ymin + CW/2 + iy*CW;

    if (SAMP)                    /* tag if cell should be sampled */
     tag_sample (xcell, ycell, NCx, NCy, tag, i1);



    findpotent(tree, Nsub, nbr, maxrad,  /* find potential nbors of cell */
            xcell, ycell, NCx, NCy);

    if (nbr->n > maxnbr) maxnbr = nbr->n;              /* output only */


    for (iz = NZ-nslice; iz < NZ; iz++)
    {
     irrad_removed = 0;
                prior_irrad = cell_irrad[ix][iy][iz];
                cell_irrad[ix][iy][iz] = irrad;  /*store this year's irrad. level*/

     findshare(whorl, share, nbr, iz, no_cells);
                /* find actual nbors of cell given z */

                if (share->n)
                {

      if (share->n > maxshare) maxshare = share->n;     /* output only */
```

```
                    calc_foliage (prior_irrad, &foliage_den, share, irrad, total_laf, tree,
occupancy);
                  /* foliage_den is each sharer's proportion */

        if (FOLIAGE && (iyear == NYEAR))
         OutFol (ffp, share, foliage_den, Sam_tr, xcell, ycell,
              iz, NSAMP, irrad);

                  Esetting = parameter_setting(E, prior_irrad);

        for (inbr = 0; inbr < share->n; inbr++)  /* assign uptake to quad */
        {
         uptake [iz][share->id[inbr]][share->q[inbr]] +=
                Esetting * irrad * foliage_den;


         foliage[iz][share->id[inbr]][share->q[inbr]] += foliage_den;

         irrad_removed +=irrad * foliage_den;
        }
        irrad -= irrad_removed;  /* adjust irrad to enter cell below */
                }

                if(tag)
      { /* REMOVE AFTER DEBUGGING AT SOME POINT
                 sample1[iz].irr[i1] = irrad; /* sample irrad leavingXX
                 sample1[iz].fol[i1] =
                 foliage[iz][share->id[inbr]][share->q[inbr]];
                 sample1[iz].up[i1] = uptake[iz][share->id[inbr]][share->q[inbr]];
                 */
      }
     }   /* bottom of column */
    }   /* y end */
   }   /* x end */


  if (SAMP)
   printf("no. of samples averaged per layer %d  Nc  %d\n", i1, Nc);

 /*printf("\nMAXIMUM NBR: %d   NNBR %d\n", maxnbr, NNBR);*/ /*#pot.nbrs-
real,max*/
```

```
 /*printf("MAXIMUM NSHARE: %d  NNSR %d\n", maxshare, NNSR);*//*#sharer-
real,max*/

   maxshare = maxnbr =0;  /* reset for next year's call */
} /* end acquisition */




/********************** findpotent **********************************/
/*****
                called from acquisition()

 ID trees that potentially overlap a cell.  Think of a cell as a column
 in space.  For this cell, loop through tree array, computing distance
 from the tree to the center of the cell.  If the length of that tree's
 longest branch is equal to or greater than the distance, add the tree
 to the list of potential neighbors.

*****/

void findpotent (tree, Nsub, nbr, maxrad, xcell, ycell, NCx, NCy)
struct trees *tree;   /* array of trees */
int   Nsub;         /* number of trees */
NBRPTR nbr;           /* array of potential neighbors */
double *maxrad;       /* maximum whorl radius of each tree */
double xcell, ycell;  /* coordinates of cell center */
int   NCx, NCy;      /* number of x, y cells */
{
 int isub;          /* tree loop index */
 double dist;       /* distance from cell center to tree */
 double mx, my;     /* delta x and delta y from chosen tree to cell center */


 nbr->n = 0;

 for (isub = 0; isub < Nsub; isub++)      /* loop through trees */
 {
  if (tree[isub].h != 0)
  {
   dist = findist(tree[isub].x, tree[isub].y, maxrad[isub],
```

```
                xcell, ycell, NCx, NCy, &mx, &my);

    if (dist <= maxrad[isub])        /* if potential nbr */
    {
     nbr->id[nbr->n]   = isub;
     nbr->dist[nbr->n] = dist;

     nbr->q[nbr->n][0] = findquad (tree, mx, my, 0, isub); /* returns quad */
     nbr->q[nbr->n][1] = findquad (tree, mx, my, 1, isub); /* of tree which
                                                              contains cell for even/odd
shifted whorl */

     nbr->n += 1;
    }
   }
  }
}


/********************** findist ***********************************/
/*****
            called from findpotent().

  Returns the distance between a tree and a cell center.  This is easy
  except at the edges, where we will assume a wrap around effect.  If the
  tree is within maxrad of one edge, and the cell is within maxrad of the
  opposite edge, the x (or y) coordinate is transposed by subtracting the
  plot dimension from the larger one, so they are close together.
*****/

double findist (xtree, ytree, maxrad, xcell, ycell, NCx, NCy, mx, my)
double xtree, ytree;     /* coordinates of tree */
double maxrad;           /* maximum branch length of that tree */
double xcell, ycell;     /* coordinates of cell */
int   NCx, NCy;          /* number of x, y cells */
double *mx, *my;         /* delta x, delta y between tree and cell center */
{
 double xtrans = NCx*CW;  /* plot size in x direction */
 double ytrans = NCy*CW;  /* plot size in y direction */
            /* adjust for edge overlap */
 if ((xcell < maxrad) && (xtree > (xtrans - maxrad)))      /* west cell,  */
```

```
    xtree -= xtrans;                                /*  east tree  */
  else if ((xcell > (xtrans - maxrad)) && (xtree < maxrad))  /* east cell,  */
    xtree += xtrans;                                /*  west tree  */

  if ((ycell < maxrad) && (ytree > (ytrans - maxrad)))      /* south cell,  */
    ytree -= ytrans;                                /*  north tree  */
  else if ((ycell > (ytrans - maxrad)) && (ytree < maxrad))  /* north cell,  */
    ytree += ytrans;                                /*  south tree  */

  *mx = xcell - xtree;
  *my = ycell - ytree;

  return (distance (xtree, ytree, xcell, ycell));
}


/*************** findquad ***********************************/
/*****
                called from findpotent().

   Calculates the quad (0,1,2,3) of the tree that contains this cell.
   Calculations are relative to the set of axes defined by the tree as
   center and the first branch of the top whorl lying along the x-axis.
   Tree.theta is the angle of the first branch to the plots x-axis.
   For odd whorls, the first branch is shifted by PI/NQUAD, so branches
   are staggered.  The angle from tree to cell center is computed, then
   transposed by theta to give angle relative to the whorl's quadrant.  The
   quadrant containing the cell is then computed.

*****/
int findquad (tree, mx, my, shift, isub)
struct trees *tree;  /* tree array*/
double mx, my;      /* delta x, delta y between tree and cell */
int   shift;       /* 0 for even whorls, 1 for odd */
int   isub;        /* index of this tree in tree array */
{
 double r;         /* distance from tree to cell center */
 double rho;       /* angle between tree and cell center */
 double trans;     /* angle of branch to x axis.  */

 /* get the angle of the first branch */
 trans = tree[isub].theta + shift*PI/NQUAD;
```

```
  /* get the angle of the cell from the tree, in plot grid coordinate system */
  r = sqrt(mx*mx + my*my);
  if (my > 0)
   rho = acos(mx/r);
  else
   rho = 2 * PI - acos(mx/r);

  /* transpose rho by rotating the first branch of the whorl down to
    coincide with the x-axis. */
  rho -= trans;      /* rho relative to coord. system of tree and 1st branch */
  if (rho < 0)
   rho+=2*PI;

  return ((rho*NQUAD)/(2*PI));      /* quad of tree cell is in */
}



/********************** findshare **********************************/
/*****
                called from acquisition().

  Identifies trees and quadrants that actually share a cell in the vertical
  dimension in the x-y column.

*****/
void findshare (whorl, share, nbr, z, no_cells)
struct whorls *whorl;  /* whorl structure for the trees */
SHAREPTR share;        /* array of actual neighbors (returned ) */
NBRPTR  nbr;           /* array of potential neighbors */
int z;                 /* vertical dimension index  */
int no_cells[NZ][NT][NQUAD];
{
 int i;        /* Note: nbr.* is indexed by i while Nsub length structures
                  are indexed by nbr.id[i] */

 share->n = 0;                  /* index of share structure array */
 for (i = 0; i < nbr->n; i++)
 {
  /* for each neighbor tree, see if there is a whorl at that level
       that extends into the cell */
```

```
    if (nbr->dist[i] <=
        whorl[ nbr->id[i] ].rad[z][ nbr->q[i][whorl[nbr->id[i]].shift[z]] ])   {
     /* add it to the share array */
     share->id[share->n] = nbr->id[i];
     share->q[share->n]  = nbr->q[i][ whorl[nbr->id[i]].shift[z] ];
     /* update the number of cells, at this whorl level, occupied by
            this quad of this tree */
     no_cells[ z ][ share->id[share->n] ][ share->q[share->n] ] += 1;
     share->n += 1;              /* update actual # of sharers */
     }
  }
}


/********************** calc_foliage *********************************/
/*****
                called from acquisition().

 Calculate foliage density for each sharer.  FOLIAGE density is defined
by the interception rate it results in.

*****/

void calc_foliage (prior_irrad, foliage_den, share, irrad, p_total_laf, tree, occupancy)

double prior_irrad;     /* cell's irrad. level last year */
double  *foliage_den;  /* interception rate of light in this cell */
SHAREPTR share;         /* array of actual neighbors in this cell */
double  irrad;         /* irradiation coming into this cell */
double  *p_total_laf;    /* address of  leaf area index */
struct trees *tree;                 /* array of trees structures */
struct occupy *occupancy;  /* array of joint occupancy of cube registers */

{
 double shift, Ksetting;  /* D-the min. req. maintenance irrad. */
                                                /* Ksetting - irradiation capture efficiency
*/
 double tot_fraction_rm = 1.0;  /* fraction of light remaining, later
                    becomes fraction of light removed */
 int inbr;                          /* tree array index */
```

```
/* get "shift" (D, min. req. maintenance irrad level) as function of
  prior year's irrad. environment */

  shift = parameter_setting(D, prior_irrad);

  if (irrad > shift)
  {                    /* % light captured by ith nbr */
   Ksetting = parameter_setting(K, prior_irrad);
   *foliage_den = fol_den(irrad, Ksetting, shift);
  }
  else
   *foliage_den = 0.0;

tot_fraction_rm *= pow(1 - *foliage_den,(double) share->n);  /* % light uncaptured */

*p_total_laf += -log(tot_fraction_rm)/(CW*10*KK); /* at this point
                tot_fraction_rm is total proportion remaining */

tot_fraction_rm = 1.0 - tot_fraction_rm;   /* now total proportion removed*/
                 /* equal to probability irrad captured by a nbr */

/*
  Adjust foliage_den: If share->n > 1.  Calculate total proportion
removed by all nbors.  Partition this total proportion equally among
the sharers.  */

if (share->n > 1)
 {
   *foliage_den =  tot_fraction_rm/share->n;
 }

/* update spatial occupancy registers */
occupancy[share->n].cubes++;
if (*foliage_den > 0.0) occupancy[share->n].biocubes++;

for (inbr=0; inbr < share->n; inbr++)
 {  /* update spatial occupancy indices */
  tree[share->id[inbr]].cubes++;
  if (*foliage_den > 0.0) {
    tree[share->id[inbr]].biocubes++;
    tree[share->id[inbr]].folmass += *foliage_den;
```

```
    }
  }
}


/*************** Parameter_setting*******************************/
/* calculate the physiological parameter setting (sun / shade) for a cell of
                foliage based on the previous year's light environment. */

double parameter_setting(Param, prior_irrad)

 double Param[3];  /* shade setting, sun setting, breakpoint */
 double prior_irrad;

 {
  if ((prior_irrad/100.0) <= Param[2])
                return(Param[0]);
  else
                return(Param[1]);
 }
```

```
/*********************************************************************
        • file:

 UTILIZATION.C
                        *

*  FUNCTIONS:  utilization, collect_sums, grow_tree, grow_rad, findnth*
*                                  param_linear                    5/18/95
                        *
*********************************************************************/

#include <math.h>
#include <stdio.h>
#include "whorl.auto.h"
#include "globals.h"


extern double radius();
extern void OutTree();
extern void OutHinc();
```

```
/***************** utilization ***********************************/
/****
                Called from main(), year loop.

   Calculate the total relative uptake of each sector of each whorl of each
   tree. Calculate the growth of each sector of each whorl of each tree, and
   the height increment of each tree, based on this relative uptake of
   resource and the growth relations in grow_tree().

                                                        *****/


void utilization ( Nsub, iyear, nslice, tree, whorl,  foliage, uptake,
                                no_cells, relup, maxrad, total_dead, total_ht,
                                TREE, Sam_tr, NSAMP, HINC,  tfp, hifp)

struct trees *tree;
struct whorls *whorl;

int             Nsub, iyear, *nslice,  *total_dead, no_cells[NZ][NT][NQUAD];
double          foliage[NZ][NT][NQUAD], uptake[NZ][NT][NQUAD];
double  relup[NT], *maxrad, *total_ht;

int             TREE, *Sam_tr, NSAMP, HINC;

FILE            *tfp, *hifp;

{
 int            isub;
 double         area[NT],       /* horiz. area occuppied */
                                hinc,                           /* ht. incr. for a tree */
                                htmax;                          /* max. current tree ht. */



  htmax = 0.0;
  for (isub = 0; isub < Nsub; isub++)           /* loop through trees */
  {
   hinc = 0.0;

   if ((TREE) && (iyear == NYEAR))              /* every SAM_INT years */
```

```
    OutTree (tfp, isub, Sam_tr, NSAMP, foliage, uptake, no_cells, *nslice);

   if (tree[isub].h != 0)          /* for currently live trees */
   {
    collect_sums (relup, area, isub, foliage, uptake, no_cells, *nslice);
    grow_tree (whorl, tree,  maxrad, foliage, uptake, no_cells, relup,
          &hinc, &htmax, total_dead, isub, *nslice);
    *total_ht += tree[isub].h;          /* update total ht of stand */

            if (iyear > NYEAR -2)
                        {
                        tree[isub].ht_inc += hinc;      /* record ht. incr. info */
                        }
   }
    if (HINC && iyear > NYEAR-2)
            OutHinc (hifp, hinc, isub, Nsub);
  }


  /*printf("  MEAN HEIGHT %lf\n", *total_ht/Nsub);*/


  /* --------- extend vertical grid ---------- */

  *nslice = round(htmax/DZ);

} /* END UTILIZATION FUNCTION */



/****************** collect_sums ******************************/
/*****
  Called from main in the resource utilization section.  It sums
  up the ammount of relative uptake and area covered over the
  whorls and quadrants for a given tree.

*****/
void collect_sums (relup, area, isub, foliage, uptake, no_cells, nslice)
double foliage[NZ][NT][NQUAD],  /* foliage per quad per whorl for each tree */
    uptake[NZ][NT][NQUAD],  /* uptake per quad per whorl for each tree */
    *relup,   /* relative uptake  uptake/foliage ammount*/
```

```
    *area;    /* horizontal cell area occupied by the tree */

int   no_cells[NZ][NT][NQUAD], /* no cells occupied per qd per wh per tree */
      isub,    /* tree index */
      nslice;  /* number of vertical slices */
{
 int iz, iq;

 relup[isub] = area[isub] = 0.0;

 for (iz = NZ-nslice; iz < NZ; iz++)        /* sum over whorls */
  for (iq = 0; iq < NQUAD; iq++)            /* sum over quads of whorl  */
  {
   if (foliage[iz][isub][iq] != 0)
    relup[isub] += uptake[iz][isub][iq]/foliage[iz][isub][iq];
   area[isub]  += (double) no_cells[iz][isub][iq];
  }
 area[isub] *= CW*CW;
}


/****************** grow_tree *********************************/
/***** Called from main after all vertical columns of all cells have been
   processed, and some sums collected for each tree:  relative uptake, area
   occuppied.  Computes  height increment, calls "grow_rad" to compute
   radius increments of quadrants, and flags trees which are determined
   to have died.

*****/

void grow_tree (whorl, tree,  maxrad, foliage, uptake, no_cells,
                              relup, hinc, hmax, p_total_dead, isub, nslice)

struct whorls *whorl;  /* whorl structure for the trees */
struct trees *tree;    /* tree array */

double *maxrad,        /* maximum radius array for trees */
               foliage[NZ][NT][NQUAD], /* foliage per qd. per wh. per tree */
               uptake[NZ][NT][NQUAD],  /* uptake "              "  */
               *relup,        /* relative uptake array for trees */
               *hinc,         /* height increment of tree */
               *hmax;          /* maximum height of tree */
```

```
int   no_cells[NZ][NT][NQUAD], /* no cells occupied per qd. per wh. per tree*/
              *p_total_dead,    /* total number of dead trees */
              isub, nslice;   /* tree index,  number vertical sections */
{
 int iq, new_z;

 /*  algorithm:  WGROW_TYPE = 1 or 2 or 3   (1-SECTOR, 2-HOMOG, 3-XMAS)
                                (1) determine hinc
                                (2) grow radii (dependent or independent on hinc)
                                (3) grow new whorl then add hinc IF any radii!=0  ELSE
ht = 0 */


              /* determine hinc */
  if (NLH)                  /* nonlinear ht. growth eqn flag */
   *hinc = max (0, H[2]*relup[isub]*H[3]/(H[2]*relup[isub]+H[3]) - H[4]);
  else
   *hinc = max (0, H[0] + H[1] * relup[isub]);

  if (Pol[1] && tree[isub].pol)              /* pollution effects */
   *hinc /= 4;

  grow_rad (whorl,  foliage, uptake, no_cells, maxrad,  isub, nslice);
  /* grow whorls */

  if(maxrad[isub])  /* increment height and add new whorl if maxrad not 0 */
  {                 /* # vert. levels tree occupies, ht unupdated */
   new_z = round(tree[isub].h/DZ);
   for (iq = 0; iq < NQUAD; iq++)
    if (whorl[isub].rad[NZ - new_z][iq] == 0)  /* add new whorl if nec. */
     whorl[isub].rad[NZ - new_z][iq] = radius (*hinc, tree[isub].h, METER);
   tree[isub].h += *hinc;
  }
  else
  {
   tree[isub].h = 0;            /* kill if all sectors are dead */
   *p_total_dead +=1;                 /* count newly dead trees */
  }

  if (*hmax < tree[isub].h)       /* update max. ht. attained, if nec. */
   *hmax = tree[isub].h;
```

```
}


/************************ grow_rad ********************************/
void grow_rad (whorl,  foliage, uptake, no_cells, maxrad,  isub, nslice)
struct whorls *whorl;
double foliage[NZ][NT][NQUAD];
double uptake[NZ][NT][NQUAD];
double *maxrad;      /* maximum radius array for trees */
int   no_cells[NZ][NT][NQUAD];
int   isub, nslice;  /* tree index, number of vertical slices */
{
 int iz, iq;       /* vertical index, quad index */
 double Dead_setting;          /* maintenance cost of branch - function of lenght*/

 maxrad[isub] = 0;

                /* PLASTIC SECTOR CROWNS */

  for (iz = NZ-nslice; iz < NZ; iz++)  /* loop through whorls, top down */
  {
   for (iq = 0; iq < NQUAD; iq++)
   {

/* KILL SECTOR --------*/
     Dead_setting = param_linear(Dead, whorl[isub].rad[iz][iq]);
     if (foliage[iz][isub][iq] == 0 ||
        uptake[iz][isub][iq]/foliage[iz][isub][iq] < Dead_setting)
      whorl[isub].rad[iz][iq] = 0;
     else                          /* GROW SECTOR -------*/
               {
      whorl[isub].rad[iz][iq] +=
        max(0, W[0] + W[1] * uptake[iz][isub][iq]/foliage[iz][isub][iq]);
      if (whorl[isub].rad[iz][iq] > maxrad[isub])
       maxrad[isub] = whorl[isub].rad[iz][iq];
               }
   } /* end quad loop */
  } /* end column loop */


 for (iz = NZ-nslice; iz < NZ; iz++) /* ------------ FINISH UP --------*/
  for (iq = 0; iq < NQUAD; iq++)
```

```
    {
     uptake[iz][isub][iq]  = 0;
     foliage[iz][isub][iq]  = 0;
     no_cells[iz][isub][iq] = 0;
    }
}   /* back to grow_tree */


/********************** findnth ************************************/
/*****   Returns height of nth tallest tree in stand

*****/
double findnth (tree, n, len)
struct trees *tree;
int   n, len;
{
 int   i, count;
 double mx, ceil;

 mx=0.0;
 ceil = NZ*DZ;              /* current ht. ceiling for trees */

 for (count=0; count<n; count++)
 {

  for (i=0; i<len; i++)
   if ((tree[i].h > mx) && (tree[i].h < ceil))
    mx = tree[i].h;       /* set mx to ht. of tallest tree < ceiling */

  ceil = mx;
  mx  = 0;
 }
 return ceil;
}



/*************** param_linear ************************************/
/* Calculates the maintenance cost (Dead_setting) of a branch as a piecewise linear
  function of its length   5/95 */

double param_linear(param, radius)
```

```
double param[3];              /* Dead - [0] - minimum cost
                                                    [1] - intercept
                                                    [2] - slope*/
float radius;

{ double cost;
  cost = param[1] + (double) radius*param[2];
  return( max(cost, param[0]));
}
```

```
/*************************************************************************
**  FILE:

 CRIT_STAND.C
**

**  The function crit_stand (calculate Stand Criteria) calls the**
**  following functions: median, kstwo, height_incr          **
**                              7/11/94  jhr
                **
*************************************************************************/

#include "whorl.auto.h"

extern void kstwo_ht();
extern void height_incr();



/*--------------------Criteria_stand----------------------*/
/*  Called from main() of whorl.auto, at the end of the year loop.
   Calculates the median, mortality, Kolmogorov-Smirnov, & Height
   Increment (slope & R^2 for dominant and suppressed trees)
   criteria measures for stand simulations.

   NOTE: passing Observed LIVING heights in height_obs & Osub, already
   sorted and shifted to index from 1 ... Osub, from main.
   ALSO: need to keep track of ht_inc as element of tree structures.*/


void crit_stand (mortality, tree, Nsub, height_obs, Osub,
                 sim_crit)


 struct trees tree[];
 struct results *sim_crit;    /* structure of criteria results */

 int             mortality, Nsub, Osub;
 double  height_obs[];

{
```

```
 struct subject_tree tree_size;  /* generic for sizeof() calls */

 /* allocate memory for the array */

 struct subject_tree  *sim_tree = (struct subject_tree *)malloc( (Nsub+1) *
                sizeof(tree_size));  /* array of ht & ht.inc structures */

 struct subject_tree *living_tree;  /* ptr to subarray of living
                                                                  trees        in
sim_tree*/
 int   isub;


 /* make a permutable copy of the tree heights  in sim_tree array*/
 /* Note that index is shifted from 0..Nsub-1 to 1...Nsub */

 for (isub = 0; isub < Nsub; isub++) {
    sim_tree[isub+1].ht = tree[isub].h;
    sim_tree[isub+1].ht_inc = tree[isub].ht_inc;
    }


/* sort array of sub_tree structs by .ht elements */
qsort((char *)(sim_tree+1),Nsub,sizeof(tree_size), compare_ht);


/* only want to keep nonzeros */
living_tree = sim_tree + mortality;        /* Create subarray of live trees

                from sim_tree array, with index

                1...(Nsub-mortality) */


sim_crit->mortality = mortality;
sim_crit->median = med_live(living_tree, Nsub-mortality);
```

```
  kstwo_ht(living_tree, Nsub-mortality, height_obs, Osub, sim_crit);


  /* calculate Pre-height (height at NYEAR -2) */
  for (isub = 1; isub <= Nsub-mortality; isub++)
     living_tree[isub].ht = living_tree[isub].ht - living_tree[isub].ht_inc;



  height_incr(living_tree, Nsub-mortality, sim_crit);


 free((char *) sim_tree);

  } /* end of criteria_stand */



/* ------------------ Median ----------------------------*/
/* Calculates the median entry in an already sorted array of subject_tree
                structures*/

double med_live ( tree_array, length)

struct subject_tree tree_array[];
int   length;

{
 double med;

 if (length%2!=0)
                med=tree_array[(length/2)+1].ht;
 else
                med = (tree_array[(length/2)].ht + tree_array[(length/2) + 1].ht)/2;

 return (med);
} /* end of med_live */


/* compare function for use in standard qsort, thanks to
  Dr. Zabel                                                    */
```

```
int    compare(f, g)
   float      *f, *g;
{
   int val;
   if ((*f - *g) > 0) val = 1;
   if ((*f - *g) == 0.000) val = 0;
   if ((*f - *g) < 0) val = -1;
   return(val);
}


/* Compare heights function for sorting sim_tree array of subject_tree
                structures */

int    compare_ht(f, g)

 struct subject_tree    f, g;
{
   int val;
   if ((f.ht - g.ht) > 0) val = 1;
   if ((f.ht - g.ht) == 0.000) val = 0;
   if ((f.ht - g.ht) < 0) val = -1;
   return(val);
}
```

```
/*****************************************************************
        • FILE:

 CRIT_OPEN.C
**

* crit_open( Calculate the open grown criteria)                **
* 7/11/94 jhr                                  **
*****************************************************************/

#include "whorl.auto.h"


/*----------------Crit_Open-------------------------------*/
/* Called from main() of whorl.auto, at the end of the year loop
  for the opengrown simulation. Calculates the Crown Length Ratio,
  # live whorls, and Crown Angle.
  Is passed just the open grown tree's entry for whorl & tree arrays,
  i.e. pass whorl[0], tree[0]. As open-grown trees are symmetric,
  just look at quad '0' for whorl radius.                */

void crit_open(whorl, tree, nslice, sim_crit)

 struct whorls whorl;
 struct trees tree;

 int nslice;
 struct results *sim_crit;

{
 int    iz, last_live;
 double  sumrad=0, sumradsq=0, sumht=0, sumhtsq=0,  ssrad;
 double  cross=0, slope;

for (iz=NZ-nslice; iz < NZ; iz++)
 if (whorl.rad[iz][0] !=0) {

   /* update number of live whorls */
              ++sim_crit->whorlnum;

   /* calculate sums for regression for Crown Angle */
```

```
                sumrad += whorl.rad[iz][0];   /*x coordinates */
                sumradsq += pow(whorl.rad[iz][0], 2.0);

                sumht += (NZ-iz)*DZ;          /*y coordinates */
                sumhtsq += pow( (NZ-iz)*DZ, 2.0);
                cross += whorl.rad[iz][0]*(NZ-iz)*DZ;

    /* keep track of lowest live whorl */
                last_live = (NZ-iz);
    } /* end if live whorl */

    /*slope calculations */
    ssrad = sumradsq - pow(sumrad, 2.0)/sim_crit->whorlnum;
    slope= (cross - (sumrad * sumht)/sim_crit->whorlnum) / ssrad;

    /* formula for angle */
    sim_crit->angle = 90 - floor( atan(-slope)*180 / PI );

    /* calculate crown length ratio */
    sim_crit->cr_ratio = (tree.h - (last_live-1)*DZ)/tree.h;

}
```

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

HEIGHT_INCR.C

/*----------HEIGHT_INCR-----------------------------------*/
/*     JHR   7/94
                 */

/* A function to calculate the Ht. increment criteria from
 a passed array of subject_tree structures. Note that '.ht' element
 of structure is now height at year (NYEAR -2), i.e. pre-height. */

#include "whorl.auto.h"


void regress();


void height_incr(live_tree, num_live, sim_crit)

/* HEIGHT_INCR is passed an array of subject_tree structures and calculates
  the following criteria: slope & R^2 for a regression of the 2-yr
  height increment on the NYEAR-2 ht, separately for the suppressed trees,
  (defined as having initial heights <=2.8 meters), and the dominant trees,
  (ht >=3.2 meters). */

 struct subject_tree live_tree[];
 int  num_live;
 struct results *sim_crit;
{
 int isub, sup_max, dom_min;

 sup_max = -99;
 dom_min = -99;


/* split into sub_structures for Suppressed & Dominant Trees */
 for (isub = 1; isub <= num_live; isub++)
  if ((live_tree[isub].ht <= 2.8) && (live_tree[isub+1].ht > 2.8))
                { sup_max = isub;
                  break;}
```

```
  for (isub = sup_max; isub <= num_live; isub++)
   if ((live_tree[isub].ht < 3.2) && (live_tree[isub+1].ht >= 3.2))
                { dom_min = isub+1;
                  break;}


  if ((sup_max !=-99) && (sup_max != 1))  /* enought to regress*/
   regress(live_tree, 1, sup_max, &sim_crit->sup_slope, &sim_crit->sup_r2);
  else {
   sim_crit->sup_slope = -99.0;
   sim_crit->sup_r2 = -99.0;
   }

 if ((dom_min != -99) && (dom_min != num_live)) /*enough to regress */
 regress(live_tree, dom_min, num_live, &sim_crit->dom_slope, &sim_crit->dom_r2);
 else {
   sim_crit->dom_slope = -99.0;
   sim_crit->dom_r2 = -99.0;
   }

}



/*----------------------REGRESS-------------------------------*/

void regress(live_tree, start_index, stop_index, slope_ptr, r2_ptr)

/* REGRESS is passed an array of subject_tree structures and regresses
  the ht_inc element on the ht element, only using the data in the sub-
  array ranging between the indicies start_index & stop_index.
  REGRESS returns only the slope and R^2 estimates */

struct subject_tree live_tree[];
int   start_index, stop_index;
double *slope_ptr, *r2_ptr;

{
 int              isub, sample;
double sumHt=0, sumHtsq=0, sumIncr=0, sumIncrsq=0, cross=0;
double ssHt=0, ssIncr=0;
```

```
/* calculate slope & r2 coefficient for regressing
  live_tree[].ht_inc = b0 + b1* live_tree[].ht */

for (isub=start_index; isub <= stop_index; isub++)
  {
   sumHt += live_tree[isub].ht;
   sumHtsq += pow(live_tree[isub].ht, 2.0);
   sumIncr += live_tree[isub].ht_inc;
   sumIncrsq += pow(live_tree[isub].ht_inc, 2.0);
   cross += live_tree[isub].ht * live_tree[isub].ht_inc;
  } /* end calculation of sums of squares */


sample=stop_index - start_index +1;
ssHt=sumHtsq - pow(sumHt, 2.0)/sample;
ssIncr=sumIncrsq - pow(sumIncr, 2.0)/sample;

if (ssHt !=0.0)
 *slope_ptr= (cross - (sumHt*sumIncr)/sample) / ssHt;
else *slope_ptr = -99.0;
if ( ssIncr != 0.0)
 *r2_ptr = pow(*slope_ptr, 2.0)*ssHt / ssIncr;
else *r2_ptr = -99.0;
}
```

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

KSTWO_HT.C                                        **

**  function called from crit_stand(), calculates the two sample **
**  Kolmogorov-Smirnov test statistic.    7/11/94  jhr        **
*************************************************************************/

#include "whorl.auto.h"


void kstwo_ht( data1,  n1, data2,  n2, sim_crit)

/* From 'Numerical Recipes - 2nd ed',
                indexes the arrays 1 to n. Modified to work on already
                sorted arrays & an array of structures (subject_tree).JHReynolds 6/94 */


                struct subject_tree data1[];
                struct results *sim_crit;
                double data2[];
                int n1,n2;
{

                double d,dt;
                long j1=1,j2=1;
                float d1,d2,en1,en2,en,fn1=0.0,fn2=0.0;


                en1=n1;
                en2=n2;
                d=0.0;
                while (j1 <= n1 && j2 <= n2) {
                                if ((d1=data1[j1].ht) <= (d2=data2[j2])) fn1=j1++/en1;
                                if (d2 <= d1) fn2=j2++/en2;
                                dt = (float)fabs(fn2-fn1);
                                if (dt > d) d=dt;
                }
                sim_crit->ks_stat=d;
                en=sqrt(en1*en2/(en1+en2));
                sim_crit->ks_prob=probks((en)*(d));
```

```
                /* below from troy/zabel books (?) used until Nov 95*/
                /* replaced with above from Hinrichsen's 1988 book, test 19 Nov 95*/
                /* sim_crit->ks_prob=probks((en+0.12+0.11/en)*(d)); */
}
/* (C) Copr. 1986-92 Numerical Recipes Software #13. */

#define EPS1 0.001
#define EPS2 1.0e-8
float probks (alam)
 /* from "Numerical Recipes in C, 2nd edition", called
   from kstwo - gives prob. for Kolmogorov-Smirnov Two-sample
   statistics. */

 float alam;
{
                int j;
                float a2,fac=2.0,sum=0.0,term,termbf=0.0;

                a2 = -2.0*alam*alam;
                for (j=1;j<=100;j++) {
                                term=fac*exp(a2*j*j);
                                sum += term;
                                if (fabs(term) <= EPS1*termbf || fabs(term) <= EPS2*sum)
return sum;
                                fac = -fac;
                                termbf=fabs(term);
                }
                return 1.0;
}
#undef EPS1
#undef EPS2
/* (C) Copr. 1986-92 Numerical Recipes Software #13. */
```

```
/*  Crit_out.c:  contains function to write out criteria
    results to an output file. Called after stand/open loop
    from main() in whorl.crit.c                          */

#include "whorl.auto.h"


void criteria_out(sim_crit,loop_flag, K, D, E, Dead, H, W)

struct results sim_crit;
int loop_flag;
 double *K, *D, *E, *Dead, *H, *W;

{
 FILE  *wp;


 /* LABELS */
 if (loop_flag == 0) {
  wp = fopen("crit.out","w");
  fprintf(wp,"1  K1   K2   K3   D1   D2   D3   E1   E2   E3\n");
  fprintf(wp,"   Dead1    Dead2    Dead3    Hinc    Sinc \n");
  fprintf(wp,"0  Mort   Med.   KSpr   KSst   Ssl   SR^2   Dsl   DR^2\n");
  fprintf(wp,"  #Whorl   Angle    L.Ratio\n\n");
  fclose(wp);
  }

 /* SIMULATION OUTPUT */

 wp = fopen("crit.out","a");
 fprintf(wp,"1  %2.1f  %2.1f  %2.1f  %2.1f  %2.1f  %2.1f  %2.1f  %2.1f  %2.1f\n",
                K[0], K[1], K[2],D[0],D[1],D[2],E[0],E[1],E[2]);
 fprintf(wp,"    %4.1f    %4.1f    %4.1f  %6.5f  %6.5f\n",
                Dead[0],Dead[1],Dead[2],H[1],W[1]);
 fprintf(wp,"0  %4.3d  %5.3g  %5.3f  %5.4g  %5.3g  %5.3g  %5.3g  %5.3g\n",
 /*fprintf(wp,"%4.3d  %5.3g  %5.3f  %5.4g  %5.3g  %5.3g  %5.3g  %5.3g\n",*/
               sim_crit.mortality, sim_crit.median, sim_crit.ks_prob, sim_crit.ks_stat,
                     sim_crit.sup_slope,       sim_crit.sup_r2,       sim_crit.dom_slope,
sim_crit.dom_r2);

 fprintf(wp,"  %5.3d  %6.3g   %6.3f\n\n", sim_crit.whorlnum, sim_crit.angle,
               sim_crit.cr_ratio);
```

```
fclose(wp);
}
```

```
/*********************************************************************
*                                                                    *
        • file:

OUTPUT.C                                                             *

* annotated 6/1/92 JHR                                               *
*********************************************************************/

#include "whorl.auto.h"


/*********************************************************************
*                        DECLARATION                                 *
*********************************************************************/

/*void OutGrid();*/
void OutFol();
void OutTree();
void OutHinc();
void OutHt();
void OutTfl();
void OutDead();
void OutSamp();
void OutRad();
void OutRad2();
void OutRad3();
void OutJoint();


/*********************************************************************
*                        FUNCTIONS                                   *
*********************************************************************/
/********************** OutGrid ***********************************/
/*void OutGrid (gfp, cellmid, NCx, NCy)
FILE  *gfp;
struct cell_mid cellmid;
int   NCx, NCy;
{
 double xcell, ycell;
```

```
  fprintf(gfp,"NCx= %d, NCy= %d\n", NCx, NCy);
  fputs("x ==========\n", gfp);
  while (GetNext (cellmid.x, &xcell))
   printf (gfp, "%lf\n", xcell);
  fputs("y ==========\n", gfp);
  while (GetNext (cellmid.y, &ycell))
   fprintf (gfp, "%lf\n", ycell);
  fclose(gfp);
 } */


/******************** OutFol ************************************/
void OutFol (ffp, share, foliage_den, Sam_tr, xcell, ycell, iz,
                    NSAMP, irrad)
FILE    *ffp;
SHAREPTR share;
double  foliage_den;
int     *Sam_tr;
double  xcell, ycell;
int     iz;
int             NSAMP;
double  irrad;
{
 int inbr, isam;

 for (inbr = 0; inbr < share->n; inbr++)
 {
  isam=0;
  while (isam < NSAMP)
   if (share->id[inbr] != Sam_tr[isam])
    isam++;
   else
   {
    fprintf(ffp," %d  %d  %d   %lf %lf   %lf %lf\n",
              share->id[inbr], iz, share->q[inbr],
              xcell, ycell, foliage_den,
                                  foliage_den * irrad);
    break;
   }
 }
}
```

```
/************************ OutTree ***********************************/
void OutTree (tfp, isub, Sam_tr, NSAMP, foliage, uptake, no_cells, nslice)
FILE *tfp;
double foliage[NZ][NT][NQUAD];
double uptake[NZ][NT][NQUAD];
int  isub;
int  *Sam_tr;
int  NSAMP;
int no_cells[NZ][NT][NQUAD];
int  nslice;
{
 int iz, iq, isam;

 isam=0;
 while (isam < NSAMP)
  if (isub != Sam_tr[isam])
   isam++;
  else
  {
   for (iz = NZ-nslice; iz < NZ; iz++)  /* run through all veritical cells */
   {
    fprintf (tfp, "%d    ", iz - NZ + nslice);

    for (iq = 0; iq < NQUAD; iq++)
              /* print uptake, foliage density, # cells occup. by each quad */
     fprintf (tfp, "%lf  %lf  %lf  ", uptake[iz][isub][iq],
                        foliage[iz][isub][iq],
                        (double)no_cells[iz][isub][iq]*CW*CW);
    fputs("\n", tfp);
   }

   break;
  }
}


/************************ OutHinc ***********************************/
void OutHinc (hifp, hinc, isub, Nsub)
FILE  *hifp;
double hinc;
int   isub,Nsub;
```

```
{
 fprintf (hifp, "%lf\n ", hinc);

 if (isub == Nsub) fputs("\n", hifp);
}


/***************** OutHt ************************************/
void OutHt (htfp, tree, Nsub)
FILE   *htfp;
struct trees *tree;
int    Nsub;
{
 int isub;

 for (isub = 0; isub < Nsub; isub ++)
   fprintf(htfp, "%d  %lf\n", isub, tree[isub].h);
}

/***************** OutSpace *********************************/
void OutSpace (spacefp, sumspfp, tree, Nsub, iyear)
FILE *spacefp, *sumspfp;
struct trees *tree;
int Nsub, iyear;
{
int isub=0;
int sumcubes=0;
int sumbiocubes=0;
int living=0;

double sumfolmass=0.0;


for (isub =0; isub < Nsub; isub++)
 {
  fprintf(spacefp, "%d  %lf  %d   %d   %lf\n", isub, tree[isub].h,
                              tree[isub].cubes, tree[isub].biocubes, tree[isub].folmass);

  if (tree[isub].h > 0.0)
   { /* calculate stand totals for living trees */
   sumcubes += tree[isub].cubes;
   sumbiocubes += tree[isub].biocubes;
```

```
    sumfolmass += tree[isub].folmass;
    living ++;
   }
  }
 fprintf(sumspfp, "%d %d %d %lf %d %lf %lf %lf\n",
                          iyear, living, sumcubes, (double) sumcubes/ (double)
living, sumbiocubes,
                          (double) sumbiocubes/ (double) living, sumfolmass,
sumfolmass/ (double) living);
}

/*************** OutJoint ***********************************/
void OutJoint (jfp, occupancy, iyear)
FILE *jfp;
int iyear;
struct occupy *occupancy;
{
 int index;

 for (index =1; index<=39; index++)
 if (occupancy[index].cubes!=0 || occupancy[index].biocubes!=0)
  fprintf (jfp,"%d  %d  %d  %d\n", iyear, index, occupancy[index].cubes,
        occupancy[index].biocubes);
}

/********************** OutTfl ***********************************/
void OutTfl (tffp, iyear, total_laf, total_dead, total_ht, nslice, Nsub)
FILE  *tffp;
int   iyear;
double total_laf;
int   total_dead;
double total_ht;
int   nslice;
int   Nsub;
{
 fprintf (tffp, "%3d %4d  %10lf %5d %10lf %10lf\n", iyear, nslice,
    total_laf, total_dead, total_ht, total_ht/(Nsub - total_dead));
}


/******************** OutDead ***********************************/
void OutDead (dtfp, relup, iyear, Nsub)
```

```
FILE  *dtfp;
double *relup;
int   iyear;
{
 int isub;

 for (isub = 0; isub < Nsub; isub++)
  if (relup[isub] == 0)
   fprintf(dtfp, "%3d  %3d \n",iyear, isub);
}


/******************** OutSamp ***********************************/
void OutSamp (ifp, i1, sample1, nslice)
FILE  *ifp;
int   i1;
struct cell_sample *sample1;
int   nslice;
{
 int iz;

 for (iz = NZ-nslice; iz < NZ; iz++)
  fprintf(ifp,"%d  %lf  %lf  %lf\n", iz, kmean(i1, sample1[iz].irr),
                                kmean(i1, sample1[iz].fol),
                                kmean(i1, sample1[iz].up));
}


/****************** OutRad ************************************/
void OutRad (rfp, whorl, Sam_tr, iyear, nslice, NSAMP)
FILE  *rfp;
struct whorls *whorl;
int   *Sam_tr;
int   iyear;
int   nslice;
int   NSAMP;
{
 int iz, iq, isam;

 for (isam = 0; isam < NSAMP; isam++)
  for (iz = NZ-nslice; iz < NZ; iz++)
  {
```

```
    fprintf (rfp, "%3d  %3d  %3d", iyear, Sam_tr[isam], iz-NZ+nslice);
    for (iq = 0; iq < NQUAD; iq++)
     fprintf (rfp, "   %lf", whorl[Sam_tr[isam]].rad[iz][iq]);
    fputs ("\n", rfp);
   }
}

/***************** OutRad2************************************/
void OutRad2 (rfp, whorl, iyear, nslice, NSUB)
FILE  *rfp;
struct whorls *whorl;
int   iyear;
int   nslice;
int   NSUB;
{
 int iz, iq, isam, bottom, top;
 double  radius[4];

 /* set registers */
 for (iq = 0; iq < NQUAD; iq++)
   radius[iq]=0.0;
                /* max ceiling slice is 0, counts down stack */
bottom=NZ-nslice; /* top slice is NZ-nslice*/
top = NZ;  /* bottom slice is NZ */

for (isam = 0; isam < NSUB; isam++)
 {              /* find max radii, min and max iz */
 for (iz = NZ-nslice; iz < NZ; iz++) /* walk down stack*/
 {
  for (iq = 0; iq < NQUAD; iq++){
               if (radius[iq]<whorl[isam].rad[iz][iq])
                  radius[iq]=whorl[isam].rad[iz][iq];
    if (whorl[isam].rad[iz][iq]!=0.0) {
               if (iz > bottom) bottom=iz;
    if (iz < top) top=iz;
               }
              }
 }


                                        /* re-reference crown depth */
fprintf (rfp, "%3d  %3d %3d %3d ", iyear, isam, NZ-bottom, NZ-top);
for (iq = 0; iq < NQUAD; iq++)
```

```
      fprintf (rfp, "   %lf", radius[iq]);
   fputs ("\n", rfp);

   /* reset registers */
   for (iq = 0; iq <NQUAD; iq++)
    radius[iq]=0.0;
   bottom=NZ-nslice;
   top=NZ;
  } /* move to next tree */
}
/******************** OutRad3*************************************/
void OutRad3 (r3fp, whorl, iyear, nslice, NSUB)
FILE   *r3fp;
struct whorls *whorl;
int    iyear;
int    nslice;
int    NSUB;
{
 int iz, iq, isam,quadcount,whorlcount;
 double totquadlength;


 for (isam = 0; isam < NSUB; isam++)
  {              /* reset registers*/
                 quadcount = 0;
                 whorlcount=0;
                 totquadlength=0.0;

  for (iz = NZ-nslice; iz < NZ; iz++)
  {
 if (whorl[isam].rad[iz][0]||whorl[isam].rad[iz][1] ||
                 whorl[isam].rad[iz][2]||whorl[isam].rad[iz][3])
                 whorlcount++;

 for (iq = 0; iq < NQUAD; iq++)
  if (whorl[isam].rad[iz][iq]) {
                 quadcount++;
                 totquadlength+=whorl[isam].rad[iz][iq];
                 }
 } /* next whorl */

 fprintf (r3fp, "%3d  %3d   %3d  ", iyear, NZ-nslice, isam);
```

```
    fprintf (r3fp, "%3d  %3d   %lf\n", whorlcount, quadcount, totquadlength);
  } /* move to next tree */
}
```

```
/*****************************************************************
*
        • file:

DEQUE.C                                          *

*                                                *
*****************************************************/

#include "deque.h"


/*****************************************************************************
********
*                        Definition                        *
*****************************************************************************
*******/
/****************** type ************************************/

typedef struct node NODE, *NODEPTR;
typedef struct header HEADER, *HEADERPTR;


/*****************************************************************
*                        Declaration                        *
*****************************************************************/
/*********************** variable *********************************/

static int successful;     /* TRUE if the last funtion called was successful */


/********************* structure **********************************/

struct node
{
 ELEMTYPE elem;
 NODEPTR  next;
};

struct header
{
```

```
  NODEPTR head;
  NODEPTR tail;
  NODEPTR curs;
};


/****************** local function ********************************/

static NODEPTR NewElemRec();


/*************************************************************
*                          Functions                          *
*************************************************************/

/****************** NewElemRec ********************************/

static NODEPTR NewElemRec ()
{
 NODEPTR p;

 p = (NODEPTR) calloc (1, sizeof (NODE));
 p->next = NULL;
 return (p);
}

/******************** IsOpOk **********************************/

int IsOpOk ()
{
 return (successful);
}


/****************** IsEmpty **********************************/

int IsEmpty (Deque)
HEADERPTR Deque;
{
 if (Deque->head == Deque->tail)
  return TRUE;
 else
```

```
   return FALSE;
}


/***************** Create ************************************/

HEADERPTR Create ()
{
 HEADERPTR p;

   p     = (HEADERPTR) calloc (1, sizeof (HEADER));
   p->head = NewElemRec();
   p->tail = p->head;
   p->curs = p->head;

   successful = TRUE;
   return (p);
}


/***************** InsertFront ********************************/

void InsertFront (Deque, elem)
HEADERPTR Deque;
ELEMTYPE elem;
{
 NODEPTR p;

 if (Deque == NULL)                    /* not yet created */
  successful = FALSE;
 else if (Deque->head == Deque->tail)       /* empty queue */
  InsertRear (Deque, elem);
 else
                /* non-empty deque */
 {
  p            = NewElemRec();
  p->next        = Deque->head->next;
  p->elem        = elem;
  Deque->head->next = p;

  successful = TRUE;
 }
```

```
}


/***************** InsertRear ***********************************/

void InsertRear (Deque, elem)
HEADERPTR Deque;
ELEMTYPE  elem;
{
 if (Deque == NULL)
   successful = FALSE;
 else
 {
   Deque->tail->next = NewElemRec();
   Deque->tail       = Deque->tail->next;
   Deque->tail->elem = elem;

   successful = TRUE;
 }
}


/***************** Traverse ***********************************/

void Traverse (Deque, fn)
HEADERPTR Deque;
void (* fn)();
{
 if (Deque == NULL)
   successful = FALSE;
 else
 {
   Deque->tail = Deque->head;
   while (Deque->tail->next != NULL)
   {
     Deque->tail = Deque->tail->next;
     fn (Deque->tail->elem);
   }
   successful = TRUE;
 }
}
```

```
/***************** GetNext ***********************************/

int GetNext (Deque, elem)
HEADERPTR Deque;
ELEMTYPE *elem;
{
 Deque->curs = Deque->curs->next;

 if (Deque->curs == NULL)
 {
  Deque->curs = Deque->head;
  successful = FALSE;
  return FALSE;
 }
 else
 {
  *elem = Deque->curs->elem;
  successful = TRUE;
  return TRUE;
 }
}


/************* GetFront **********************************/

ELEMTYPE GetFront (Deque)
HEADERPTR Deque;
{
 if (Deque == NULL || Deque->head == Deque->tail)
  successful = FALSE;
 else
 {
  successful = TRUE;
  return (Deque->head->next->elem);
 }
}


/***************** GetRear **********************************/

ELEMTYPE GetRear (Deque)
```

```
HEADERPTR Deque;
{
 if (Deque == NULL || Deque->head == Deque->tail)  /* not created or empty */
  successful = FALSE;
 else
 {
  successful = TRUE;
  return (Deque->tail->elem);
 }
}
/********************** DeleteFront ********************************/

void DeleteFront (Deque)
HEADERPTR Deque;
{
 NODEPTR p;

 if (Deque == NULL || Deque->head == Deque->tail)  /* not created or empty */
  successful = FALSE;
 else
 {
  if (Deque->curs == Deque->head)
   Deque->curs = Deque->head->next;

  p = Deque->head->next;
  Deque->head->next = p->next;
  free (p);

  if (Deque->head->next == NULL)
   Deque->tail = Deque->head;

  successful = TRUE;
 }
}


/********************** DeleteRear ********************************/

void DeleteRear (Deque)
HEADERPTR Deque;
{
 if (Deque == NULL || Deque->head == Deque->tail)  /* not created or empty */
```

```
    successful = FALSE;
  else
  {
    Deque->tail = Deque->head;
    while (Deque->tail->next != NULL && Deque->tail->next->next != NULL)
      Deque->tail = Deque->tail->next;

    free(Deque->tail->next);
    Deque->tail->next = NULL;

    successful = TRUE;
  }
}
```

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

- file:

```
MISC.C                                                          *

* FUNCTIONS: AutoSam, checklive, round, kmean, kstdev                    *
*****************************************************************/

#include "whorl.auto.h"
#include "globals.h"



/****************** AutoSam *********************************/
/*****
  Prints NSAMP trees to the file afp in order of ascending height.
  First sorts the temporary tree2 structure by height.

****/
void AutoSam(afp, Nsub, tree, once)
FILE *afp;  /* file pointer for output */
int  Nsub,  /* number of subject trees */
    *once; /* set to 0 at end, to assure doing this only once */
struct trees *tree;  /* tree array */
{
 int isub;          /* index of trees */
 int icho = 0;      /* index of sample trees  */
 struct trees2 tree2[NT];  /* temporary tree structure for sorting & print */

 printf("Number of trees to choose: %d\n",NSAMP);
 puts("(If not listed below, Sam_tr = 0)");  /* not sample tree */
 for(isub = 0; isub < Nsub; isub++)
 {      /* select out just index and height into temporary array */
  tree2[isub].i = isub;
  tree2[isub].h = tree[isub].h;
 }
 sort_struct(Nsub,tree2,h);     /* sort by height in ascending order */

 printf("   --- ID   height\n");
```

```
  for(isub = 0; isub < Nsub; isub++)
  {
   if ((tree2[isub].h >  Sam_ht[icho])  & (icho < NSAMP))
   {
    Sam_tr[icho] =  tree2[isub].i; /* captures smallest tree > Sam_ht[icho]*/
    icho++;
   }
  }
  for (icho = 0; icho < NSAMP; icho++)
  {
   fprintf(afp,"%d   %d  %lf\n",icho, Sam_tr[icho], tree[Sam_tr[icho]].h);
   printf("    --- %d   %lf\n",Sam_tr[icho], tree[Sam_tr[icho]].h);
  }
  *once  = 0;
}

/*-------------------- CheckLive ----------------------------------*/
/****
   If any of the sample trees have died, search through the sorted
   tree array for the next available shortest tree, and replace the
   sample tree entry with that one. Then print out the sample.
****/

void CheckLive(afp, Nsub, tree)
FILE *afp;  /* file pointer to auto-sam output file */
int Nsub;   /* number of subject trees */
struct trees *tree;  /* tree array */
{
 int isub, icho, found;      /* indices and flag */
 struct trees2 tree2[NT];    /* temporary tree array, for sorting*/

 for(isub = 0; isub < Nsub; isub++)
 {
  tree2[isub].i = isub;
  tree2[isub].h = tree[isub].h;
 }
                                        /* remove already sampled trees from consideration
*/
 for(icho = 0; icho < NSAMP; icho++)
  tree2[Sam_tr[icho]].h = 0;

 sort_struct(Nsub,tree2,h);  /* sort remaining trees in ascending ht */
```

```
  puts("New selection");
  for (icho = 0; icho < NSAMP; icho++)
  {
   if (tree[Sam_tr[icho]].h == 0)    /* loop: check for trees with 0 height */
   {
     isub = 0;
     found = 1;
     while((isub < Nsub) & (found) )              /* loop: all trees */
     {
      if ((tree2[isub].h > (Sam_ht[icho]) ))
      {
                found = 0;                    /* stop when find replacemnt */
       Sam_tr[icho] =  tree2[isub].i;
                tree2[isub].h = 0.0;           /* remove from consideration */
               }
               isub+=1;
     }
   }
  }
  for (icho = 0; icho < NSAMP; icho++)
  {
   printf("%d  %d  %lf (%lf)\n",icho, Sam_tr[icho], tree[Sam_tr[icho]].h,
              Sam_ht[icho]);
   fprintf(afp,"%d   %d  %lf\n",icho, Sam_tr[icho],
     tree[Sam_tr[icho]].h);
  }
  puts("");
}
/*-----------------------------------------------------------------*/
/***** round() ***** rounds up for dec>=.5, down otherwise */
int round(x)
double x;
{
 int roundown; double dec;

 roundown = x;
 dec = x - roundown; if (dec >= .5556) return(roundown +1);
 else return(roundown);
}
```

```
/*-----------------------------------------------------------------*/
/***** kmean() ***** takes mean of a vector */
double kmean(n,v)
int n; double v[];
{
 int i; double s=0;
 for (i = 0; i <= n; i++) s += v[i];
 return(s/(n+1));
}


/*-----------------------------------------------------------------*/
/***** kstdev() ***** standard deviation of a vector */
double sam_ckstdev(n,v)
int n; double v[Nc];
{
 int i; double s1=0, s2=0;
 for (i = 0; i <=n; i++)
 {
  s1 += pow(v[i],2.0);
  s2 += v[i];
 }
 return(pow( (s1 - pow(s2,2.0)/n+1)/n,.5));
}
```

APPENDIX B: EVOLUTIONARY OPTIMIZATION C CODE FOR GENERATING WHORL'S PARETO OPTIMAL SET

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

                EVOLVE_PARETO.H

last revised: 12/9/94

*************************************************************************/

#include <stdio.h>

#include <sys/wait.h>

#include <math.h>

#include <malloc.h>

#include <string.h>


                /*   CONSTANTS   */


#define GEN_NUM   80    /* Number of Generations to let solution evolve */

#define MUTATE_PROB  .6

#define BETA    5 /* exp. in mutate function, setting from book  */
```

```
/* Define Functions */

#define min(a,b) ((a)>(b) ? (b):(a))

#define max(a,b) ((a)>(b) ? (a):(b))

/* struct definitions */

struct assessvec {
        int mort;       /* assessment vector for simulation results*/
        int median;
        int ks_prob;
        int sup_slope;
        int sup_r2;
        int dom_slope;
        int dom_r2;
        int whorlnum;
        int angle;
```

```
        int cr_ratio;
        };

struct siminfo {                        /* Parameterization & Criteria Results */
        struct siminfo *next;

        double      K[3];
        double      D[3];
        double      E[3];
        double      Dead[3];
        double      H;
        double      W;

        int         mortality;
        double      median;
        double      ks_prob;
        double      ks_stat;
        double      sup_slope;
```

```
        double      sup_r2;
        double      dom_slope;
        double      dom_r2;
        int         whorlnum;
        double      angle;
        double      cr_ratio;

        double fitness;   /* fitness of simulation */
        };

struct pset {
        struct pset *next;

        struct assessvec assess;        /* assessment vector*/
        struct siminfo *sim;            /* ptr to param & crit.results*/
        int         count;              /* track the # param giving each assessvector*/
        double group_fitness;  /* sum of elements' fitnesses */
```

```
        };


struct dset {
        struct siminfo *sim;                    /*  ptr  to  param  &
crit.results*/
        int             count;                  /* track the # dominated
param*/
        double group_fitness;  /* sum of elements' fitnesses */
        };


struct param {
        struct param *next;
        double K[3];
        double      D[3];
        double      E[3];
        double      Dead[3];
        double      H;
        double      W;
        };
```

```
/* FUNCTION DECLARATIONS */

void evaluation();

void pareto_calc();

void read_input();

void criteria_assess();

void compare_assess();

int crit_comp();

void update_pareto();

void remove_element();

void free_rest_list();

void free_pareto();

void update_dominated();

void print_pareto();


void breed();

int no_offspring();
```

```
void select_parent();

void mutate_offspring();

void element_mutate();

void Dead_mutate();

void single_mutate();

double mutation();

void crossover_offspring();

void cross();

void permute_parents();

void free_list();
```

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

                EVOLVE_PARETO START_GEN STOP_GEN

*************************************************************************
                evolve_pareto.wh2 -> an Evolutionary Strategy program
                to optimize the Pareto Optimal Set of a model, (currently
    set for running with versions of WHORL which produce
                a crit.out file of the ten criteria).

                Application: creating the Pareto Optimal Set for
                the canopy competition model whorl.

                NOTE ON EXECUTION: evolve_pareto can be called with a nonzero
                starting generation (default is to start at gen. 0):
                e.g.                            evolve_pareto 5
                evolve_pareto can also be told to stop short of GEN_NUM (default):
                e.g.                            evolve_pareto 5 15
                starts at the 5th generation (reads 'crit.out' as generated
                from 5th round), and will stop having created the 15th gen's
                crit.out).

                Outline:
                Reads in the current crit.out file, generates the  Pareto
                Optimal set (retaining a list of the dominated parameterizations);
                calculates the FITNESS of each simulation result (criteria_assess);
                Selects PARENTS
                from each group of the Pareto Optimal Set as well as from the
                Dominated elements of the last round (based on FITNESS); the
                proportion of parents coming from the Pareto Optimal Set vrs the
                Dominated simulations increases with each generation; Apply a
                GENETIC OPERATOR to each parent (either non-uniform mutation or
                crossover between two parents); write offspring to next
                generation's input file; initiate WHORL via dist_sh, a shell
                script that distributes the simulations among a variety of machines
                in order to increase the speed of the optimization; repeat.

                Output:
                Write's out each generation's crit.out (includes input) as well as
                Pareto Optimal Set.
                Stops when number of generations exceeds GEN_NO or Pareto Optimal
                Set consists of one group achieving all criteria.
```

```
                Decisions:
                 Acceptance Bounds of Pareto Optimal Set (in the
                criteria_assess()  of pareto() in evaluate);
                FITNESS function (1 for each achieved criteria, .5 for each
                criteria with 10% of radius of boundary).
                Pmutate = 1 - Pcrossover      Pmutate set in evolve_pareto.h
                MINRANGE, MAXRANGE, and SCALE are set below and define the
search
                grid (mesh).



*************************************************************************
*               Designed, written, and implemented by JHR 3-5/95        *
*************************************************************************/

#include "evolve_pareto.h"

                                /* DEFINE SEARCH MESH */
/* K, D, E - shade, sun, switch. Dead has own function */
                /* Ranges from extremes in Phase 2 investigations */
                /* K - D - E - Switch - Height - Width */
double  MAXRANGE[6] = {0.5, 12.0, 3.0, 1.0, .00010, .0007};
double  MINRANGE[6] = {0.1, 0.0, 1.0, 0.0, .00002, .0001};
double  SCALE[6] = {0.1, 1.0, 0.5, .05, .00001, .0001};

extern void srand48();


void main(argc, argv)

 int argc; /* number of arguments passed, including program call*/
 char **argv; /* ptr to list of strings */

{
/*=================    DECLARATION    ==================*/
FILE *ip;

struct pset  *first;
struct dset *dom_list;
```

```
int generation, stop_gen, pareto_flag;
int status;       /* for calling whorl */

char critname[75]; /* buffer for renaming crit.out */

long seed = 589764;

srand48(seed);  /* seed random number generator */



                /*  Initialize Pareto Set linked list  */

first=(struct pset *) malloc(sizeof(struct pset));
first->next=NULL;     /* beginning element of Pareto Set */
bzero((char *) &first->assess, sizeof(struct assessvec));
first->count=0;
first->group_fitness=0;  /* sum of fitness for each element */
first->sim=(struct siminfo *) malloc(sizeof(struct siminfo));
bzero((char *) first->sim, sizeof(struct siminfo));
first->sim->next=NULL;

                /*  Initialize Dominated Element list  */

dom_list=(struct dset *) malloc(sizeof(struct dset));
dom_list->count=0;
dom_list->group_fitness=0;  /* sum of fitness for each element */
dom_list->sim=(struct siminfo *) malloc(sizeof(struct siminfo));
bzero((char *) dom_list->sim, sizeof(struct siminfo));
dom_list->sim->next=NULL;



              /* Check for !0 Generation Starting point
                and partial stopping point */

if (argc > 1) {
 char *a1 = argv[1];  /* read generation to start at */
 sscanf(a1, "%d", &generation);
 if (argc == 3) {
              char *a2 = argv[2]; /* read generation to stop at */
```

```
                sscanf(a2, "%d", &stop_gen);
                }
 else stop_gen =  GEN_NUM;
 }
else generation = 0;


                                /* initialize stopping flags */
pareto_flag=0;


                                /* Generation loop */

while ((generation <= stop_gen) && (pareto_flag!=1))
 {              /* continue search ~ evolution */

 ip = fopen("crit.out","r");

              /*generate Pareto Set and Dominated Lists from current results*/

 evaluation( first, dom_list, ip, &pareto_flag);

 fclose(ip);

              /* write out Pareto file */

 print_pareto(first,generation);

 sprintf(critname,"%s%d","mv crit.out crit.out",generation);
 system(critname);


 if (pareto_flag != 1)  /* continue optimization */
 {

                               /* generate offspring */
  breed( first, dom_list, generation);


             /* call shell script or WHORL to do next
                               round of simulations */
```

```
if ((status=system("dist_sh")<0)|| !WIFEXITED(status))
                {
                /* error occurred during whorl's running */
                printf(" error during dist_sh execution");
                exit(1);
    }
 } /* end of next generation's generation process */

 generation++;

                /* clear Dominated List */

 printf("# dominated %d",dom_list->count);
 free_rest_list(dom_list->sim->next);
 dom_list->sim->next=NULL;
 dom_list->count=0;
 dom_list->group_fitness=0;   /* sum of fitness for each element */
 bzero((char *) dom_list->sim, sizeof(struct siminfo));

 } /*end of generation loop*/

                /* free up memory used */

 free_rest_list(dom_list->sim);
 free(dom_list);
 free_pareto(first);

} /* end main */
```

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

 UPDATE.C

            Collection of functions, called by evolve_pareto.c, to
  read in current crit.out, calculate Pareto Set, Dominated List,
  and fitness of each simulations.

 FUNCTIONS: evaluation, pareto_calc,read_input, criteria_assess,
                  compare_assess, crit_comp, update_pareto, update_dominated,
                  remove_element, free_rest_list, print_pareto
*************************************************************************
*                          JHR fall 94 - 4/95
                  *
*************************************************************************/
#include "evolve_pareto.h"



/*************************EVALUATION*********************************/

void evaluation(first_pareto_ptr, dom_list,  input_ptr, pareto_flag_ptr)
/* main driver of evaluation & Pareto Set Update loop */

 struct pset *first_pareto_ptr;
 struct dset *dom_list;
                          /*already initialized beginning ptrs */
 FILE *input_ptr;
 int *pareto_flag_ptr;

 {
 struct pset *paretoptr;
 struct siminfo  current_sim;

 int erc;

 current_sim.next=NULL;
 erc = 0;

            /* Throw out headers */
```

```
  fscanf(input_ptr, "%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s");
  fscanf(input_ptr, "%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s%*s");

  while (!erc)
   {  /* while reading results fine */
   read_input(&erc, input_ptr, &current_sim);

   paretoptr = first_pareto_ptr;
                                        /* reset current element of Pareto Set (for
comparisons)*/


              /* UPDATE PARETO SET WITH THESE RESULTS */
   if (erc!=1) pareto_calc(first_pareto_ptr, &paretoptr, dom_list,
                                        &current_sim, pareto_flag_ptr);
  } /* read in and processed simulations */



 } /* end of evaluation() */



/*==========Pareto FUNCTIONS==========*/


/********* PARETO_CALC*****************************/

void pareto_calc(first, ptr_paretoptr, dom_list, current_sim, pareto_flag_ptr)
 /* assess current simulation, update Pareto Optimal Set */

  struct pset *first, **ptr_paretoptr;
  struct dset *dom_list;
  struct siminfo *current_sim;
  int *pareto_flag_ptr;


   {
   struct assessvec current_assess;
   int dominance, on_list, stop_loop;
```

```
/* current_sim read in, now create assessvec */
dominance = 0;                          /* Set flag: current dominated? */
on_list = 0;                /* current dominates a previous Pareto set elem */
stop_loop = 0;                          /* stop searching the Pareto Set? */

criteria_assess(current_sim, &current_assess);
            /* check if 'TOTAL SATISFACTION' of criteria */
if (current_sim->fitness == 10 ) *pareto_flag_ptr=1;

/* go through current Pareto List, compare assessvec's */
while ((*ptr_paretoptr !=NULL) && (!stop_loop))
 {
 compare_assess(current_assess, (*ptr_paretoptr)->assess, &dominance);
 update_pareto(dominance, current_assess, current_sim, ptr_paretoptr,
                                        first, dom_list, &stop_loop, &on_list);

 *ptr_paretoptr=(*ptr_paretoptr)->next;
 } /*end search of Pareto List */

if ((!on_list) && (!stop_loop))   /* current is codom with Pareto List */
 {
 *ptr_paretoptr=first;
 while ((*ptr_paretoptr)->next !=NULL)    /* find last element of list */
              *ptr_paretoptr=(*ptr_paretoptr)->next;
 (*ptr_paretoptr)->next= (struct pset *) malloc (sizeof(struct pset));
 ((*ptr_paretoptr)->next)->next=NULL;
 ((*ptr_paretoptr)->next)->assess=current_assess;
 ((*ptr_paretoptr)->next)->sim=(struct siminfo *) malloc(sizeof(struct siminfo));
 *((*ptr_paretoptr)->next)->sim= *current_sim;  /* copy contents */
 ((*ptr_paretoptr)->next)->sim->next=NULL;
 ((*ptr_paretoptr)->next)->count=1;
 ((*ptr_paretoptr)->next)->group_fitness=current_sim->fitness;
 }
} /* end of pareto_calc function*/


/**********************FUNCTIONS**********************************/
```

```
/************************READ_INPUT*************************************/

void read_input(erc_ptr, ip, current_sim_ptr)
 /* reads in current simulation's parameters & criteria results*/

 FILE *ip;
 int *erc_ptr;
 struct siminfo *current_sim_ptr;

 {
  if (fscanf(ip, "%*d %lf %lf %lf", &current_sim_ptr->K[0],
                    &current_sim_ptr->K[1], &current_sim_ptr->K[2]) != 3 )
                          *erc_ptr = 1;
  else if (fscanf(ip, "%lf %lf %lf", &current_sim_ptr->D[0],
                     &current_sim_ptr->D[1], &current_sim_ptr->D[2]) != 3)
                          *erc_ptr = 1;
  else if (fscanf(ip, "%lf %lf %lf", &current_sim_ptr->E[0],
                          &current_sim_ptr->E[1], &current_sim_ptr->E[2]) != 3)
                          *erc_ptr =1;
  else if (fscanf(ip, "%lf %lf %lf", &current_sim_ptr->Dead[0],
                               &current_sim_ptr->Dead[1],   &current_sim_ptr-
>Dead[2]) != 3)
                          *erc_ptr =1;
  else if (fscanf(ip, "%lf %lf", &current_sim_ptr->H,
                          &current_sim_ptr->W) != 2)
                          *erc_ptr =1;
  else if (fscanf(ip, "%*d %d %lf %lf", &current_sim_ptr->mortality,
                               &current_sim_ptr->median,   &current_sim_ptr-
>ks_prob) != 3)
                          *erc_ptr =1;
  else if (fscanf(ip, "%lf %lf %lf", &current_sim_ptr->ks_stat,
                              &current_sim_ptr->sup_slope,   &current_sim_ptr-
>sup_r2) != 3)
                          *erc_ptr =1;
  else if (fscanf(ip, "%lf %lf %d", &current_sim_ptr->dom_slope,
                               &current_sim_ptr->dom_r2,   &current_sim_ptr-
>whorlnum) != 3)
                          *erc_ptr =1;
  else if (fscanf(ip, "%lf %lf", &current_sim_ptr->angle,
                          &current_sim_ptr->cr_ratio) != 2)
                          *erc_ptr =1;
  else current_sim_ptr->fitness = 0.0;
```

```
  } /* end read_input */


/************************CRITERIA_ASSESS*******************************/

void criteria_assess(simresults, assessptr)
/* a function to create the assessment vector from a vector of
   criteria results. Criteria based on General's Writeup - JHR 3/94
   NOTE: Ht. Incr. Slope Bounds Updated 11/16/94:
                  beta +/- 3*sigma(beta)
                 */

/* FITNESS: if within bounds -> 1,
                  if within 10% (of radius) of bounds -> .5 */

struct siminfo *simresults;
struct assessvec *assessptr;

{
 if ((simresults->mortality <= 137) && (simresults->mortality >= 77))
   { assessptr->mort = 1;
     simresults->fitness++;}
 else {
   if ((simresults->mortality <= 140) && (simresults->mortality >= 74))
     simresults->fitness +=.5;
   assessptr->mort = 0;}

 if ((simresults->median <= 3.35) && (simresults->median >= 2.82))
   {assessptr->median = 1;
    simresults->fitness++;}
 else{
   if ((simresults->median <= 3.38) && (simresults->median >= 2.79))
     simresults->fitness +=.5;
   assessptr->median = 0;}

 if (simresults->ks_prob >= .01)
   { assessptr->ks_prob = 1;
    simresults->fitness++;}
 else{
   if (simresults->ks_prob>=.009) simresults->fitness +=.5;
    assessptr->ks_prob = 0;}
```

```
if ((simresults->sup_slope <= .046) && (simresults->sup_slope >= .014))
  {assessptr->sup_slope = 1;
   simresults->fitness++;}
else{
  if ((simresults->sup_slope <= .048) && (simresults->sup_slope >= .012))
     simresults->fitness +=.5;
  assessptr->sup_slope = 0;}

if ((simresults->sup_r2 <= .44) && (simresults->sup_r2 >= .04))
  {assessptr->sup_r2 = 1;
   simresults->fitness++;}
else{
  if ((simresults->sup_r2 <= .46) && (simresults->sup_r2 >= .02))
     simresults->fitness +=.5;
  assessptr->sup_r2 = 0;}

if ((simresults->dom_slope <= .085) && (simresults->dom_slope >= -.005))
  {assessptr->dom_slope = 1;
   simresults->fitness++;}
else{
  if ((simresults->dom_slope <= .09) && (simresults->dom_slope >= -.01))
     simresults->fitness +=.5;
  assessptr->dom_slope = 0;}

if ((simresults->dom_r2 <= .27) && (simresults->dom_r2 >= .0))
  {assessptr->dom_r2 = 1;
   simresults->fitness++;}
else{
  if ((simresults->dom_r2 <= .29) && (simresults->dom_r2 >= .0))
     simresults->fitness +=.5;
  assessptr->dom_r2 = 0;}

if ((simresults->whorlnum <= 17) && (simresults->whorlnum >= 9))
  {assessptr->whorlnum = 1;
   simresults->fitness++;}
else{
  if ((simresults->whorlnum <= 18) && (simresults->whorlnum >= 8))
     simresults->fitness +=.5;
  assessptr->whorlnum = 0;}

if  ((simresults->angle <= 15) && (simresults->angle >= 10))
  {assessptr->angle = 1;
```

```
   simresults->fitness++;}
 else{
  if ((simresults->angle <= 16) && (simresults->angle >= 9))
     simresults->fitness +=.5;
  assessptr->angle = 0;}

 if  (simresults->cr_ratio >= .9)
  {assessptr->cr_ratio = 1;
   simresults->fitness++;}
 else{
  if (simresults->cr_ratio >= .85)
     simresults->fitness +=.5;
  assessptr->cr_ratio = 0;}

 } /* end criteria_assess() */


/**********************COMPARE_ASSESS************************/

 void compare_assess(simassess, paretoassess, dominanceptr)
/* This function compares to assessment vectors, returning
                0: equal vectors
                1: simassess dominates
                2: paretoassess dominates
                3: codom */

int *dominanceptr;
struct assessvec paretoassess, simassess;

{
 *dominanceptr=0;
 *dominanceptr += crit_comp(simassess.mort,paretoassess.mort);
 *dominanceptr += crit_comp(simassess.median,paretoassess.median);
 *dominanceptr += crit_comp(simassess.ks_prob,paretoassess.ks_prob);
 *dominanceptr += crit_comp(simassess.sup_slope,paretoassess.sup_slope);
 *dominanceptr += crit_comp(simassess.sup_r2,paretoassess.sup_r2);
 *dominanceptr += crit_comp(simassess.dom_slope,paretoassess.dom_slope);
 *dominanceptr += crit_comp(simassess.dom_r2,paretoassess.dom_r2);
 *dominanceptr += crit_comp(simassess.whorlnum,paretoassess.whorlnum);
 *dominanceptr += crit_comp(simassess.angle,paretoassess.angle);
 *dominanceptr += crit_comp(simassess.cr_ratio,paretoassess.cr_ratio);
```

```
 if (*dominanceptr > 0 ) *dominanceptr=2;
 if ((*dominanceptr < 0) && !(*dominanceptr%10)) *dominanceptr=1;
 if ((*dominanceptr < 0) && ((*dominanceptr%10)!=0)) *dominanceptr=3;
 /* if *dominanceptr==0, then keep it */
}


/**********************CRIT_COMP**********************************/

int crit_comp(Acrit, Bcrit)

int Acrit, Bcrit;
{ int x;
 if (Acrit == Bcrit) x = 0;
 if (Acrit < Bcrit) x = 1;
 if (Acrit > Bcrit) x =  -10;
 return(x);
}


/*********************UPDATE_PARETO*********************************/

void update_pareto (dominance, current_assess, current_sim_ptr, ptr_paretoptr,
                                                  first, dom_list, stop_loop_ptr, on_list_ptr)

/* updates the Pareto Set list, except for adding new elements on end */

int             dominance, *stop_loop_ptr, *on_list_ptr;
struct assessvec                current_assess;
struct siminfo                  *current_sim_ptr;
struct pset                     **ptr_paretoptr, *first;
struct dset                     *dom_list;

{
struct siminfo *new;
struct pset *temp;



if (dominance == 0)             /* current_assess equals paretoptr->assess */
 { new = (struct siminfo *) malloc(sizeof(struct siminfo));
  *new = *current_sim_ptr;
```

```
  new->next = (*ptr_paretoptr)->sim;
  (*ptr_paretoptr)->sim = new;
  (*ptr_paretoptr)->count +=1;
  (*ptr_paretoptr)->group_fitness +=new->fitness;
  *on_list_ptr=1;
 }

if ((dominance == 1) && (*on_list_ptr == 1))  /* current dominates, >=2nd time*/
 {
            /* so need to move element from Pareto list to Dominated list*/
 update_dominated(&dom_list, (*ptr_paretoptr)->sim,
            (*ptr_paretoptr)->count, (*ptr_paretoptr)->group_fitness,0);

            /*reset Pareto list - find prior's next & reset to current next */
 temp=first;
 while (temp->next!=*ptr_paretoptr) temp=temp->next;
 temp->next=(*ptr_paretoptr)->next;
 }


if ((dominance == 1) && (*on_list_ptr == 0)) /* current dominates, first time*/
 {
  (*ptr_paretoptr)->assess = current_assess;  /* replace assess contents */
  if ((*ptr_paretoptr)->count!=0)  /* if there is something to move */
    { update_dominated(&dom_list, (*ptr_paretoptr)->sim,
                        (*ptr_paretoptr)->count,          (*ptr_paretoptr)-
>group_fitness,0);

              /* start new pareto list element */
      new = (struct siminfo *) malloc(sizeof(struct siminfo));
      *new = *current_sim_ptr;  /* copy contents */
      new->next=NULL;        /* change nextptr */
      (*ptr_paretoptr)->sim = new;
      (*ptr_paretoptr)->count =1;
      (*ptr_paretoptr)->group_fitness =new->fitness;
    } /* had to move old & create new sim struct */
  else /*nothing to move, empty simstruct already in place */
            {
             *(*ptr_paretoptr)->sim=*current_sim_ptr;   /*copy contents */
             (*ptr_paretoptr)->count = 1;
             (*ptr_paretoptr)->group_fitness = current_sim_ptr->fitness;
    }
```

```
   *on_list_ptr=1; /* set flag, current has already dominated something*/
  }


 if (dominance == 2)                          /* current is dominated */
  {
   update_dominated(&dom_list,current_sim_ptr, 1, current_sim_ptr->fitness,1);
   *stop_loop_ptr = 1;
  }

 /* if (dominance == 3) -> codom, add to end of list in main */
} /* end update_pareto */



/*****************REMOVE_ELEMENT*************************/

void remove_element(ptr_paretoptr, first)
struct pset **ptr_paretoptr, *first;

{
 struct pset *prior;

 prior = first;                               /* start at top of list*/

 while (prior->next != *ptr_paretoptr)
   prior=prior->next;                                  /* find prior element */

 prior->next = (*ptr_paretoptr)->next;   /* reset it's next, deleting p..ptr */
 free(*ptr_paretoptr);           /* release that chunk of memory */
 *ptr_paretoptr=prior;           /* reset current address looked at,
                                                        so next is correct next in
main */

}


/*********************FREE_REST_LIST*********************************/

void free_rest_list(start_simptr)
/* runs down a siminfo list and frees all the memory */
```

```
struct siminfo *start_simptr;  /*beginning address of 'deletion' list*/
{
 struct siminfo *simptr;

 if (start_simptr!=NULL)   /* something to delete */
  {
  while (start_simptr->next!=NULL) /* more than this element to delete */
   {
    simptr=start_simptr;     /* start at beginning of 'deletion' list */
     while (simptr->next->next!=NULL)
       simptr=simptr->next;   /* find ptr to last element */
    free(simptr->next);              /* free last element */
    simptr->next = NULL;      /* make new end */
   } /* while: repeat process */

  free(start_simptr);       /* delete first element */
  } /* end if*/
} /* end free_rest_list */


/**********************FREE_PARETO**********************************/

void free_pareto(start_pset)
                /* frees up the memory of the Pareto List */
 struct pset *start_pset;

 {
  struct pset *temp;

  if (start_pset != NULL)
   {
    while (start_pset->next != NULL)  /* more elements to delete*/
    {
    temp=start_pset;              /*find ptr to last element */
    while(temp->next->next!=NULL) temp=temp->next;
    free_rest_list(temp->next->sim); /* remove sim list */
    free(temp->next);
    temp->next = NULL;
    } /* while:  repeat process */
    free_rest_list(start_pset->sim);
    free(start_pset);
   }
```

```
} /* end free_pareto */


/*********************UPDATE_DOMINATED*****************************/

void update_dominated( dom_list_ptr, sim_list_ptr, sim_list_count,
                                                sim_list_fitness,currentflag)

 struct dset **dom_list_ptr;
 struct siminfo *sim_list_ptr;

 int sim_list_count, currentflag;
 double sim_list_fitness;

 {
  struct siminfo *endptr, *temp;
  struct dset *start_dom;

  start_dom=*dom_list_ptr;

  endptr = sim_list_ptr;
  while (endptr->next != NULL) endptr = endptr->next;

  if (start_dom->count == 0)  /* blank list */
    {
    *start_dom->sim = *sim_list_ptr;   /* copy contents */
     start_dom->count = sim_list_count;
     start_dom->group_fitness = sim_list_fitness;
    }

  else
    { /* dominated list has elements already */
     if (currentflag !=1)  /* reassigning elements from Pareto list */
     {
      endptr->next = start_dom->sim;   /*attach new elements to beginning */
      start_dom->sim = sim_list_ptr;   /* reset beginning */
      start_dom->count += sim_list_count;
      start_dom->group_fitness += sim_list_fitness;
     }
     else   /* adding current simulation, need to add memory */
```

```
    {
    temp=(struct siminfo *) malloc(sizeof(struct siminfo));
    *temp = *sim_list_ptr;
    temp->next = start_dom->sim;  /* attach new memory to beginning */
    start_dom->sim = temp;        /* reset beginning */
    start_dom->count += sim_list_count;
    start_dom->group_fitness +=sim_list_fitness;
    }

   }
  *dom_list_ptr=start_dom;
 } /* end update_dominated */


/******************PRINT_PARETO*********************************/


void print_pareto( first, counter)
 /* prints out Pareto Set - elements, simulations giving those elements */

struct pset *first;
int counter;

{
FILE *op;
struct pset *paretoptr;
struct siminfo *temp;
char  paretofile[13];

sprintf(paretofile,"%s%d", "Pareto.set",counter);
op=fopen(paretofile,"w");
fprintf(op,"Pareto Set: \n");
paretoptr=first;
while (paretoptr !=NULL)   /* run down list of Pareto elements */
{
temp = paretoptr->sim;

fprintf(op,"==========================================================
===\n");
 fprintf(op,"              mort med. ks. Ssl Sr2 Dsl Dr2 W# ang. Cr.R\n");
 fprintf(op,"Assessment Vector:");
 fprintf(op," %d  %d   %d  %d  %d  %d  %d  %d  %d  %d\n\n",
```

```
                paretoptr->assess.mort, paretoptr->assess.median,
                paretoptr->assess.ks_prob, paretoptr->assess.sup_slope,
                paretoptr->assess.sup_r2, paretoptr->assess.dom_slope,
                paretoptr->assess.dom_r2, paretoptr->assess.whorlnum,
                paretoptr->assess.angle, paretoptr->assess.cr_ratio);
  fprintf(op,"Simulations: %d\n",paretoptr->count);
  fprintf(op,"1  K1   K2   K3   D1   D2   D3   E1   E2   E3\n");
  fprintf(op,"   Dead1   Dead2   Dead3   Hinc   Sinc \n");
  fprintf(op,"0  Mort  Med.  KSpr  KSst  Ssl  SR^2  Dsl DR^2\n");
  fprintf(op,"  #Whorl  Angle  L.Ratio\n\n");
  while (temp!=NULL)  /* run down simulations list */
   {
   fprintf(op,"1  %2.1f",temp->K[0]);
   fprintf(op,"  %2.1f",temp->K[1]);
   fprintf(op,"  %2.1f",temp->K[2]);
   fprintf(op,"  %2.1f",temp->D[0]);
   fprintf(op,"  %2.1f",temp->D[1]);
   fprintf(op,"  %2.1f",temp->D[2]);
   fprintf(op,"  %2.1f",temp->E[0]);
   fprintf(op,"  %2.1f",temp->E[1]);
   fprintf(op,"  %2.1f\n",temp->E[2]);
   fprintf(op,"    %4.1f",temp->Dead[0]);
   fprintf(op,"   %4.1f",temp->Dead[1]);
   fprintf(op,"   %4.1f",temp->Dead[2]);
   fprintf(op,"   %6.5f",temp->H);
   fprintf(op,"   %6.5f\n",temp->W);
   fprintf(op,"0  %4.3d",temp->mortality);
   fprintf(op,"  %5.3g", temp->median);
   fprintf(op,"  %5.3f",temp->ks_prob);
   fprintf(op,"   %5.4g", temp->ks_stat);
   fprintf(op,"  %5.3g", temp->sup_slope);
   fprintf(op,"   %5.3g", temp->sup_r2);
   fprintf(op,"  %5.3g", temp->dom_slope);
   fprintf(op,"  %5.3g\n", temp->dom_r2);
   fprintf(op,"  %5.3d", temp->whorlnum);
   fprintf(op,"  %6.3g", temp->angle);
   fprintf(op,"   %6.3f\n\n", temp->cr_ratio);
   temp=temp->next;  /* move to next element */
  } /* end simulations list */
paretoptr=paretoptr->next;  /* move to next Pareto element */
}/* end Pareto list */
```

```
fprintf(op,"==========================================================
====\n");
 fclose(op);
 } /* end print_pareto() */
```

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

                BREED.C

  File contains functions called by evolve_pareto during
  generation of the next set of parameterizations.

  Basically, the number of parents from the Pareto Optimal Set,
  and, by complement, the Dominated List, are calculated as a
  function of the generation (no_offspring). Two parameterizations are
  chosen from each Pareto Group, the rest of paroff parents are
  selected randomly from the Pareto Set, and domoff parents from the
  Dominated List.
  When a parent is selected (select_parent), it is randomly decided
  if the offspring will be from mutation or crossover.
  After all parents are selected, the list of mutational parents
  (mutate_list) are mutated (single parameter setting changed); the
  crossover parents are randomly sorted and crossed, walking down the list
  in pairs (cross).
  In the end, the memory used in the mutate_list and crossover_list are
  freed.


  functions:  breed, no_offspring, select_parent, mutate_offspring,
                                element_mutate, single_mutate, mutation,
                                crossover_offspring, cross, permute_parents,
                                free_list.
*************************************************************
element_mutation() edited 15 Nov 95 - force mutation to
occur on sun or shade setting if they are equal (don't waste
time changing the switch setting).

Dead_mutate() search ranges changed 15 Nov 95.

Odds of mutating Dead parameter increased 22 Nov 95, in
Mutate_offspring.
*************************************************************/


#include "evolve_pareto.h"

extern double MAXRANGE[6];
```

```
extern double MINRANGE[6];
extern double SCALE[6];
extern double drand48();
extern double floor();


/****************** BREED *********************************/

void breed(first, dom_list, generation)

 struct pset *first;
 struct dset *dom_list;

 int generation;

 {
 struct pset *temp;
 struct param *mutate_list, *temp_mutate, *temp_crossover,  *crossover_list;

 FILE *op;
 char par_in_file[18];

 int paroff, domoff, i, j;
 double total_fit, cum_fitness, rand;


                              /* Initialize variables */
 mutate_list = (struct param *) malloc(sizeof(struct param));
 mutate_list->next = NULL;
 /* CHECK THAT THIS IS OKAY */
 bzero((char *) mutate_list, sizeof(struct param));

 crossover_list = (struct param *) malloc(sizeof(struct param));
 crossover_list->next = NULL;
 bzero((char *) crossover_list, sizeof(struct param));

 temp_mutate = mutate_list;  /* set current ptr at beginning */
 temp_crossover = crossover_list;   /*  "   "  */
```

```
/*calculate no. of parents to select from each Set*/
paroff = no_offspring(first,generation);
domoff = 75 - paroff;

if (dom_list->count == 0 )
  { paroff = 75;
            domoff = 0;
  }

temp=first;       /* calculate total fitness of Pareto Set */
cum_fitness = temp->sim->fitness;
total_fit = 0.0;
while (temp != NULL)
            {total_fit += temp->sim->fitness;
             temp = temp->next;
            }

           /*************** SELECT PARENTS ***************/
/*select 2 parents per Pareto Group */
temp=first;
while (temp!=NULL)
 {
 rand = drand48();
 select_parent(temp->sim,temp->group_fitness, rand,
                                        &temp_mutate, &temp_crossover);

 rand = drand48();
 select_parent(temp->sim,temp->group_fitness, rand,
                                        &temp_mutate, &temp_crossover);

 temp = temp->next;
 paroff -= 2;
 }

i = 1;
while (i <= paroff)
 { /* randomly select Pareto parents based on assessment fitness */
 rand = drand48();

 temp = first;
 if (temp->sim->fitness >= rand*total_fit)
    select_parent(temp->sim, temp->group_fitness, drand48(),
```

```
                                        &temp_mutate, &temp_crossover);
  else
   {                            /* not first group */
    while (cum_fitness <= rand*total_fit && ((cum_fitness +
                                temp->next->sim->fitness) <= rand*total_fit))
     {
                cum_fitness += temp->next->sim->fitness;
                temp = temp->next;
               }
    select_parent(temp->next->sim, temp->next->group_fitness, drand48(),
                                        &temp_mutate, &temp_crossover);
   }
  i++;
  cum_fitness = first->sim->fitness;  /* reset cum_fitness */
 }


j = 1;
 while ( j <= domoff )
  { /* randomly select Dominated parents based on fitness */
   rand = drand48();
   select_parent(dom_list->sim, dom_list->group_fitness, rand,
                                        &temp_mutate, &temp_crossover);
   j++;
  }

              /***** APPLY GENETIC OPERATORS *****/

/* sprintf(par_in_file, "%s%d", "par.input.test",generation); */
op = fopen("par.input.test","w");
fprintf(op," K1  K2  Ksw  D1  D2  Dsw  E1  E2  Esw\n");
fprintf(op," Deadmin  Deadint  Deadsl  Hinc  Sinc \n");
fclose(op);
op = fopen("par.input.test","a");


if (mutate_list->H != 0 ) /* if there is something to mutate */
              mutate_offspring(mutate_list,op, generation);

                          /* check to see if there is something to cross */
if (crossover_list->H != 0 && crossover_list->next->H != 0)
              crossover_offspring(crossover_list, op);
```

```
  fclose(op);

  /* free mutate_list, crossover_list memories */
  free_list(mutate_list);
  free_list(crossover_list);

 } /* end breed() */
```

```
/*************************** NO_OFFSPRING ************************/

int no_offspring(first, generation)
                /* calculate number of offpsring to select from Pareto Set
                list */
  struct pset *first;
  int generation;

  {
   int count_available, count_desire;
   struct pset *temp;

   count_available = 0;

   temp = first;

   while (temp != NULL)
    {
                                          /* take at least 2 from each Pareto Group */
    count_available +=  max(2, temp->count);
    temp = temp->next;
    }

  if (generation <= 30) count_desire = 35 + generation;
  else count_desire = 65;

  return (min( count_desire, count_available));
```

```
}


/***************************** SELECT_PARENT ************************/

void select_parent( sim_ptr, group_fitness, prob,
                                        temp_mutate_ptr, temp_crossover_ptr)
            /* select parent from Simulation group sim_ptr, based on fitness */

 struct siminfo *sim_ptr;
 struct param **temp_mutate_ptr, **temp_crossover_ptr;

 double prob, group_fitness;

 {
  struct siminfo *temp, *parent;
  struct param *temp_mutate, *temp_crossover;
  double rand_fitness, cum_fitness;
  double example[3];

  temp_mutate=*temp_mutate_ptr;
  temp_crossover=*temp_crossover_ptr;
  temp = sim_ptr;
  cum_fitness = temp->fitness;
  rand_fitness = prob * group_fitness;


  if (cum_fitness >= rand_fitness)   /* select first sim. */
    parent = temp;
  else {
            while ((cum_fitness < rand_fitness) &&
                        ((cum_fitness + temp->next->fitness) <= rand_fitness))
      {
            cum_fitness += temp->next->fitness;
            temp = temp->next;
      }
     parent = temp->next;
    }

  /* parent selected, now select Genetic Operator */
```

```
prob = drand48();
if (prob <= MUTATE_PROB )   /*mutate parent */
 {
 if (temp_mutate->H == 0.0)  /*first parent */
  {
   bcopy((char *) parent->K, (char *) temp_mutate->K, sizeof(example));
   bcopy((char *) parent->D, (char *) temp_mutate->D, sizeof(example));
   bcopy((char *) parent->E, (char *) temp_mutate->E, sizeof(example));
   bcopy((char *) parent->Dead, (char *) temp_mutate->Dead, sizeof(example));
              temp_mutate->H = parent->H;
              temp_mutate->W = parent->W;
  }
 else /* list already exist, add memory */
  {
              temp_mutate->next = (struct param *) malloc(sizeof(struct param));
              temp_mutate=temp_mutate->next;
              temp_mutate->next = NULL;
   bcopy((char *) parent->K, (char *) temp_mutate->K, sizeof(example));
   bcopy((char *) parent->D, (char *) temp_mutate->D, sizeof(example));
   bcopy((char *) parent->E, (char *) temp_mutate->E, sizeof(example));
   bcopy((char *) parent->Dead, (char *) temp_mutate->Dead, sizeof(example));
   temp_mutate->H = parent->H;
   temp_mutate->W = parent->W;
  }
 } /* end mutate list addition */
else /* add parent to crossover list */
 {
 if (temp_crossover->H == 0.0)  /*first parent */
  {
   bcopy((char *) parent->K, (char *) temp_crossover->K, sizeof(example));
   bcopy((char *) parent->D, (char *) temp_crossover->D, sizeof(example));
   bcopy((char *) parent->E, (char *) temp_crossover->E, sizeof(example));
   bcopy((char *) parent->Dead, (char *) temp_crossover->Dead, sizeof(example));
              temp_crossover->H = parent->H;
              temp_crossover->W = parent->W;
  }
 else /* list already exist, add memory */
  {
             temp_crossover->next = (struct param *) malloc(sizeof(struct param));
             temp_crossover=temp_crossover->next;
             temp_crossover->next = NULL;
  bcopy((char *) parent->K, (char *) temp_crossover->K, sizeof(example));
```

```
      bcopy((char *) parent->D, (char *) temp_crossover->D, sizeof(example));
      bcopy((char *) parent->E, (char *) temp_crossover->E, sizeof(example));
      bcopy((char *) parent->Dead, (char *) temp_crossover->Dead, sizeof(example));
      temp_crossover->H = parent->H;
      temp_crossover->W = parent->W;
     }
    } /* end crossover list addition */


   *temp_mutate_ptr=temp_mutate;
   *temp_crossover_ptr=temp_crossover;
  } /* end of select_parent */



/************************ MUTATE_OFFSPRING ********************/

void mutate_offspring(mutate_ptr, op, generation)
   struct param *mutate_ptr;
   FILE *op;
   int generation;

  {
  struct param *temp;
  double rand;
  int choice;

  temp = mutate_ptr;

  while (temp != NULL)
   {    /* pick parameter and mutation value of parent */

    rand = drand48();
   /*  rand = rand*6; */
                /* Changed 22 Nov. 95 to focus on Dead_mutate */
                /* 2/7 chance mutate Dead parameter now */
   rand = rand*7;
   choice = (int) floor(rand);

   switch (choice)
    {
                case 1:  /* mutate element of K parameter */
```

```
                element_mutate(temp->K,0, generation, 0);
                break;

        case 2:  /* mutate element of D parameter */
                element_mutate(temp->D,1, generation, 0);
                break;

        case 3:  /* mutate element of E parameter */
                element_mutate(temp->E,2, generation, 1);
                break;

        case 4:  /* mutate W */
                single_mutate(&temp->W, 5, generation);
                break;

        case 5:  /* mutate H */
                single_mutate(&temp->H, 4, generation);
                break;

        default:  /* mutate element of Dead parameter */
                Dead_mutate(temp->Dead, generation);
                break;
      }

     /* write out mutation */

fprintf(op," %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f\n",
                temp->K[0], temp->K[1], temp->K[2], temp->D[0], temp->D[1],
                temp->D[2], temp->E[0], temp->E[1], temp->E[2]);
 fprintf(op,"   %4.1f   %4.1f   %4.1f  %6.5f  %6.5f\n\n",
                temp->Dead[0], temp->Dead[1], temp->Dead[2], temp->H, temp->W);


    temp = temp->next;
 } /* end while */

} /* end mutate_offspring */
```

```
/************************ ELEMENT_MUTATE ***********************/

void element_mutate (elem, range_index, generation, Eflag)
 /* A function to select a single parameter from a three parameter
    element (K, D, E), and mutate it. */


 double elem[3];                /* 0 - shade, 1 - sun, 2 - switch */
 int range_index, generation, Eflag;

 {
 int index, sign, choice, try1, try2, ok, redo;
 double temp[3];
 double downrange, uprange;

 /* initialize flags and choice registers */
 try1 = 4;
 try2 = 4;
 redo = 1;
 ok = 0;
 if (Eflag) {
  try1 = 0;
  temp[1] = elem[0];
  temp[0] = elem[1];  /* sun is smallest for E */
  temp[2] = elem[2];
   }
 else {
  temp[0] = elem[0];
  temp[1] = elem[1];
  temp[2] = elem[2];
   }

/* loop through until get a successful mutation */

if (temp[0] == temp[1]) try1 = 2;
                /* make sure don't change switch if sun == shade */
while (!ok) {
 /* pick element */
 while (redo) {
   choice = (int) floor(drand48()*3);
   if ((choice!=try1) && (choice!=try2)) redo=0;
   /* else pick again */
```

```
  }

 switch (choice)
   {
              case 0:  /* Shade Foliage */
     downrange = temp[0] - MINRANGE[range_index];
     uprange = temp[1] - temp[0];
               index = range_index;
               break;

              case 1:  /* Sun Foliage */
     downrange = temp[1] - temp[0];
     uprange = MAXRANGE[range_index] - temp[1];
               index = range_index;
               break;

             case 2:  /* Switch */
     downrange = temp[2];
     uprange = 1.00  - temp[2];
               index = 3;
               break;
   }           /* end calculating ranges */

 sign = (int) floor(drand48()*2);
 if (!sign){  /* attempt decreasing mutation */
   if (downrange>0.0) {
    temp[choice] -= mutation(generation,downrange,SCALE[index]);
    ok = 1;
    }
   else if (uprange>0.0) {  /* consolation attempt */
    temp[choice] += mutation(generation,uprange,SCALE[index]);
    ok = 1;
    }
      else {   /* if can do neither */
       try2 = try1;
       try1 = choice;
       redo = 1;
          }
   } /* end !sign */
 else {
   if (uprange>0.0) {  /* attempt increasing mutation */
    temp[choice] += mutation(generation,uprange,SCALE[index]);
```

```
      ok = 1;
      }
     else if (downrange>0.0) { /* consolation attempt */
      temp[choice] -= mutation(generation,downrange,SCALE[index]);
      ok = 1;
      }
      else {  /* if can do neither */
       try2 = try1;
       try1 = choice;
       redo = 1;
       }
    } /* end mutation attempts with this choice */
  } /* ok? */


  if (Eflag) {
   elem[0] = temp[1];
   elem[1] = temp[0];
   elem[2] = temp[2];
  }
  else {
   elem[0] = temp[0];
   elem[1] = temp[1];
   elem[2] = temp[2];
   }

  } /* end element_mutation */


/**********************DEAD_MUTATE ***************************/

void Dead_mutate (elem,  generation )
 /* A function to select and mutate a single parameter from
    the Dead function's parameter array. Dead, branch maintenance
    cost is currently (5/95) modelled by whorl2 as a linear
                function with a minimum. */
/* min & max of search ranges changed 15 NOV 1995 */


 double elem[3];              /* 0 - minimum, 1 - intercept, 2 - slope */
```

```
int  generation;

{
int sign, choice, try1, try2, ok, redo;
double temp[3];
double downrange, uprange, scale;

/* initialize flags and choice registers */
try1 = 4;
try2 = 4;
redo = 1;
ok = 0;

temp[0] = elem[0];
temp[1] = elem[1];
temp[2] = elem[2];

/* loop through until get a successful mutation */
while (!ok) {
 /* pick element */
 while (redo) {
  choice = (int) floor(drand48()*3);
  if ((choice!=try1) && (choice!=try2)) redo=0;
  /* else pick again */
  }

 switch (choice)  /* SET RANGE LIMITS BELOW */
   {
              case 0:  /* Minimum Cost */
    downrange = temp[0] - 0.0;
    uprange = 15.0 - temp[0];
               scale = 1.0;
               break;

             case 1:  /* Intercept */
    downrange = temp[1] - (-10.0);
    uprange = temp[0]  - temp[1]; /* def. minimum cost */
               scale = 1.0;
               break;

            case 2:  /* Slope */
    downrange = temp[2];  /* minimum slope is zero */
```

```
      uprange = 80.0  - temp[2];
                scale = 1.0;
                break;
    }           /* end calculating ranges */

 sign = (int) floor(drand48()*2);
 if (!sign){  /* attempt decreasing mutation */
   if (downrange>0.0) {
    temp[choice] -= mutation(generation,downrange,scale);
    ok = 1;
    }
   else if (uprange>0.0) {  /* consolation attempt */
    temp[choice] += mutation(generation,uprange,scale);
    ok = 1;
    }
      else {   /* if can do neither */
       try2 = try1;
       try1 = choice;
       redo = 1;
          }
   } /* end !sign */
 else {
   if (uprange>0.0) {  /* attempt increasing mutation */
    temp[choice] += mutation(generation,uprange,scale);
    ok = 1;
    }
   else if (downrange>0.0) { /* consolation attempt */
    temp[choice] -= mutation(generation,downrange,scale);
    ok = 1;
    }
    else {   /* if can do neither */
     try2 = try1;
     try1 = choice;
     redo = 1;
     }
  }  /* end mutation attempts with this choice */
} /* ok? */


elem[0] = temp[0];
elem[1] = temp[1];
elem[2] = temp[2];
```

```
} /* end Dead_mutate */


/********************** SINGLE_MUTATE ************************/

void single_mutate( param_ptr, range_index, generation)
 /* a function to calculate the mutation on a single element
   parameter (Height  or Width [sector increment]) */

 double *param_ptr;
 int range_index, generation;

 {
 int sign;

 sign = (int) floor(drand48()*2);

 if (sign == 0)  /* increasing mutation */
  {
  if((MAXRANGE[range_index] - *param_ptr) > 0.0)
   *param_ptr += mutation(generation, MAXRANGE[range_index] - *param_ptr,
                          SCALE[range_index]);
  else
   *param_ptr -= mutation(generation, *param_ptr - MINRANGE[range_index],
                          SCALE[range_index]);
  }

 else  /* decreasing mutation */
  {
  if ((*param_ptr - MINRANGE[range_index]) > 0.0)
   *param_ptr -= mutation(generation, *param_ptr - MINRANGE[range_index],
                          SCALE[range_index]);
  else
   *param_ptr += mutation(generation, MAXRANGE[range_index] - *param_ptr,
                          SCALE[range_index]);
  }
} /* end single_mutate */
```

```
/********************** MUTATION *****************************/

double mutation(gen, range, scale)
 /* mutation function, called by both single_mutate and element_mutate,
    gives non-uniform mutation using the formula in Michalewicz 1992.
    BETA parameter is set in the header file, as is GEN_NUM.
    Function should return decreasing mutations as the number of
    generations increases.*/

    /* NOTE: 'scale' rounds 'change' to the appropriate scale of
                response - discretizes search basically */


 int gen;
 double range, scale;

 {
 double change;

 change = range*( 1 - pow(drand48(),pow(1 - (gen/GEN_NUM), BETA)));
 change = rint(change*(1/scale));
 change = change*scale;

     /* make sure nonzero mutation */
 if (change == 0.0) change = scale;

 return (change);
 } /* mutation */



/************************** CROSSOVER_OFFSPRING ********************/

void crossover_offspring(crossover_ptr, op)
 /* This function selects two parents and randomly crosses
    their 'elements' (genes). */

 struct param *crossover_ptr;
 FILE *op;

 {
 struct param *random_parents, *ordered_parents;
```

```
int cross_total;
int odd;

cross_total = 0;
random_parents = crossover_ptr;
ordered_parents = crossover_ptr;

while (ordered_parents != NULL)  /* get total count of parents */
 {
  cross_total++;
  ordered_parents = ordered_parents->next;
 }

ordered_parents = crossover_ptr;

                /* rearrange list to randomize parents */
permute_parents(ordered_parents, random_parents, cross_total);

odd=(int)fmod((double)cross_total,2.0);
if ( odd == 0)  /* even # parents */
 {
  while (random_parents != NULL)
   {
    cross(random_parents, random_parents->next, op, 1);
    random_parents = random_parents->next->next;
   }
 }
else /* odd # parents */
 {
  while (random_parents->next != NULL)
   {
    cross(random_parents, random_parents->next, op, 1);
    random_parents = random_parents->next->next;
   }
  cross(random_parents, crossover_ptr, op, 0);  /* cross last with first */
 }
} /* end crossover_offspring */
```

```
/************************ CROSS ********************************/

void cross(parent1_ptr, parent2_ptr, op, flag)
 /* crosses the two parent paremeterizations & writes the
    offspring to the file 'file' */

 struct param *parent1_ptr, *parent2_ptr;
 FILE *op;
 int flag;

 {
 struct param *temp;
 int cut;
 double example[3];

 temp = (struct param *) malloc(sizeof(struct param));
 cut = (int) floor(drand48()*5);

 switch (cut)
  {
  case 1: /* cut after K */
    bcopy((char *) parent2_ptr->D, (char *)temp->D, sizeof(example));
    bcopy((char *) parent2_ptr->E, (char *)temp->E, sizeof(example));
    bcopy((char *) parent2_ptr->Dead, (char *)temp->Dead, sizeof(example));
    temp->H = parent2_ptr->H;
    temp->W = parent2_ptr->W;

    bcopy((char *) parent1_ptr->D, (char *)parent2_ptr->D, sizeof(example));
    bcopy((char *) parent1_ptr->E, (char *)parent2_ptr->E, sizeof(example));
    bcopy((char *)parent1_ptr->Dead,(char *)parent2_ptr->Dead, sizeof(example));
    parent2_ptr->H = parent1_ptr->H;
    parent2_ptr->W = parent1_ptr->W;

    bcopy((char *) temp->D, (char *)parent1_ptr->D, sizeof(example));
    bcopy((char *) temp->E, (char *)parent1_ptr->E, sizeof(example));
    bcopy((char *) temp->Dead, (char *)parent1_ptr->Dead, sizeof(example));
    parent1_ptr->H = temp->H;
    parent1_ptr->W = temp->W;
    break;

  case 2: /*cut after D */
    bcopy((char *) parent2_ptr->E, (char *)temp->E, sizeof(example));
```

```
  bcopy((char *) parent2_ptr->Dead, (char *)temp->Dead, sizeof(example));
  temp->H = parent2_ptr->H;
  temp->W = parent2_ptr->W;

  bcopy((char *) parent1_ptr->E, (char *)parent2_ptr->E, sizeof(example));
  bcopy((char *) parent1_ptr->Dead, (char *)parent2_ptr->Dead, sizeof(example));
  parent2_ptr->H = parent1_ptr->H;
  parent2_ptr->W = parent1_ptr->W;

  bcopy((char *) temp->E, (char *)parent1_ptr->E, sizeof(example));
  bcopy((char *) temp->Dead, (char *)parent1_ptr->Dead, sizeof(example));
  parent1_ptr->H = temp->H;
  parent1_ptr->W = temp->W;
  break;

 case 3: /* cut after E */
  bcopy((char *) parent2_ptr->Dead, (char *)temp->Dead, sizeof(example));
  temp->H = parent2_ptr->H;
  temp->W = parent2_ptr->W;

  bcopy((char *) parent1_ptr->Dead, (char *)parent2_ptr->Dead, sizeof(example));
  parent2_ptr->H = parent1_ptr->H;
  parent2_ptr->W = parent1_ptr->W;

  bcopy((char *) temp->Dead, (char *)parent1_ptr->Dead, sizeof(example));
  parent1_ptr->H = temp->H;
  parent1_ptr->W = temp->W;
  break;

 case 4: /* cut after Dead */
  temp->H = parent2_ptr->H;
  temp->W = parent2_ptr->W;

  parent2_ptr->H = parent1_ptr->H;
  parent2_ptr->W = parent1_ptr->W;

  parent1_ptr->H = temp->H;
  parent1_ptr->W = temp->W;
  break;

 default: /* cut after H */
  temp->W = parent2_ptr->W;
```

```
    parent2_ptr->W = parent1_ptr->W;

    parent1_ptr->W = temp->W;
    break;
  } /* end switch */


  /* append offspring to file */

fprintf(op," %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f\n",
                parent1_ptr->K[0], parent1_ptr->K[1], parent1_ptr->K[2],
                parent1_ptr->D[0], parent1_ptr->D[1], parent1_ptr->D[2],
                parent1_ptr->E[0], parent1_ptr->E[1], parent1_ptr->E[2]);
 fprintf(op,"   %4.1f   %4.1f   %4.1f  %6.5f %6.5f\n\n",
               parent1_ptr->Dead[0], parent1_ptr->Dead[1], parent1_ptr->Dead[2],
               parent1_ptr->H, parent1_ptr->W);

 if (flag) { /*write both offspring */
fprintf(op," %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f %2.1f\n",
                parent2_ptr->K[0], parent2_ptr->K[1], parent2_ptr->K[2],
                parent2_ptr->D[0], parent2_ptr->D[1], parent2_ptr->D[2],
                parent2_ptr->E[0], parent2_ptr->E[1], parent2_ptr->E[2]);
 fprintf(op,"   %4.1f   %4.1f   %4.1f  %6.5f %6.5f\n\n",
               parent2_ptr->Dead[0], parent2_ptr->Dead[1], parent2_ptr->Dead[2],
               parent2_ptr->H, parent2_ptr->W);
 } /* end writing both offspring */

free(temp);
} /* end cross */


/********************** PERMUTE_PARENTS ***************************/

void permute_parents(order_ptr, random_ptr, cross_total)

 /* Randomizes the order of the parents in the crossover_list */

 struct param *order_ptr, *random_ptr;
 int cross_total;
```

```
{
struct param *pre_ord, *temp_ord, *ord_begin,  *temp_random;
int index, chosen, left;

ord_begin = order_ptr->next; /* 1st elem of order list is first of rand */
temp_ord = ord_begin;
left = cross_total - 1;    /* account for first element being removed */
temp_random = random_ptr;
temp_random->next = NULL;

while (left != 1)
 {
 chosen = (int) floor(drand48()*left) + 1;
                              /* randomly select next element*/
 if (chosen ==1)  /* first element selected */
  {
              temp_random->next = ord_begin;   /* add ord beginning to rand. list*/
              ord_begin = ord_begin->next;     /* reset ord beginning */
              temp_random = temp_random->next; /* move temp_random ptr */
              temp_random->next = NULL;        /* resent end of temp_random list*/
              left--;                          /* decrease count of ordereds */
  }
 else
  {
              index = 1;
              temp_ord = ord_begin;         /* reset to current beginning of list*/
              while (index < chosen)
               {
                pre_ord = temp_ord;
                temp_ord = temp_ord->next;
                index++;
    } /* stop at chosen element */
   temp_random->next = temp_ord;   /* add to random list */
              pre_ord->next = temp_ord->next; /* remove from ord list */
              temp_random= temp_random->next;
              temp_random->next = NULL;       /* resent end of random list */
              left--;
 } /* end non-first else */
} /* end of left != 1 while */

/* left = 1 */
temp_random->next = ord_begin;   /* add ord beginning to rand. list*/
```

```
    ord_begin = ord_begin->next;     /* reset ord beginning */
    temp_random = temp_random->next; /* move temp_random ptr */
    temp_random->next = NULL;        /* resent end of temp_random list*/
   } /* end permute_parents */




/********************** FREE_LIST **********************/

void free_list(start_ptr)
/* runs down a param list and frees all the memory */

struct param *start_ptr;  /*beginning address of 'deletion' list*/
{
 struct param *temp;

 if (start_ptr!=NULL)   /* something to delete */
  {
  while (start_ptr->next!=NULL) /* more than first element to delete */
  {
   temp=start_ptr;     /* start at beginning of 'deletion' list */
   while (temp->next->next!=NULL) temp=temp->next;   /* find ptr to last element */
   free(temp->next);                 /* free last element */
   temp->next = NULL;      /* make new end */
  } /* while: repeat process */

  free(start_ptr);       /* delete first element */
 } /* end if*/
} /* end free_rest_list */
```

JOEL HOWARD REYNOLDS

Education

University of Washington, Seattle, 1989 - 1996

Quantitative Ecology and Resource Management. Ph. D.

University of California, Los Angeles, 1/1989 - 9/1989

Biomathematics. M.S.

University of California, Los Angeles, 1984 - 1988

Mathematics. B.A.

Publications

Reynolds, J. H. and E. D. Ford. (submitted). Multi-Criteria Assessment of Ecological Process Models.

North, M. and J. H. Reynolds. (1996). Microhabitat analysis using radio telemetry locations and polytomous logistic regression. Jrnl of Wildlife Management 60(3): 639-653.