

MODELING AND COMPUTATIONAL ADVANCES
REGARDING THE PREDICTIONS OF TREE GROWTH
RESPONSES TO ENVIRONMENTAL STRESS

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

YONGTAO GUAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

JULY 2002

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a master's thesis by

Yongtao Guan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

HARLAN W. STECH KATHRYN E. LENZ

Name of Faculty Advisors

Signature of Faculty Advisors

Date

GRADUATE SCHOOL

ABSTRACT

In this thesis, three updated versions of the tree growth simulation program ECOPHYS have been developed. The three versions are designed to take special advantage of various computing platforms. A new shading algorithm Dynamical Canvas-Building Algorithm (DCBA) has been developed and implemented into the ECOPHYS simulation. This algorithm dramatically improves the speed and accuracy of the simulation. From a comparison between DCBA and a previously implemented shading algorithm, a bug in the older algorithm has been observed. Based on DCBA, Bit-DCBA, which uses one bit to denote each canvas pixel, has been developed. A simulation of Bit-DCBA shows that it is not only memory-saving but also faster than DCBA. A class of simple differential equation models of branch internode growth has been developed and studied. A carbon allocation strategy called “constant allocation” has been developed and implemented into the ECOPHYS simulation. It solves the problem of inconsistency of internode growth that existed with the previous strategy.

Keywords: ECOPHYS, Shading Algorithm, Carbon Allocation, DCBA

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Harlan Stech, my advisor, for his guidance and support throughout this research. His efforts contributed greatly to my understanding and enthusiasm for this research.

I would like to thank my co-advisor, Dr. Kathryn Lenz, for her guidance and support throughout this research. Dr. Lenz brought me into the carbon allocation issue and gave me many valuable suggestions.

I would like to thank Dr. George Host, who served on my examining committee. Dr. Host supported me not only academically, but also financially with his research funds.

I would like to thank Dr. Guihua Fei, who helped me on the idea and development of the DCBA algorithm.

I would like to thank my fellow students in our research group: Larry Tordson, Kyle Roskoski and Mariah Olson. They helped me a lot on the coding and with the field data associated with the ECOPHYS simulation. Also they made my research experience full of fun!

This work was funded jointly by the Computational Biology Program of the National Science Foundation, Grant No. DBI-972395, the Northern Global Change Program of the USDA Forest Service, and the US Department of Energy under Interagency Agreement # DE-A105-800R20763.

CONTENTS

Chapter 1. ECOPHYS Overview.....	1
1.1 Sunlight Interception.....	3
1.2 Photosynthesis.....	4
1.3 Transportation	5
1.4 Growth	8
Chapter 2. Implementations of ECOPHYS.....	10
2.1 General Code Structure.....	10
2.2 COM, COM-less and Linux.....	13
2.3 Sub-model Implementation.....	18
Chapter 3. A New Shading Algorithm	19
3.1 A Brief Introduction to the Shading Algorithm.....	19
3.2 Dynamical Canvas-Building Algorithm (DCBA)	20
3.3 A Bit-based Dynamical Canvas-Building Algorithm (Bit-DCBA)	28
Chapter 4. Carbon Allocation.....	32
4.1 The Transportation Matrices Model	32
4.2 ODE Models of Internode Carbon Intake and Respiration.....	36
4.3 Constant Carbon Allocation and Its Implementation.....	46
Chapter 5. Simulation and Comparison	50
5.1 ScanDensity vs. Model Speed and Model Predictions	50
5.2 Old Shading Algorithm vs. DCBA	55
5.3 DCBA vs. Bit-DCBA.....	59
5.4 Quick Sort vs. Shading Speed.....	61
Chapter 6. Conclusions and Further Study.....	67
REFERENCE.....	70
Appendix.....	73
1. Makefile for Linux Version	73
2. Downward Transportation in the Old Code.....	74
3. Code for DCBA With Quick Sort.....	75
4. Code for Bit-DCBA With Quick Sort.....	78
5. Constant allocation code	81

Chapter 1. ECOPHYS Overview

ECOPHYS is an explanatory physiological whole tree process model designed to simulate the growth of a juvenile poplar tree during its establishment years (the first 7 years). ECOPHYS version of the model was developed as part of the North Central Forest Experiment Station's research program on Intensively Cultured Plantations for Fiber and Energy Production, funded cooperatively through the Department of Energy [1, 2]. Process models, which embody hypotheses about growth mechanisms and which are parameterized on the basis of experimental data, can be defined in such a way as to construct formal and precise statements about the functioning of particular systems and their responses to stimuli. Whole-tree process models thus provide a framework for analyzing tree growth and offer greater explanatory power than dimensional, best-fit, descriptive models, which are lacking in theoretical basis [2].

ECOPHYS is based on experimental field studies of the morphology, physiology, and growth of *Populus* at the USDA Forest Service Forestry Science Laboratory in Rhinelander, Wisconsin, USA. Originally, the model was designed to simulate the growth of poplar cuttings grown in short-rotation intensive culture plantations, where moisture and nutrients are maintained at optimal levels [3]. Currently, the model is being used to understand the effects of multiple environmental stress effects on poplar and aspen growth [20].

Generally speaking, ECOPHYS has four modules: shading, photosynthesis, carbon allocation / transportation and whole tree growth. From the canopy structure, we know the position and orientation of each leaf, and for each hour of each particular day, we know the direction of the solar beam. The shading module uses the above information to calculate the shaded fraction of each leaf. Based on the shading information of each leaf and current environmental conditions, i.e. PPFD (Photosynthesis Photon Flux Density), Temperature, Relative Humidity, CO₂ and O₃, the photosynthesis module calculates the amount of photosynthate that each leaf produces. The above two steps are done each hour. We assume the canopy structure does not change during the sunlit hours of the day. After sunset (PPFD < 25 $\mu\text{mol m}^{-2} \text{s}^{-1}$) of each day, the transportation module allocates the photosynthate to different growth centers within the tree. The growth module grows each part of the tree respectively according to its genetically – determined growth pattern. The general procedure is illustrated in the figure below:

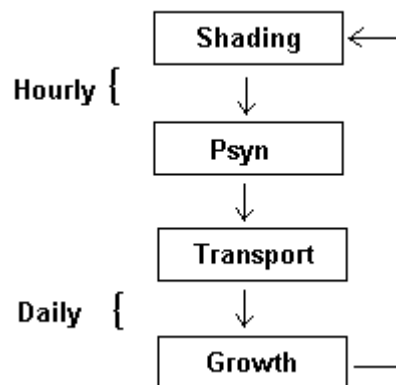


Figure 1.1 General procedure of the ECOPHYS simulation

ECOPHYS consists of a number of sub-models. A more detailed description of each follows.

1.1 Sunlight Interception

There are two sub-models in the sunlight interception module:

- (1) The Shading Algorithm, which calculates the shaded / sunlit fraction of each individual leaf.
- (2) The Estimation of Diffused Sunlight (EDS).

A new shading algorithm has been developed and will be discussed thoroughly in Chapter 3. The EDS algorithm is based on Leaf Area Index (LAI) and Beer's Law (Exponential Decay) [4].

ECOPHYS models the leaf as being diamond-shaped for simplification purposes, as shown in Figure 1.2 below.

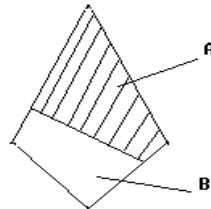


Figure 1.2 Leaf with shading

For example, in the leaf above, A denotes the shaded area and B denotes the sunlit area. Therefore, B receives both diffused / ambient and direct sunlight, while A receives only diffuse sunlight. The ECOPHYS shading algorithm can determine the fractions for A and B. Suppose the fraction for A is a . Then the fraction for B is $(1-a)$. Suppose the leaf area is S , a known parameter. We have that $A = S*a$ and $B =$

$S^*(1-a)$. For B, direct sunlight PPFD is read directly from a weather file, taking leaf orientation into account. For both A and B, diffuse sunlight PPFD is calculated by the EDS algorithm.

1.2 Photosynthesis

The purpose of this sub-model is to calculate photosynthate production for each individual leaf. ECOPHYS divides the leaves on a tree into four maturity classes: expanding leaves (MC1), recently mature leaves (MC2), mature leaves in upper crown (MC3), and mature leaves in lower crown (MC4). Each maturity class has its own set of clone-specific parameters.

A well-developed biochemical mechanistic model, the RUBISCO model, has been implemented in ECOPHYS.

This model takes various parameters as inputs. The inputs can be classified into two categories: clonal parameters and environmental conditions. Clonal parameters include F_{max} (Maximum CO₂ exchange rate), Photosynthetic Efficiency (Efficiency of production at low light intensities) and R_d (Dark respiration). Environmental conditions include PPFD (light), Temperature, Relative Humidity, CO₂ and O₃. Clonal parameters are preset and depend on the particular poplar clone to be simulated. Environmental conditions are read from data files generated at a particular location, currently the FACE (Free Air CO₂ Enrichment) site located at Rhinelander, Wisconsin.

1.3 Transportation

The Carbon Allocation / Transportation module is the model used to determine the proportion of photosynthates exported from a leaf and the pattern of photosynthate allocation from the leaf to various growth centers within the plant, such as the plant apex, leaves, stem and roots. The transportation model is an extremely important part of the ECOPHYS model because it controls the allocation of photosynthate, the basic currency of growth in plants. But, because of lack of knowledge of the specific mechanisms governing the dynamic processes of carbon allocation, and uncertainties about the dynamics of transient structures, this explanatory model is less reliable [6]. However, an empirical model, based on transportation matrices, has been developed here and implemented in ECOPHYS. New transportation models will be discussed in Chapter 4.

LPI (Leaf Plastochron Index) plays a very important role in modeling carbon allocation. Within a branch, LPI is the order of the leaves on the branch counting from top to bottom and starting with LPI zero. LPI is correlated to a leaf's age. This is the reason why plant physiologists adopt the concept of LPI instead of leaf age. They can observe and record LPI conveniently, while it is difficult to record leaf age. Before the budset day, the top LPI is zero. After the budset day in the ECOPHYS simulation code, the LPI of each leaf increases as if there were imaginary new leaves emerging. In this way, LPI in ECOPHYS is directly tied to leaf age throughout the growing season.

Based on the data of field experiments using radiotracer C14, carbon transportation matrices were determined [1, 7]. The percentage of assimilates available for translocation from a given leaf is a function of LPI, as shown below.

Source LPI	0-4	5	6	7	8	9	10-19	>19
%Exported	0	15	30	45	60	75	90	100

Table 1.1 Export percentage of a leaf with particular LPI [1,7]

The listed percentage is computed after the deduction of the maintenance respirations for the source leaf.

Each leaf has its own carbon allocation pattern, also based on LPI. Photosynthate may be transported upward to developing leaves or stem internodes, or downward to the lower stem, hardwood cutting, and roots. Correspondingly, we have both an upward transport matrix and a downward transport matrix.

Source LPI	0-4	5	6	7	8	9	10	11	12	13	14	15	...
Leaf Sink	0	.73	.67	.49	.42	.31	.22	.17	.12	.13	.1	.1	...
Internode Sink	0	.25	.27	.23	.24	.17	.17	.22	.16	.1	.1	.05	...

Table 1.2 Upward Transportation Matrix [1,7]

Source LPI	0-4	5	6	7	8	9	10	11	12	13	14	15	...
Internodes Sink	0	.02	.03	.09	.11	.26	.33	.30	.30	.30	.30	.30	...
Cutting Sink	0	0	.01	.02	.03	.04	.05	.07	.08	.09	.10	.11	...
Roots Sink	0	0	.02	.17	.20	.22	.23	.24	.34	.38	.40	.44	...

Table 1.3 Downward Transportation Matrix [1,7]

Notice that for each LPI, the sum of upward transport coefficients and downward transport coefficients is 1.

The above three matrices tell us how much photosynthate a leaf with specific LPI will export. The next question is how much each specific leaf and internode will receive. Two more matrices that specify the distribution of photosynthate were determined from C14 experiments. The amount of photosynthate received by leaves and internodes can be calculated from these two matrices [1, 7].

1.4 Growth

After assimilates are allocated within the plant, maintenance respiration is subtracted from the photosynthates pool at the sinks; i.e., sink leaves, internodes, cutting, large roots, and fine roots. During early establishment, poplar does not store significant amounts of assimilates in long-term reserve pools. It is, therefore, assumed that all assimilates remaining after maintenance respiration has been subtracted are available for growth [2].

Both leaves and internodes have two phases of growth. For the leaves and internodes that are within the expanding zone (usually $LPI < 10$), leaves grow in area and internodes grow in length only. For those that are out of the expanding zone, leaves grow in thickness / mass and internodes grow in diameter only.

New leaves emerging and old leaves dropping are also classified within the Growth module. After bud-break day, new leaves come out at a certain rate that is pre-determined by genetic properties. For example, for Aspen 259 this rate is 37 hours, which means every 37 hours a new leaf will come out. The new leaf's orientation is also pre-determined. The new leaf orientation is determined inductively, based on the previous leaf orientation. All these pre-determined properties are encoded independent of clonal type.

Leaf dropping ("senescence") depends on a ten-day average of the photosynthate each individual leaf produces. It also depends on two clonal parameters – a leaf drop

threshold and a leaf drop factor, which are also pre-determined genetically and may vary among clones.

Now we have a general idea of the main ECOPHYS modules. In the coming chapters we will discuss both new algorithms and development of the code. Specifically, in Chapter 2 we will discuss the creation of codes for three common computing environments and the sub-model implementation, especially to synchronize the three different versions of codes. In Chapter 3, a new shading algorithm will be discussed. In Chapter 4, we will discuss carbon allocation strategies based on first order differential equation models. Chapter 5 contains the results, especially the speed comparison between different shading algorithms. The final chapter contains conclusions and topics for future study.

Chapter 2. Implementations of ECOPHYS

In this chapter, we will discuss three versions of codes that have been developed as part of this thesis for the ECOPHYS simulation: a COM version, a COM-less version and a Linux version. Each version has its special strengths. We will discuss the general code structure first, then discuss in detail each specific version. Comparisons between different versions of codes are also explained. At the end of this chapter, we will demonstrate sub-model implementations.

2.1 General Code Structure

The ECOPHYS code is a combination of process programming and object orient programming. The simulation subroutine (Simulate.cpp) is the general process routine for the general procedure described in Figure 1.1. Within each step there are corresponding Classes / Objects to implement different functionalities.

The classes in ECOPHYS can be classified into three categories: Descriptive, Functional and Supportive. The descriptive classes describe the morphology of the tree and therefore encapsulate the genetic properties / functions within description classes (for example, the pattern of growth). Functional classes encapsulate the implementation of different algorithms. For example, class WIMOVAC implements the photosynthesis algorithms, class Shading implements the shading algorithm. Supportive classes are classes that support the functionality of Descriptive and Functional classes, such as class Matrix, class Line and class xPoints implementing

basic geometry and arithmetic calculations, and class CLONE_PARAMETERS and class PARAMETERS describing the clonal parameters and control properties.

ECOPHYS has five descriptive classes: Leaf, Internode, Branch, Root and Tree. For the above ground part, a two-way link-list of Branch and Internode objects constitutes the skeleton of the tree. The Branch object consists of Internode objects, and a branch itself attaches to an internode. The Tree object is the head of this link-list. Leaf objects attach to internodes. The Branch class is the most important class in our simulation because the basic functionalities are based on the branch class and the recursive calls are also implemented at the branch level. In addition, photosynthate transportation is mainly implemented at the branch level. The whole structure is illustrated as below.

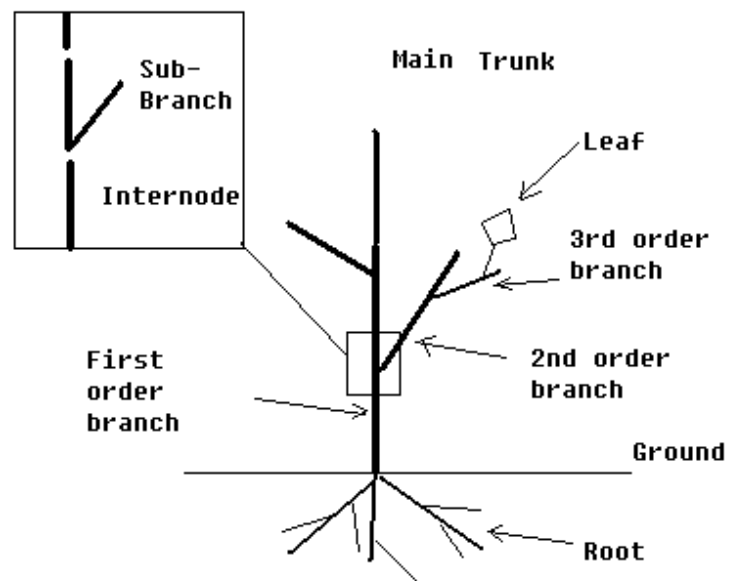


Figure 2.1 Structure of an ECOPHYS tree

When we traverse the leaves on the tree, we go from the TREE object to BRANCHes, to INTERNODEs then to individual LEAF. Due to the recursive property of the tree structure, we can use recursive calls to make implementation of canopy traversal easy. There are six main Functional classes defined in ECOPHYS. WIMOVAC encapsulates the photosynthesis algorithms. Shading / MyShade implements the shading algorithm. TRANSPORT does the carbon allocation at both the branch level and the whole tree level. Class glView implements the OpenGL for tree growth visualization. WEATHDAY currently reads in weather data from a file and may in the future be a weather simulator that generates weather conditions automatically. Finally, JULIAN is used to find solar direction.

Among the Supportive classes, class CLONE_PARAMETERS and class PARAMETERS are the ones that people should pay most attention to. CLONE_PARAMETERS encapsulates all clonal parameters. If, for example, we want to simulate the clone ASPEN 259 instead of Eugenei, we simply change the clonal parameters defined in this class and recompile the code. The class PARAMETERS defines all the control parameters for the simulation, such as the begin and end dates of the simulation, the dates to drop output data files, etc.

2.2 COM, COM-less and Linux

1. COM version

The COM (Component Object Model) version was developed by a group of people (Gang Wu, Guoqiang Zang, Wanlun Zhao in 1998 – 2000). The main strengths of the COM version are:

- (1) The use of parallel computation based on DCOM (Distributed COM).
- (2) Preparing an interface for weather simulation.

Simply put, COM is a way of building objects that is independent of any programming language. In other words, COM transcends language-specific ways of building reusable objects and gives us a true binary standard for object architectures. C++ classes have member functions; COM objects have methods. Methods are grouped into interfaces and are called through interface pointers.

An executable that implements a COM object is called a COM server. COM servers come in two basic varieties: in-process and out-of-process. In-process servers (often referred to as *in-proc* servers) are DLLs (Dynamic Link Library). They're called in-procs because in the Win32 environment, a DLL loads and runs in the same address space as its client. EXEs, in contrast, run in separate address spaces that are physically isolated from one another. In most cases, calls to in-proc objects are very fast because they're little more than calls to other addresses in memory. Calling a method on an in-proc object is much like calling a subroutine in your own application.

Out-of-process servers (also known as *out-of-proc* servers) come in EXEs. One advantage to packaging COM objects in EXEs is that clients and objects running in two different processes are protected from one another if one crashes. A disadvantage is speed. Calls to objects in other processes are roughly 1,000 times slower than calls to in-proc objects because of the overhead incurred when a method call crosses process boundaries.

Microsoft Windows NT 4.0 (and Windows 2000) introduced Distributed COM (DCOM), which gives out-of-proc servers the freedom to run on remote network servers. It's simple to take an out-of-proc server that has been written, tested, and debugged locally and deploy it on a network. For detailed information about how to implement DCOM to ECOPHYS, please refer to Gang Wu's thesis [14] and the following books [8,9,10,11,12,13].

In the original COM implementation (1998), almost all the ECOPHYS algorithms were packaged into separate COM objects, both of type DLL and EXE. This made debugging difficult, and slowed the implementation of new algorithms. In the new COM version created as part of this thesis, only the shading algorithm and weather simulation module are packaged as COM objects. The shading algorithm was built as an EXE and weather module was packaged as a DLL. This particular approach has the advantage of allowing a cluster of networked computers to collaborate on shade tasks, and allowing module replacement of the weather server.

2. COM-less version

The COM-less version was developed for two considerations. First, it is more convenient for programmers who are familiar with C++ but not COM to start working on the ECOPHYS project. The new participants do not need to study COM, which requires a lot of time for personal study, to grasp the meat of the whole ECOPHYS structure in order to implement new algorithms. Secondly, this is a middle step in preparing for the implementation of ECOPHYS to the Linux operating system. Finally, it is a version that (unlike the COM version) is unaffected by operating system upgrades.

Converting from the COM version to the COM-less version is relatively easy, compared to programming in COM, if a developer knows how COM works. Remember that COM's methods are the analogues of class member functions in C++. The following is the procedure for how to convert a COM server to a COM-less implementation.

- (1) Define a new class in C++.
- (2) Add new member functions corresponding to the COM methods.
- (3) Copy the code of methods into the associated member functions.
- (4) Add the member variables of the COM objects to C++ class.

Special attention must be paid when converting global variables of each COM server into C++ class variables. Redefinition error may occur when doing the converting. One must examine the lifetime of global variables in COM, making the corresponding class variables compatible to the whole C++ project.

The following figure shows the relationships between the ECOPHYS modules and the classes / objects.

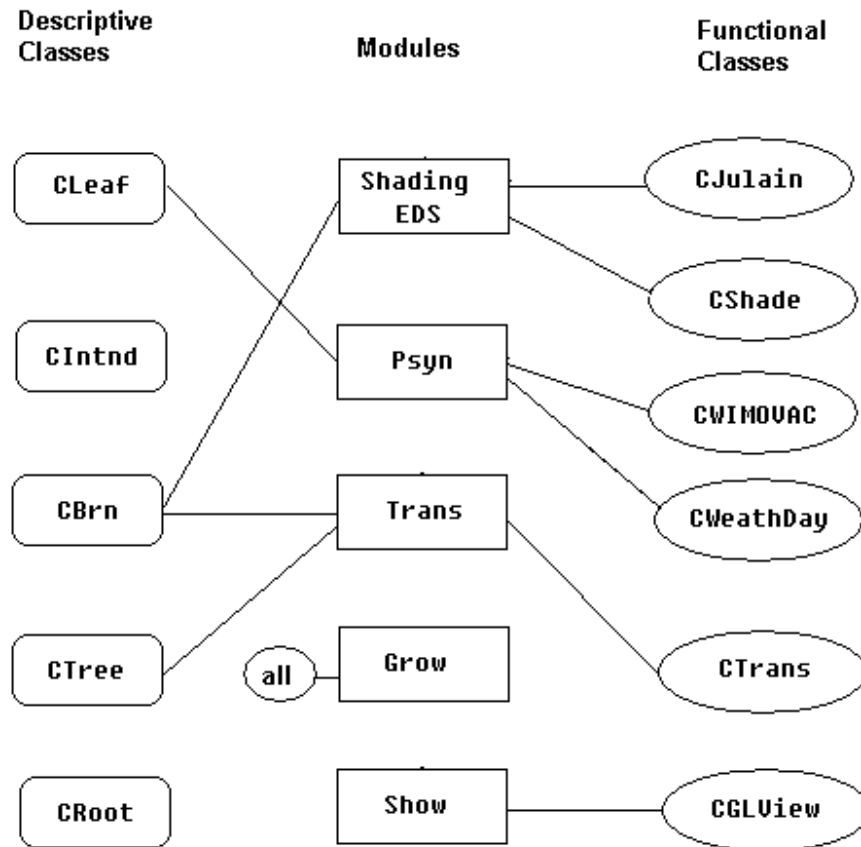


Figure 2.2 Relationships between ECOPHYS modules and classes.

From the above figure, it is very clear where the differences between the COM and the COM-less versions are. In the COM version, CShade and CWeathDay define two COM objects, while in the COM-less version they are two classes.

3. Linux version

A Linux version was developed as part of this thesis because we want to take advantage of Beowulf Clusters in the parallel computation and generally want to avoid the overhead costs of the windows operating systems.

The most difficult part about the Linux version is the OpenGL implementation with X-windows. Luckily, there are more than enough example codes and documentations on Internet. For example, <http://www.opengl.org> and <http://nehe.gamedev.net> have very good resources. The current code that was developed as part of this thesis using OpenGL under Linux is sufficient for ECOPHYS visualization.

Under Linux, another difficulty is how to write a makefile for ECOPHYS. Again, we can also take advantage of Internet resources. The GNU make manual should be considered as the best resource. The current makefile is listed in Appendix 1. This is a user-friendly style. If we want to add more files into ECOPHYS, we simply add the file name to the corresponding macro definitions.

The scheme of the code structure for the Linux version is the same as the COM-less version. Some slight differences exist because some convenient functions in Visual C++ do not apply under Linux. For example, there is no `__min()`, `__max()` function under Linux, so we must write these by ourselves.

2.3 Sub-model Implementation

From descriptions above, especially figure 2.2, we already have a general idea of the structure of the three versions of the ECOPHYS code. The differences between new COM version and COM-less version are that in the new COM version, the Shading and WeathDay modules are built with COM. The differences between the COM-less version and the Linux version are in the GIView module and some slight changes of functions like `__min()`, `__max()`. The majority of the code stays the same. This makes it easy to implement the sub-models of modules other than Shading and GIView.

We take an example from the transportation module to demonstrate how easy it is to implement new algorithms and synchronize between different versions. For example, starting with the COM-less version. If we want to use a strategy that transports carbon downward to internodes proportionate to internode lateral surface area instead of internode length (please refer to Chapter 4), we need to modify a function in class TRANSPORT called `SetDownTC()`, which is located in the file of `Transport.cpp` [see also Appendix 2]. After we finish the change, we compile and debug and run it first. If everything goes smoothly, we make exactly the same changes on the Linux code and the new COM code.

Chapter 3. A New Shading Algorithm

In this chapter, we will discuss a new algorithm based on dynamic canvas building for computing the shaded fraction of each leaf. It was designed and implemented into the ECOPHYS simulation. We will see that it improved both speed and accuracy. In order to save memory usage, an alternative way to build a canvas, which uses one bit to denote each pixel, is also introduced.

3.1 A Brief Introduction to the Shading Algorithm

In ECOPHYS simulation, the shading algorithm is the most time consuming part. Previously, it “consumes almost 80% of the total run time” [16]. Among these 80%, the canvas building subroutine costs the majority time [16,17].

Efforts have been made to improve the simulation speed of ECOPHYS, including two different approaches. One is parallel computation; a COM-based parallel computing technique has been implemented by Gang Wu [14, 15]. Another approach is to improve the sequential speed of the shading algorithm itself. Chandramouli Balasubramaniam simplified the canvas-building algorithm and observed a “45-50 %” improvement at speed [16].

3.2 Dynamical Canvas-Building Algorithm (DCBA)

In this section, we will discuss a new shading algorithm that is based on dynamical canvas building. The following are the main steps of the new shading algorithm.

- (1) Changing coordinate system.
- (2) Sorting leaves due to the z values of Leaf Center Point (LCP).
- (3) Building the canvas that each leaf will project on.
- (4) Projecting leaves onto canvas one by one using a scan conversion algorithm.
- (5) Counting pixels calculating the sunlit and shade fractions.

The detailed explanations of each step follow.

1. Changing Coordinate Systems.

There are four different coordinate systems in the ECOPHYS simulation, namely, (1) TCS (Tree Coordinate System), (2) LCS1 (Leaf Coordinate System 1), (3) LCS2 (Leaf Coordinate System 2) [18] and (4) SCS (Shading Coordinate System). The main purpose of these coordinate systems is to express each of the four leaf vertex coordinates in terms of the TCS whose origin coincides with the base of the tree stem. In the TCS the positive X-axis points east, the positive Y-axis points north, and the positive Z-axis points upwards (i.e. the right hand system). LCS1's x, y, and z-axes are parallel to those of TCS and its origin coincides with the vertex of the leaf attaching to the petiole. LCS2's origin is the LCP (Leaf Center Point) with the x and y-axes lying within the lamina plane. The Z-axis is perpendicular to the lamina plane and is coincident with the normal to that plane. Obviously, both LCS1 and LCS2 are leaf specific and they work together to express the vertices into TCS conveniently.

Mathematically, this is achieved by multiplying different matrices. The SCS' origin is the same as TCS', but with the z-axis pointing to the sun and the x, y-axes lying within the plane that is perpendicular to the z-axis (sun vector). The following figure illustrates all the coordinate systems [18].

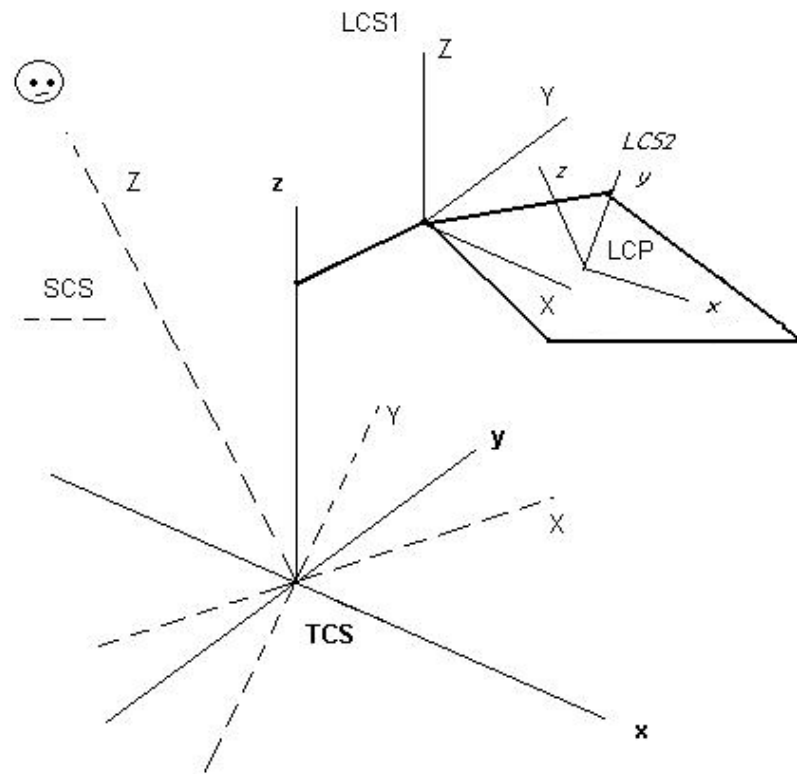


Figure 3.1 Four different coordinate systems in ECOPHYS

Changing coordinate systems requires changing the leaves' vertices and LCP from TCS to SCS. This is done by multiplying two matrices dependant on the angles of the sun vector. Meanwhile, we enlarge the size of each leaf by multiplying a number called ScanDensity, which we will discuss later.

2. Sorting the leaves according to the z-value of their LCPs.

This step is pretty straightforward, involving just sorting the leaves due to their center point z-values. These z-values define the spatial relationships between different leaves. A larger z-value means that it is at the top, from the viewpoint of the sun.

There are a number of well-developed sorting algorithms that we can choose, like quick sort, bubble sort, etc. The sorting algorithm plays a very important role in the performance of speed. In Chapter 5, we will see big differences on shading speeds between the implementations of different sorting algorithms. The sorting algorithms, although they are different in many ways, mainly do two different things: comparing objects and changing orders. For a single-machine version of ECOPHYS, we can pass pointers (the addresses of the objects) instead of the objects themselves (which are big). We get two advantages. One is that the data transfer is very small. The other is that when changing orders, we only need to change the addresses of the objects, which are typically 4 bytes, while if we move objects, it would be significantly more time consuming. However, the advantages of sending pointers does not apply to the parallel computing version of ECOPHYS in which sending objects is a must because the current parallel version distributes different shading tasks to different processors. Each processor does its own sort.

3. Build the canvas

A scale factor called ScanDensity has multiplied the x and y coordinates to enlarge each individual leaf. This will make our computing more accurate. For example, if we have a leaf whose actual size is 1.5 cm x 1.5 cm, it covers only at most four integer points, assuming each centimeter corresponds to a distance of one pixel. If we multiple 10 to every dimension, we get a 15 cm x 15 cm square, which may cover $16 * 16 = 256$ integer points. Obviously, a one-point miss introduces a 25% error to the actual size, while less than a 1% error to the enlarged one. Also, later in our scan-conversion algorithm, we use integer vertices (vertex whose coordinates are all integers) for simplification. With a sufficiently large ScanDensity magnification factor, there should be only a minor loss of accuracy.

This scale procedure not only enlarges individual leaves, but also the canvas as well. In fact, building the canvas is to find the smallest square that contains the projection of every leaf. To build the canvas, the first step is to find the canopy's maximum x and y values and minimum x and y values in the SCS. In the present code, we do this step within the changing coordinate system step. Our canvas size is $(\max y - \min y + 1) * (\max x - \min x + 1)$ and it is a two-dimensional matrix. But in the code, we use a one-dimensional integer array to express the canvas. The second step is to malloc / new an integer array of our canvas size and initialize each pixel as -1.

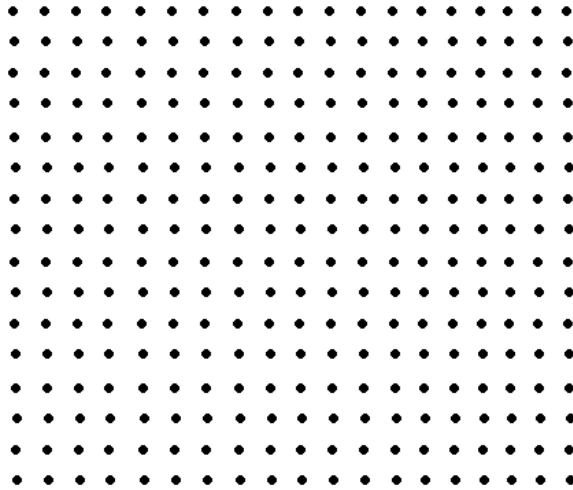


Figure 3.2 Canopy of DCBA algorithm (each pixel is -1)

The above figure shows an example of a canvas. Every pixel has value -1 and there are $(\max y - \min y + 1)$ rows and $(\max x - \min x + 1)$ columns. At the next step, we will project leaves one by one onto this canvas. Changes will be made to the canvas pixels according to whether or not they are shaded by a leaf.

4. Projecting leaves onto the canvas.

We use a scan conversion algorithm to project leaves onto the canvas. Scan conversion algorithms were originally developed to deal with the polygon-filling problem in the computer graphics. Of course, in computer graphics scan conversion fills the pixel with a color, while in ECOPHYS, we fill the pixel with different numbers, the IDs of different leaves. Say, if we have 1000 leaves in ascendant order due to their LCPs' z-values. We use ID values 0~999 to identify each leaf. We project the first lowermost leaf, using a scan conversion algorithm to fill the canvas pixels with 0 within the projected region. Meanwhile we count the total number of pixels within that projected region. This number denotes the projected area of this

leaf. Then, we project the second lowermost leaf, filling shaded pixels with 1, and so on, until the last leaf is projected.

Scan conversion algorithms are basic algorithms in computer graphics. Therefore, they have been thoroughly studied. We can find very good documentation and pseudo-codes from internet. A direct search for keyword “scan conversion” on a search engine like www.google.com will bring us many links. Among them <http://www.cs.berkeley.edu/~laura/cs184/scan/scan.html> and <http://www.gg.caltech.edu/~kurt/tr/derivation.html> are best ones for the ECOPHYS project.

The following figures illustrate an example of two leaves. Each solid dot corresponds to the initial value -1 .

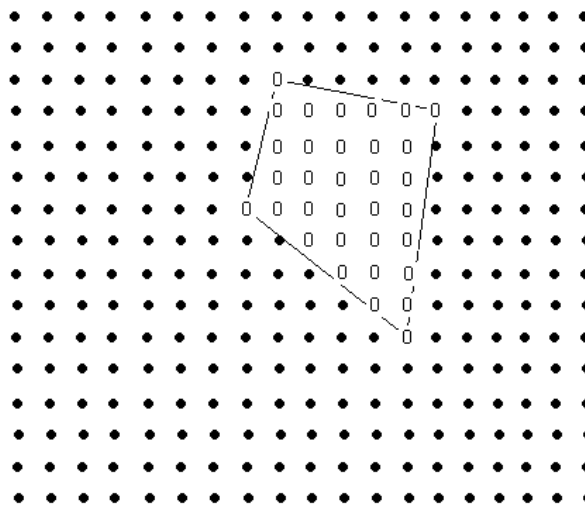


Figure 3.3 Projecting the first lowermost leaf

This is the first lowest leaf projecting on the canvas. We set all the pixels within this leaf projection region as 0, the ID of first leaf. The number of total pixels is counted and found to be 33.

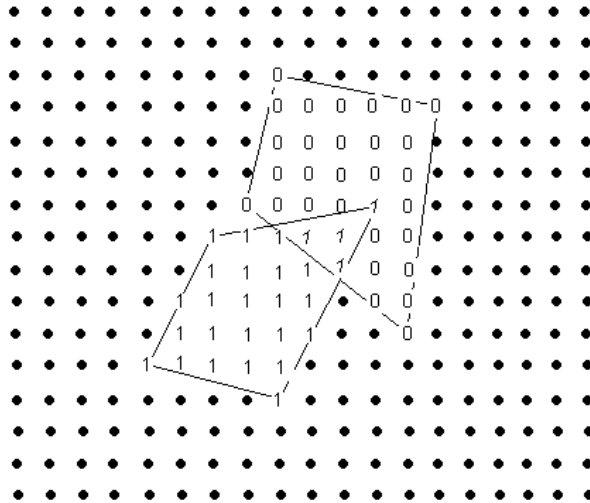


Figure 3.4 Projecting the second lowermost leaf

After the first leaf, we project the second lowest leaf, and set pixels within second leaf projection region to 1, the ID of the second leaf. Notice that some 0's have been changed to 1's as shown in *italic*. This means that the second leaf shades that part of the first leaf. The number of total pixels shaded by leaf 2, i.e. the projected area of leaf 2, is counted and found to be 27.

5. Counting the pixels and calculating the shaded portion of each leaf

After the projection of all leaves, we traverse the whole canvas and count the numbers of each ID. Each number corresponds to the area of the sunlit portion of an individual leaf. After dividing by the total number of pixels that were shaded by each leaf (counted during the projection step), we get the unshaded fraction of each leaf. For our example of two leaves, we have that 29 pixels are 0 (ID of the first leaf) and 27 pixels are 1 (ID of second leaf). So, the sunlit fraction of the first leaf is $29 / 33 = 0.8788$. The sunlit fraction of the second leaf is $27 / 27 = 1$.

The new shading algorithm is very fast compared to the old one. It spends only 25% of the shading time of the old algorithm. (Please refer to Chapter 5, Figure 5.4). The following advantages make the new shading algorithm faster than the previous ones.

(1) We only pass, and later sort, the pointers (addresses) of the leaves instead of the objects of leaves, so the data flow is small comparing to the previous algorithm. When we do the sorting, only the positions of the pointers, instead of the leaf objects, have been exchanged.

(2) Because of the dynamic canvas building, compared to the fix-size canvas building used in old algorithm [16], we don't need to judge if a leaf is projected within the existing canvas or not. This is the most time-consuming aspect of the old canvas-building algorithm. Also, because of the dynamical canvas size, during noon the canvas (pixel matrix) is relatively small. This results in a savings of time.

(3) Because we assume that during the daytime the canopy structure does not change, so we have a fixed leaf number during a day. Therefore, we can keep the sorting result from hour to hour. We do the shading every hour during the day for which $PPFD > 25$. From hour to hour, the relative leaf positions have only changed slightly. Hence when we do the sorting for the leaf pointers, we may only need to change a very small number of them, except for the first sunlit hour of a day.

Finally, because the new shading algorithm is significantly faster than the old algorithm, we can apply our algorithm with a higher ScanDensity factor. This results in higher accuracy.

3.3 A Bit-based Dynamical Canvas-Building Algorithm (Bit-DCBA)

As we described previously, the DCBA algorithm first sorts the leaf pointers, then projects leaves onto the canvas from lowermost to uppermost one by one using a scan-conversion algorithm. The canvas is an integer array and the size of the array depends on the extension of canopy and the scale of ScanDensity that we choose.

A question arises naturally: what if we have a big canopy in relation to available computer physical memory? A simple answer is that we might use a small ScanDensity. That is, we make a compromise between accuracy and memory usage. Can we get keep both advantages: having high ScanDensity and keeping memory usage at lower level? This will turn to be important when we develop the ECOPHYS towards multi-tree simulation. The following bit-wise algorithm is proposed as a possible solution.

Before we proceed, Let us do a simple calculation of memory usage in DCBA. Suppose we have a projected canopy 10 meter by 10 meter with ScanDensity equal to 10. Then we have $10,000 * 10,000 = 100 \text{ M}$ pixels. In DCBA, every pixel is an integer, which usually occupies 4 bytes. So totally we need $100 \text{ M} * 4 = 400 \text{ M}$ memory.

The reason that we use integers to denote pixels is that we use the integer numbers as IDs for different leaves. We use leaf ID because we count different IDs at the last step. However, if we project leaves one by one from topmost to lowermost, we can avoid the usage of leaf ID. If we don't use leaf ID, then we can use one bit, instead of 4 bytes, to denote a pixel. The following figures illustrate this idea for a two-leaf example.

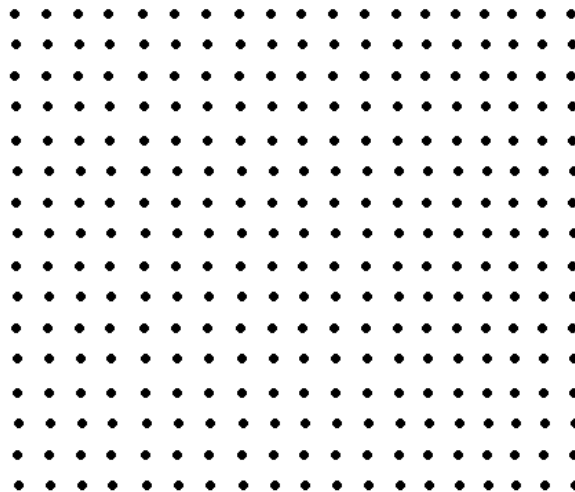


Figure 3.5 Canvas of Bit-DCBA algorithm (each pixel is 0)

The above is a canvas. A bit, 0 or 1, represents each pixel. Initially, we set every pixel to 0. Now, we project the first topmost leaf onto the canvas using a scan conversion algorithm and get the following figure.

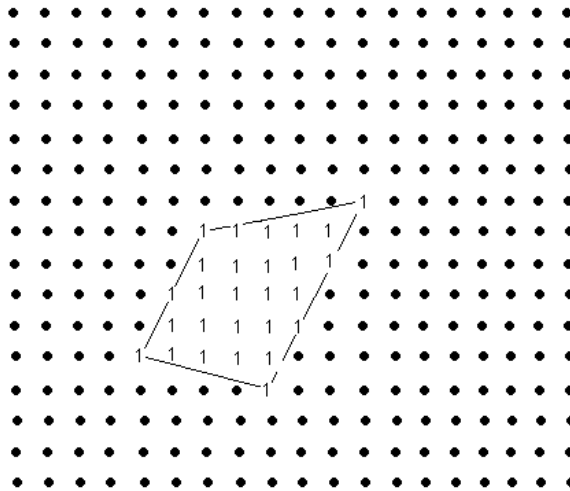


Figure 3.6 Projecting the first topmost leaf in Bit-DCBA

The total number of pixels within this projected region is counted and found to be 27. This number represents the total area of the first topmost leaf. The number of pixels that change from 0 to 1 is also counted and found to be 27. This number represents the sunlit area of the first topmost leaf. We divide the total area, which is 27, by the sunlit area, which is 27, and we get $27/27 = 1$. The sunlit fraction of the first topmost leaf is 1, which is obviously true.

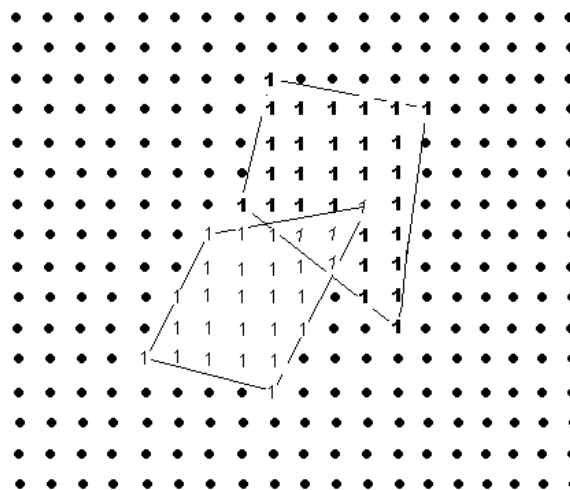


Figure 3.7 Projecting the second topmost leaf in Bit-DCBA

Then we continue to project the second topmost leaf onto the canvas, as shown in the above figure. We also count two numbers. One is the total number of pixels within this projected region, which represents the total area of the second topmost leaf. It reads 33. The other is number of pixels that change from 0 to 1, which represents the sunlit area of the second topmost leaf. It reads 29. Therefore, the sunlit fraction of the second topmost leaf is $29 / 33 = 0.8788$.

Then we project the third topmost leaf and calculate the sunlit fraction. We keep doing this until the last leaf has been projected. Obviously, we don't need the last counting step in DCBA algorithm. We can see that for the two-leaf example, we get the same sunlit fraction in both DCBA and Bit-DCBA.

Now, let's do a simple calculation on how much memory this Bit-DCBA algorithm uses. For a 10-meter by 10-meter canvas and setting ScanDensity as 10, we have $10,000 * 10,000 = 100$ M pixels. Because one byte consists of 8 bits, so we only need $100 \text{ M} / 8 = 12.5$ M memory. It is only $1/32$ memory usage of DCBA.

However, in this algorithm one must count pixel changes for each leaf as well as the total pixels obscured by each leaf. And judgments on whether each pixel has been changed from 0 to 1 or not must be done redundantly on every leaf. But the advantage is that in Bit-DCBA, we do not need to traverse the whole canvas, but only handle all the leaves. This saves time. Moreover, handling a small number of memories saves time too. The overall result is that the Bit-DCBA is faster than DCBA, which we will see in Chapter 5.

Chapter 4. Carbon Allocation

As we discussed in Chapter 1, because of the lack of precise knowledge of the mechanisms that govern the pattern of carbon allocation, we do not have a good process-based model of carbon allocation. However, we do have an empirical model based on transportation matrices [1,5,7], which we will discuss in more detail in this chapter. Also, in this chapter, we will develop a model motivated by first order differential equation models of internode growth. From this a new strategy called “constant allocation” has been developed and implemented into the ECOPHYS simulation code.

4.1 The Transportation Matrices Model

The transportation matrices model is a source-sink model; it allocates carbon at the branch level. Within a branch, leaves with LPI larger than 4 are sources, they transport out certain proportions of carbon, both upward and downward.

For the upward part, there are two kinds of sinks – leaves and internodes. It is called “upward” because that leaves with lower LPI (younger) are at the upper locations on a branch. The amount of carbon that each individual leaf / internode receives can be directly calculated by the upward transportation matrix [1,5,7].

For the downward part, there are three kinds of carbon sinks – internodes, the cutting and roots. Downward transport does not give to leaves. In Wanlun Zhao’s version of code [17], the downward transportation to the internodes is designed to evenly distribute carbon to the internodes below. This is done by modifying the downward transportation matrix [refer to Appendix, part 2, for further information]. We will see that such an allocation in which all downward internodes receive the same amount of photosynthate causes problems. Other allocation strategies will be discussed as the main topic of the following paragraphs.

The cutting is the plant section from which our virtual plant is developed. In our simulation, the carbon to root and cutting have been combined and then redirected to two (downward to mother branches and roots) or three (over winter storage, downward to mother branch and root) different sinks depending on whether it is before or after budset day. The pattern is illustrated in following diagram.

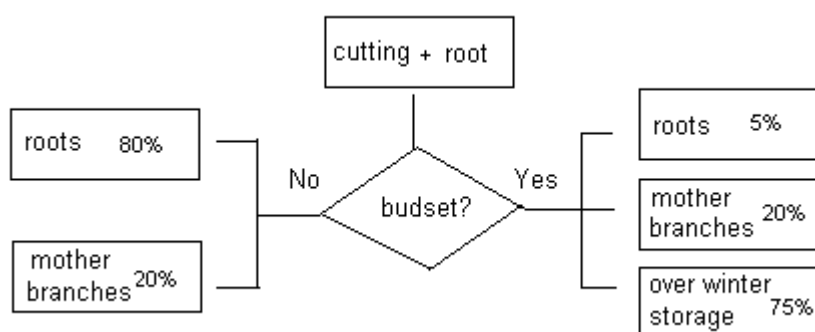


Figure 4.1 Redistribution of the carbon that initially goes to roots and cutting.

The over-winter storage acts as a carbon pool, and it is collected after the bud set day in each growing season. The pool has two kinds of distribution patterns. One is done once a year at bud break day; all available stored carbon is distributed evenly to newborn branches. The other distribution pattern occurs on a daily basis. Everyday a certain portion of the over-winter storage pool is distributed within a branch to the first four leaf-internode units, if they exist. The percentage that is transported out from the over-winter storage pool varies from 20%, if the leaf number on the branch is less than 8, to 50% if leaf number on the branch is larger than or equal to 8. In short, the output of over-winter storage has exponential decay with different rates depending on how many leaves are on the branch.

In Wanlun Zhao's code, the carbon that is transported from source leaves to mother branches was supposed to evenly distribute carbon to all the internodes below the current branch, all the way down to the first internode of the main trunk. But it was found that that part of the code does not function. In my three versions of the code, I rewrote this part of code and made it function to multiple years. [Refer to 4.3 constant allocation and Appendix, part 5]

Although the carbon allocation strategy based on transportation matrices is supported by field experiments, two negative facts motivate us to seek a better solution. One is that in our simulation, consecutive internodes can develop big differences in diameters (we call it "non-monotonicity of internodes"), which indicates a possible problem with carbon allocation and / or carbon respiration costs. The other is that the

matrices used for carbon allocation are constructed from experiments for first year trees. It seems possible that those matrices may not apply to second or later-year trees, especially for internodes.

In the next section, we will develop a class of simple differential equation models, from which we will see idealized and simplified dynamical properties of internode growth and how this simplified growth is affected by various carbon intake and growth respiration assumptions.

4.2 ODE Models of Internode Carbon Intake and Respiration

Our models will focus on the carbon allocation to internodes. In chapter 1, we already discussed that internodes have two phases of growth, phase I, growing in length only when LPI is less than or equal to 10 and phase II, expanding in diameter only when LPI is larger than 10. Usually during phase I, internodes may develop differences in their lengths, therefore, different volumes because we assume they have the same diameter at the end of phase I growth.

The internodes' growth depends on the carbon intake of the internodes and the respiration cost of the internodes. In the old transportation matrices-based model, the carbon respiration cost of internodes is assumed to be proportional to the internode volume. The intake carbon of internodes can be calculated directly from the transportation matrix. Each individual leaf passes an equal amount of carbon down to each of the internodes below the leaf. We assume that the intake of carbon for each individual leaf is some constant.

The change of an internode volume, V , is proportional to the net income of carbon.

$$\frac{dV}{dt} = I - O, \quad (4.1)$$

where I represents carbon received by the internode and O represents the amount of carbon lost due to maintenance respiration. For phase II growth, in which internodes only grow in diameter, we have the following equation,

$$\frac{d(\pi r^2 l)}{dt} = k - c \pi r^2 l,$$

where r is the radius of the cross section circle and l is the internode length. See the figure below.

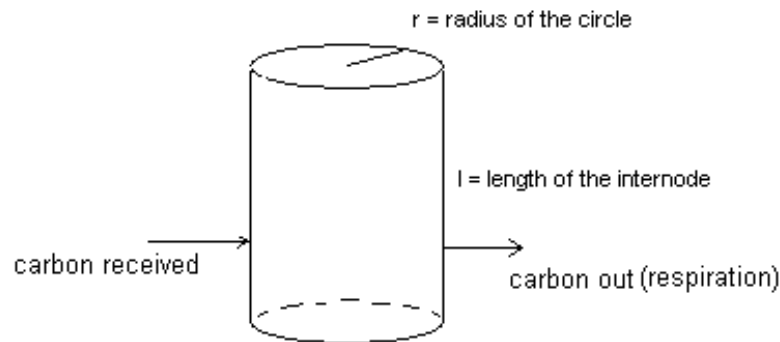


Figure 4.2 An ideal internode

Simplifying the above equation, we get

$$\frac{dr}{dt} = \frac{k}{2\pi l} \cdot \frac{1}{r} - \frac{c}{2} \cdot r$$

Obviously we have a positive stable equilibrium point $\bar{r} = \sqrt{\frac{k}{c\pi l}}$ to which r will

approach during the course of the year (assuming k constant). The length of the internode is evident in the solution of the equilibrium point, which suggests that different initial lengths at the beginning of phase II growth will cause the development of different diameters during phase II growth. This may be one of the reasons that we observe non-monotonicity of internode diameters. The following figure shows the diameter of each internode of a two-year tree in which it is assumed that the source leaves allocate carbon equally to every internode below them. We can see non-monotonicity between internodes.

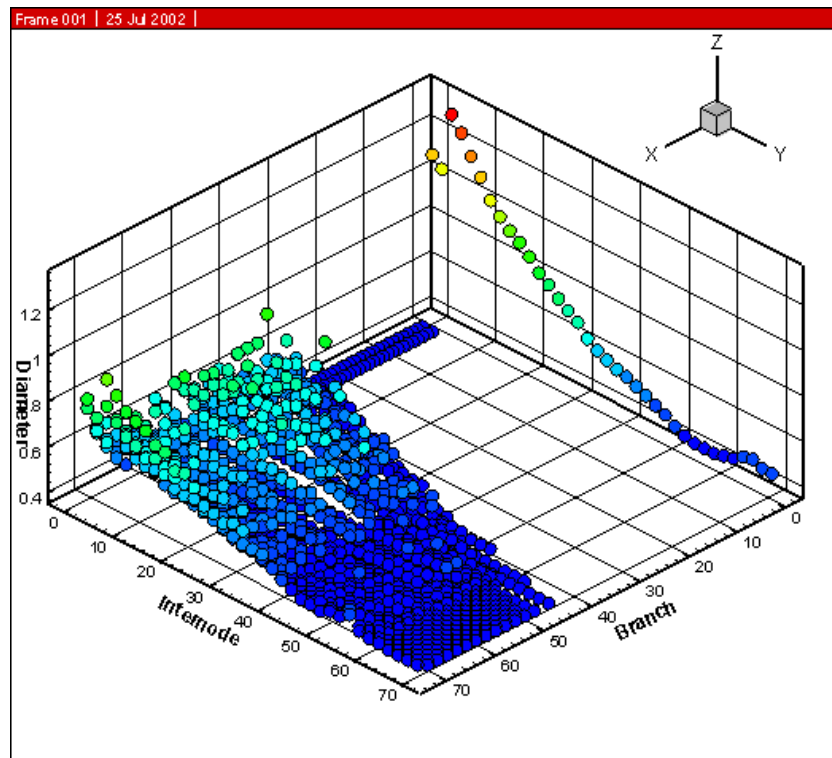


Figure 4.3 Different internode diameters of a two-year ECOPHYS tree

The x-axis denotes different branches with the lower sub-branches having lower x values. The rightmost one is the main trunk. The y-axis denotes different internodes with the lower internodes having lower y values. The z-axis denotes the diameter of each internode.

The non-monotonicity of internode diameters indicates that evenly distributing carbon to internodes is not a good strategy, and it also casts doubts on the assumption that respiration cost of carbon of internodes is proportional to the internode volume.

Because we have no sure evidence of how respiration costs relate to the geometric properties of internodes, we analyze all simple cases based on our assumptions. Let us suppose that the respiration cost of an internode is proportional to either the

internode length, the lateral surface area of internode cylinder, or the internode volume. Similarly, assume that the carbon that the internode receives is also proportional to the length, lateral surface area, or volume of the internode. Based on (4.1), we have the following models.

For carbon intake, we have three different choices:

(1) Carbon intake is proportional to internode length, i.e. $I = k_1 l$,

(2) Carbon intake is proportional to lateral surface area of internode cylinder, i.e.

$$I = k_2 C l,$$

(3) Carbon intake is proportional to internode volume, i.e. $I = k_3 A l$.

Here, k_1, k_2, k_3 are positive proportionality constants. We assume that no changes in the carbon source amount occur over time. Although for a real internode the carbon source amount varies during a growing season, i.e. k_i 's are non-constant functions of time, this constant carbon source assumption allows us to keep the mathematics simple. One could extend our analysis to k_i 's that are simple functions of time. C represents internode circumference and A represents internode cross-section area.

The same reasoning applies to the respiration cost term, O , except for different constants; we have $O = c_1 l$, or $c_2 C l$, or $c_3 A l$, depending on whether internode respiration is assumed to depend on internode length, surface area or volume. We can write down the equations as follows, using the fact that $C = 2 \pi r$ and $A = \pi r^2$, with r representing internode radius.

$$\frac{d(\pi r^2 l)}{dt} = \{k_1 l \text{ or } k_2 2\pi r l \text{ or } k_3 \pi r^2 l\} - \{c_1 l \text{ or } c_2 2\pi r l \text{ or } c_3 \pi r^2 l\}$$

After simplification, we get the following family of equations.

$$\frac{dr}{dt} = \left\{ \frac{k_1}{2\pi r} \text{ or } k_2 \text{ or } \frac{k_3}{2} r \right\} - \left\{ \frac{c_1}{2\pi r} \text{ or } c_2 \text{ or } \frac{c_3}{2} r \right\}$$

For the simplest cases, i.e. I and O both having only one term, we have a total of $3 \times 3 = 9$ cases, as shown in the following.

(1) Both intake and respiration are functions of internode length.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_1}{2\pi r} - \frac{c_1}{2\pi r}$. Obviously, $k_1 > c_1$ is needed

for internode growth. Furthermore $r(t) = r_0 \frac{k_1 - c_1}{2\pi} \sqrt{t}$. Observe that internodes with

differing initial internode radii will see an increasing difference in their radii as time increases.

(2) Intake is a function of internode length and respiration is a function of lateral surface area.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_1}{2\pi r} - c_2$. Obviously, we have a unique

stable equilibrium point $\bar{r} = \frac{k_1}{2\pi c_2}$. Internodes with differing initial radii will see a

convergence to the same radius as t increases.

(3) Intake is a function of internode length and respiration is a function of internode volume.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_1}{2\pi r} - \frac{c_3}{2} r$. Obviously, we have a unique

positive stable equilibrium point $\bar{r} = \sqrt{\frac{k_1}{\pi c_3}}$. Internodes with different initial radii

will see a convergence to the same radius as time increases.

(4) Intake is a function of internode lateral surface area and respiration is a function of internode length.

The corresponding equation reads $\frac{dr}{dt} = k_2 - \frac{c_1}{2\pi r}$. Obviously, $k_2 > \frac{c_1}{2\pi r}$ is

needed for internode growth and we have only an unstable equilibrium point

$\bar{r} = \frac{c_1}{2\pi k_2}$. Internodes with differing initial internode radii will see an increasing

difference in their radius as time increases.

(5) Intake and respiration are both functions of internode lateral surface area.

The corresponding equation reads $\frac{dr}{dt} = k_2 - c_2$. Obviously, $k_2 > c_2$ is needed for

internode growth. Furthermore, $r(t) = r_0 + (k_2 - c_2)t$. The internodes with differing

initial internode radii will see an increasing differences in their radius as time increases.

(6) Intake is a function of internode lateral surface area and respiration is a function of internode volume.

The corresponding equation reads $\frac{dr}{dt} = k_2 - \frac{c_3}{2} r$. Obviously, $k_2 > \frac{c_3}{2} r$ is needed

for internode growth and we have a stable equilibrium point $\bar{r} = \frac{2 k_2}{c_3}$. Furthermore,

$r(t) = \frac{2 k_2}{c_3} + r_0 e^{-\frac{c_3}{2} t}$. Internodes with differing initial internode radii will

converge exponentially to $\bar{r} = \frac{2 k_2}{c_3}$ as time increases.

(7) Intake is a function of internode volume and respiration is a function of internode length.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_3}{2} r - \frac{c_1}{2 \pi r}$. Obviously, we have only an

unstable positive equilibrium point $\bar{r} = \sqrt{\frac{c_1}{\pi k_3}}$. Internodes with differing initial

internode radii will see an increasing differences in their radius as time increases.

(8) Intake is a function of internode volume and respiration is a function of internode lateral surface area.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_3}{2} r - c_2$. Obviously, we have only an

unstable positive equilibrium point $\bar{r} = \frac{2 c_2}{k_3}$. Furthermore, $r(t) = \frac{2 c_2}{k_3} + r_0 e^{\frac{k_3}{2} t}$.

Internodes with differing initial internode radii will see an exponentially increasing difference in their radius as time increases.

(9) Intake and respiration are both functions of internode volume.

The corresponding equation reads $\frac{dr}{dt} = \frac{k_3}{2}r - \frac{c_3}{2}r$. Obviously, $k_3 > c_3$ is needed

for internode growth. Furthermore, $r(t) = r_0 e^{\frac{k_3 - c_3}{2}t}$. Internodes with differing initial internode radii will see exponential increasing differences in their radii as time increases.

Among all these nine cases, we have only three (cases 2, 3 and 6) that have stable fixed points. That means, biologically, under those assumptions, an internode's diameter will not explode. Instead, in the long run, the diameter will develop to some value that is independent of initial conditions. This is desirable because with a constant carbon source, the radii of internodes with different initial radius will not diverge. Furthermore, case 3 and case 6 both assume that respiration is a function of internode volume, which corresponds to the algorithm that the current code implements. This also motivates us to assume that internode carbon intake is a function of internode length or internode lateral surface area. Therefore, we are motivated to try the following simulations.

(1) Simulation of case 3: Carbon intake is proportional to internode length and respiration is a function of internode volume (Figure 4.4). This is the result of a two-year tree simulation on bud set day. As in the figure 4.3, the x-axis denotes different branches, the y-axis denotes different internode and the z-axis denotes the diameters of each internode.

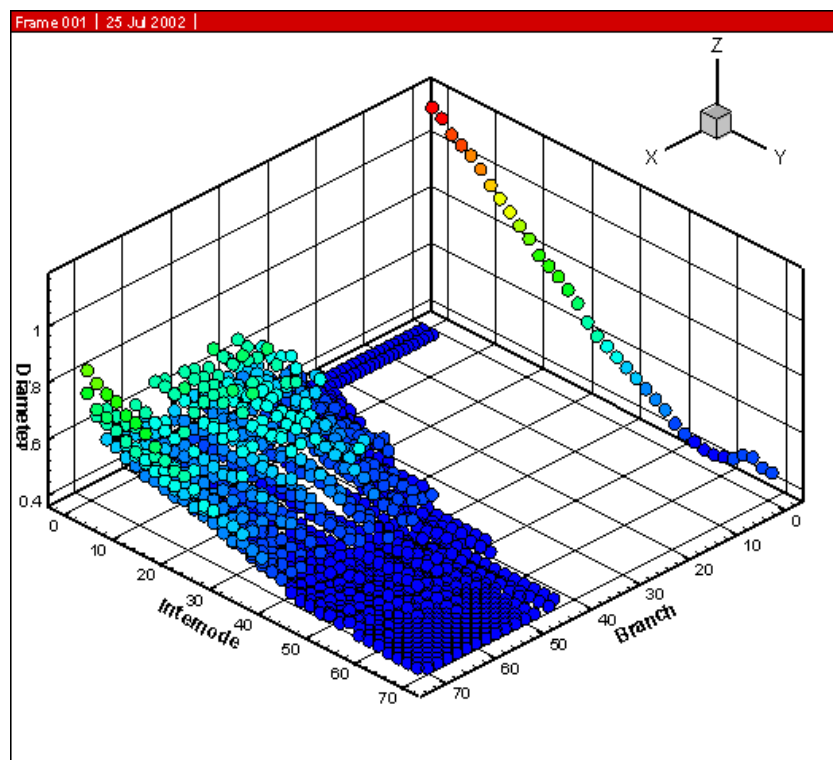


Figure 4.4 Different internode diameters of a two-year ECOPHYS tree with the case 3 carbon intake / respiration model.

(2) Simulation of case 6: Intake is a function of internode lateral surface area and respiration is a function of internode volume (Figure 4.5). This is also the result of a two-year tree simulation on bud set day.

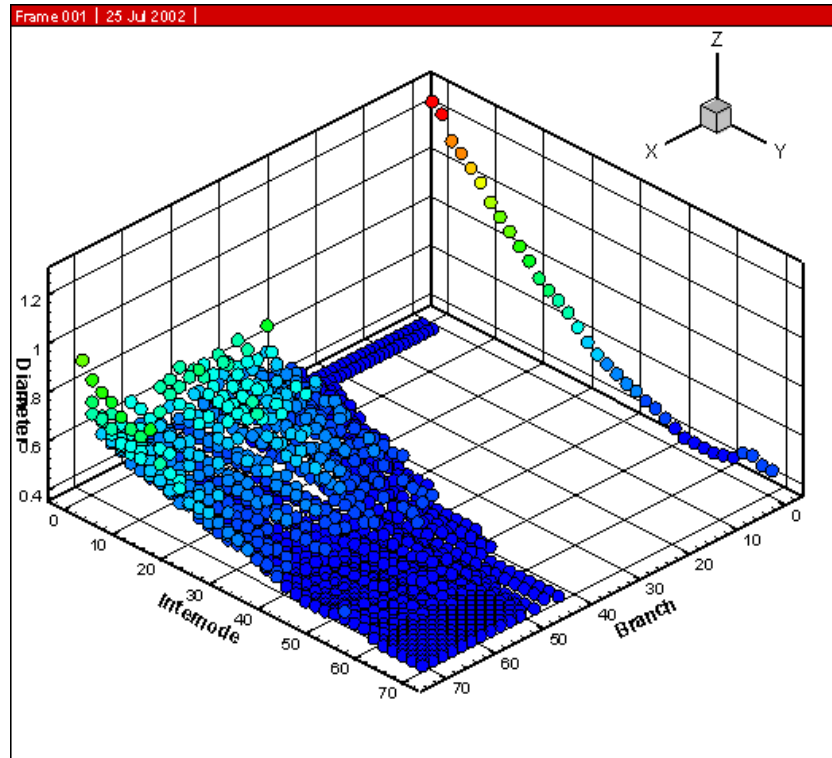


Figure 4.5 Different internode diameters of a two-year ECOPHYS tree with the case 6 carbon intake / respiration model.

For comparison, please refer to Figure 4.3, the simulation of old carbon intake / respiration model. We can see from the graphs that both length and surface allocation are better than the even allocation in that the number of incidents of internode diameter non-monotonicity has been reduced. Yet visual inspection under OpenGL shows that non-monotonicities happen at the junctions of branches. To solve this problem, one might extend the allocation all the way down to the bottom of the main trunk instead of just within a branch.

4.3 Constant Carbon Allocation and Its Implementation

The constant carbon allocation strategy, described in this section, is a prototype strategy for modeling carbon allocation with more fidelity to plant physiologists' understanding of the process. This prototype strategy was motivated by discussions of the plant physiology of carbon allocation with Jud Isebrands, George Host, Mark Kubiske, Evan McDonald, Kathryn Lenz, Harlan Stech and I [19, 21].

Constant allocation means that each specific source leaf downward-transport carbon to the bottom of the main trunk with each unit length (or surface area) of internode receiving an equal amount of carbon. The following figure illustrates the idea of constant allocation. [19,21]

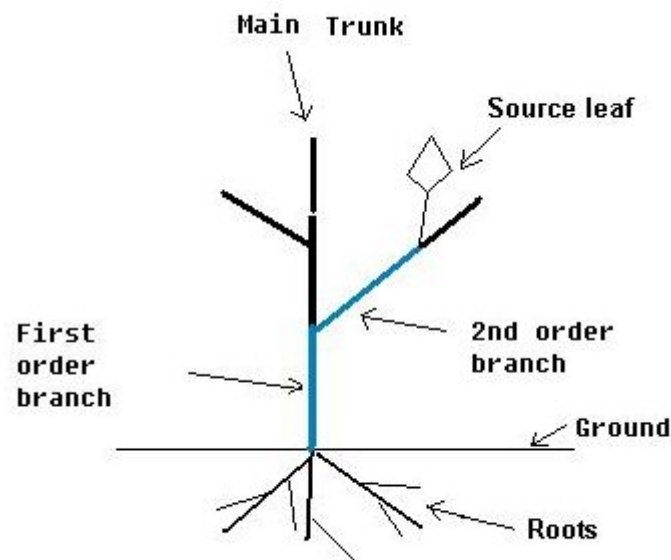


Figure 4.6 An active internode chain for constant allocation

In the figure, the blue (lower gray level) part denotes the internodes to which the source leaf will transport carbon.

The implementation of the constant allocation algorithm for a multi-year tree was designed into the transportation module. Previously, by modifying the upward internode transportation matrix [refer to Appendix part 2], the transportation module implements upward and downward internode transportation simultaneously, because previously the downward internode transportation is within a branch. So modifying the matrix is a normal approach. But this method won't work for the "constant allocation" algorithm because by doing "constant allocation", the carbon will transfer crossing the branches.

To fulfill the requirement of transferring carbon crossing the branches, we introduce an idea of "active internodes chain". For example as shown in above figure, the blue (lower gray level) part is an "active internode chain" for designated leaf. This chain consists of partial blocks of two different branches. The two-way link list structure of the tree makes it possible to express and traverse the "active internodes chain", although it is not so convenient. The chain starts from the internode that our interested leaf attaching on and ends at the first (base) internode of the main trunk. To implement "constant allocation", we first calculate the amount of carbon that our interested leaf will transport out to the downward internodes. Then we calculate the total length of the chain. Dividing the carbon by the length, we get the amount carbon that the unit length internode will receive, i.e., our "constant". Then we traverse the chain from the start internode to the end internode, at each internode, we multiple the constant to the internode length and get the amount of carbon each internode will receive. Please refer to the Appendix part 5, the code, for detail on implementation.

Actually, using the same procedure described above, we can implement the downward carbon to the parent branch that we discussed in section 4.1.

The following diameter plot is the result of the constant allocation approach for a two-year tree simulation. It solves the non-monotonicity problem.

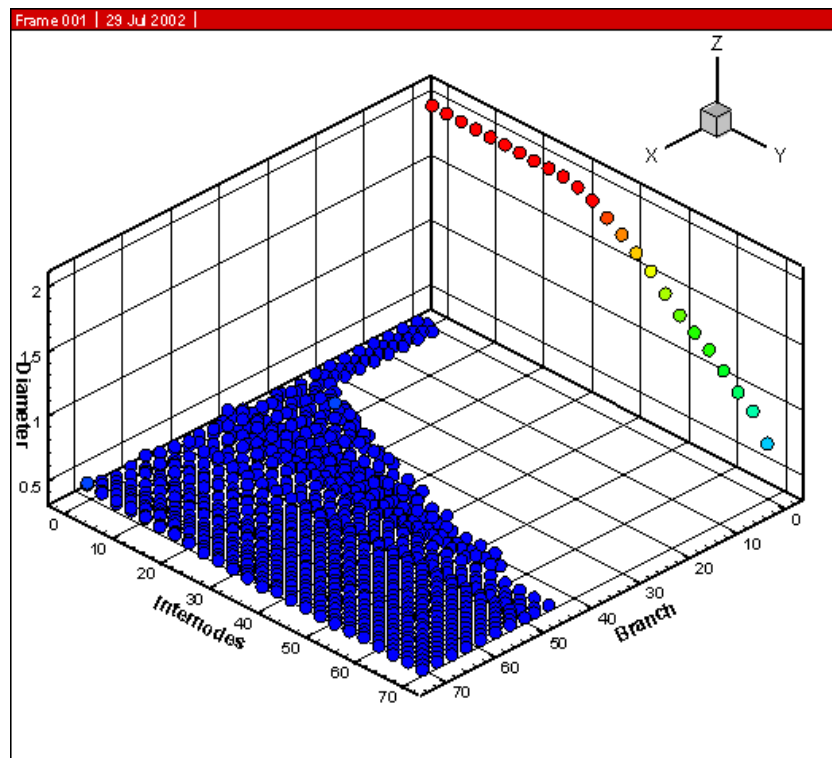


Figure 4.7 Different internode diameters of a two-year ECOPHYS tree with the constant allocation strategy.

The following figures (Figure 4.8 and Figure 4.9) are the change of diameters of the main trunk in two-year and three-year simulations. From the figures, we can see that the diameter of the main trunk is convergent to some constant in each growing season. This is compatible with our ODE models, which claim that the diameter of an internode has a positive equilibrium point.

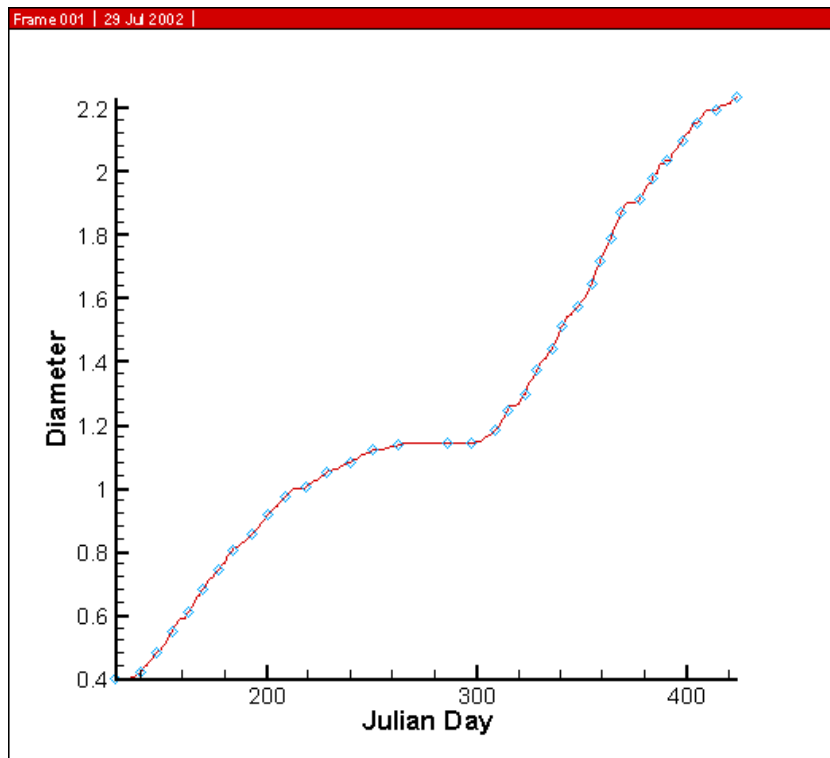


Figure 4.8 A two-year simulation of diameter changes

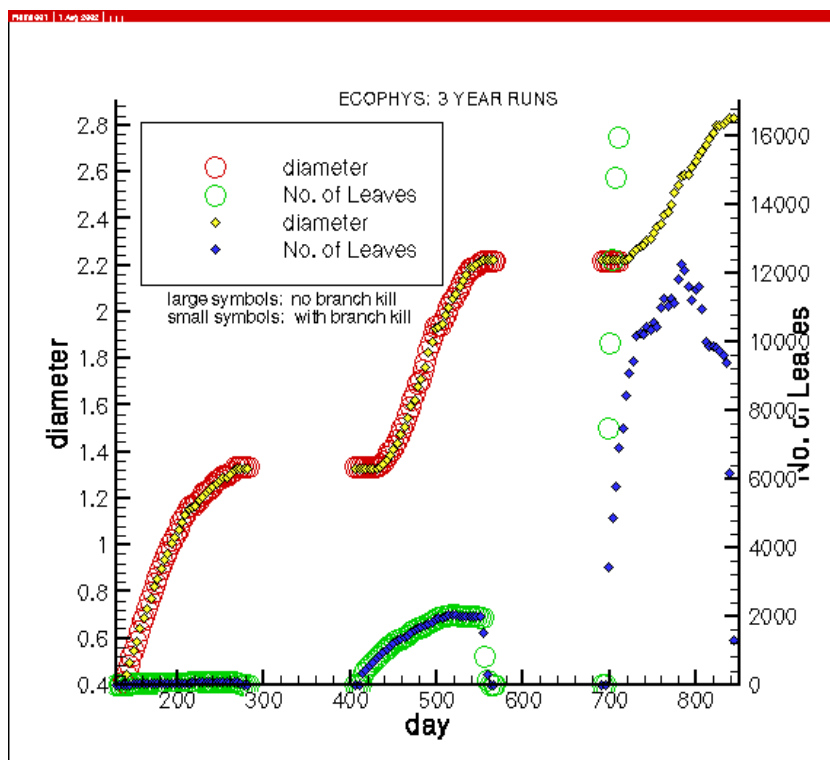


Figure 4.9 A three-year simulation of diameter changes

Chapter 5. Simulation and Comparison

In this chapter we will do some numerical tests: (1) How the ScanDensity parameters in the DCBA algorithm that we discussed in Chapter 3 affect the model speeds and model predictions. (2) A comparison between the old shading algorithm and the DCBA algorithm. A possible bug in the old shading algorithm will be pointed out. (3) A comparison between DCBA and Bit-DCBA.

5.1 ScanDensity vs. Model Speed and Model Predictions

As we discuss in Chapter 3, the ScanDensity will affect the accuracy of DCBA algorithm. Obviously, the ScanDensity will also affect the speed of DCBA too. In order to get a general idea of the speed change with different ScanDensities, we do two-year runs with different ScanDensities 8, 10, 12, 14 and 16. The following is the data set we get.

ScanDensity	8	10	12	14	16
Total Two-Year Simulation Runtime (seconds)	599	631	665	718	774

Table 5.1 Two-year simulation times with different ScanDensities

The simulation is done on a computer of type Dell Dimension 4100, with PIII 866 M Hz CPU, 384M physical memory, 512 K Cache, 13.6 G 7200 RPM hard drive and 133 M Hz peripheral speed. Using tecplot, we get the following figure.

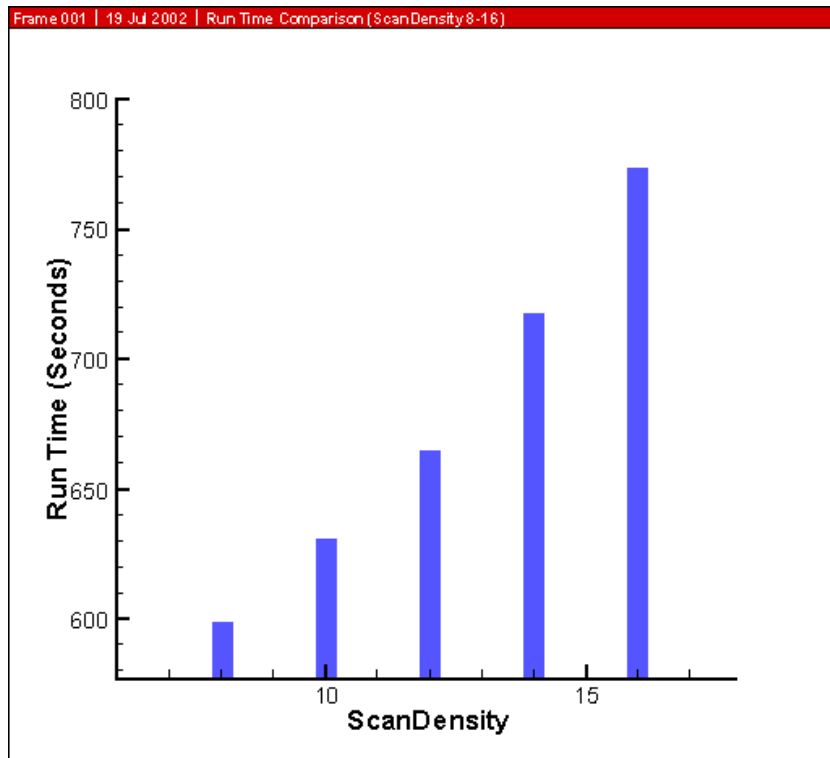


Figure 5.1 Histogram of two-year simulation times with different ScanDensities

In order to find the proper ScanDensity at the sense of compromise between speed and accuracy, we do two-year simulations using different ScanDensities of 8, 12 and 16. For each ScanDensity, we record the accumulating run time on every Julian day to demonstrate the speed, and we also record the height and the total leaf number of a tree to demonstrate the model prediction. The following are the results.

(1) ScanDensity vs. Runtime

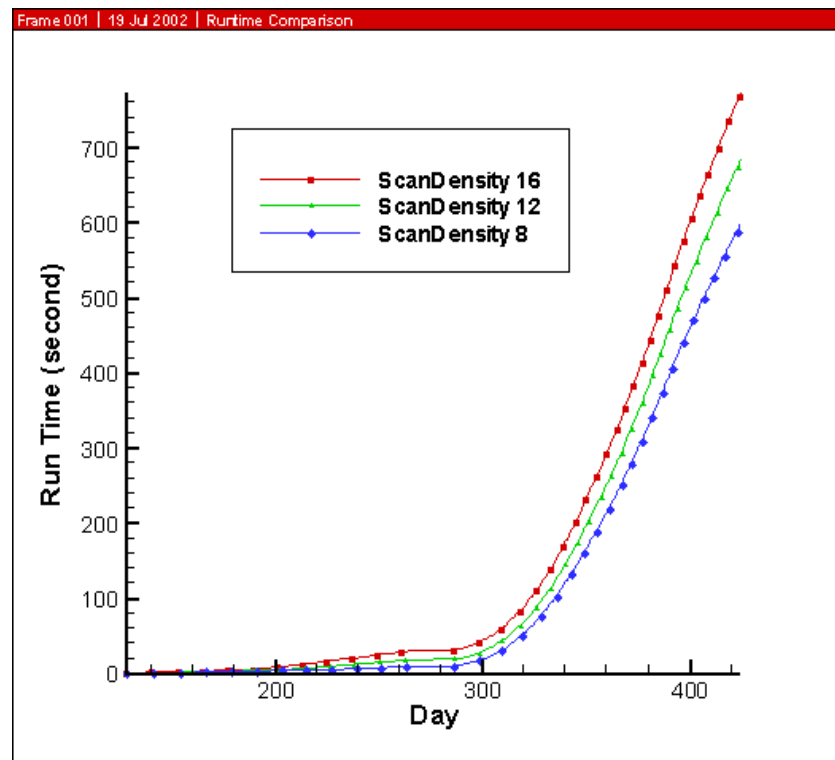


Figure 5.2 Simulation time with different ScanDensities

From both Figure 5.1 and Figure 5.2, we can see a simple relationship between ScanDensity and Run time: the higher ScanDensity, the more time is needed. From Figure 5.2, we can also find that no matter what ScanDensities are, each curve follows the same pattern. This is true because the leaf number of a tree is exponential increasing when the Julian day increases; the runtime of DCBA is roughly proportional to the number of leaves in a tree. Therefore, the whole tree simulation time increases exponentially.

(2) ScanDensity vs. Tree Height

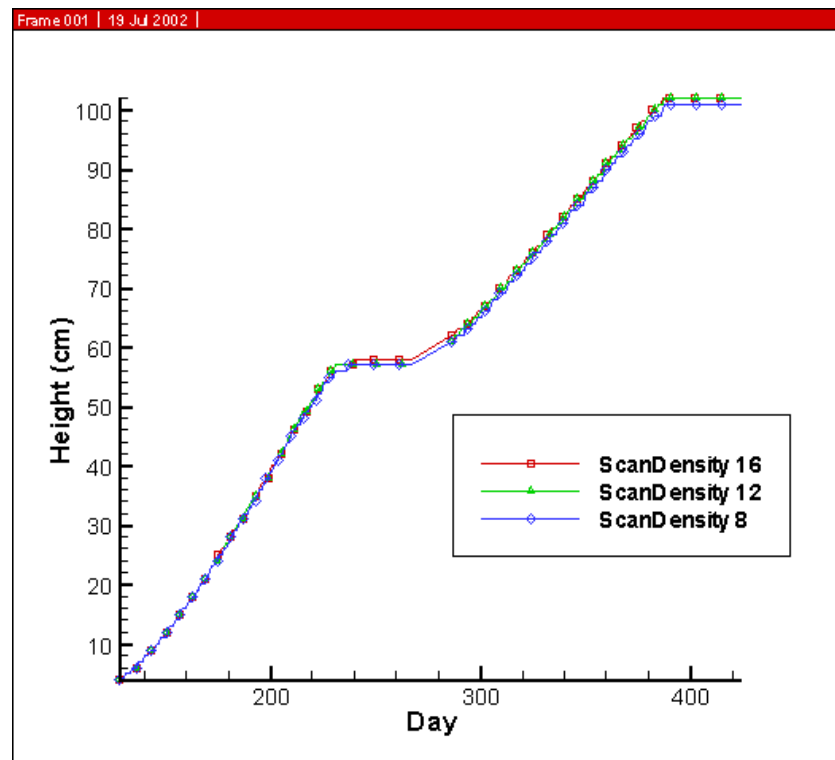


Figure 5.3 The comparison of tree height predictions when using different ScanDensities

Tree height is an important model predictor. We can see from the figure above that different ScanDensities report almost the same tree heights. This suggests that a ScanDensity as low as 8 is acceptable as a representative of higher ScanDensities.

(3) ScanDensity vs. Leaf Numbers

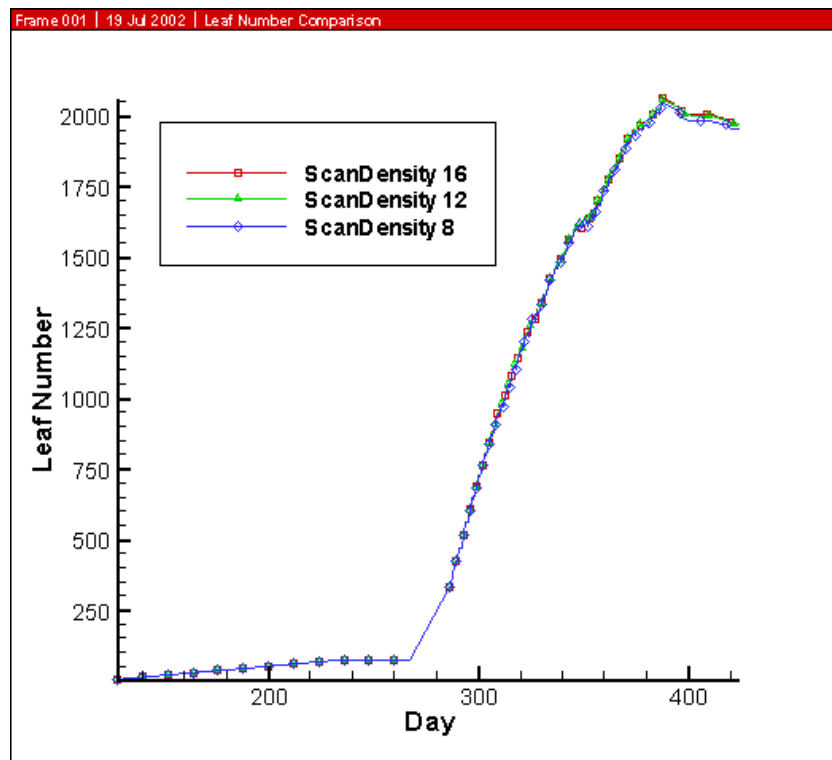


Figure 5.4 The comparisons of different leaf numbers when using different ScanDensities

Leaf numbers during the growing season is also an important data set that ECOPHYS predicts. The above figure tells us, although the higher ScanDensity tends to predict more leaf numbers, the difference between the leaf numbers that ScanDensity 16 predicts and leaf numbers ScanDensity that 8 predicts is so small that we can ignore it. This confirms that ScanDensity as low as 8 is acceptable for the ECOPHYS simulation.

5.2 Old Shading Algorithm vs. DCBA

In order to compare the speeds of two different shading algorithms, we set up an experiment that runs two shading algorithms at same time, so that we can measure and compare the shading time of each algorithm in every day during the first growing season. We set ScanDensity of both algorithms as 10. On each day, we run old algorithm first, but do not use that shading result. Then we run the DCBA and use this shading result to develop the canopy. This will ensure that both algorithms are running with the same canopy every day. The following figure is the result we get.

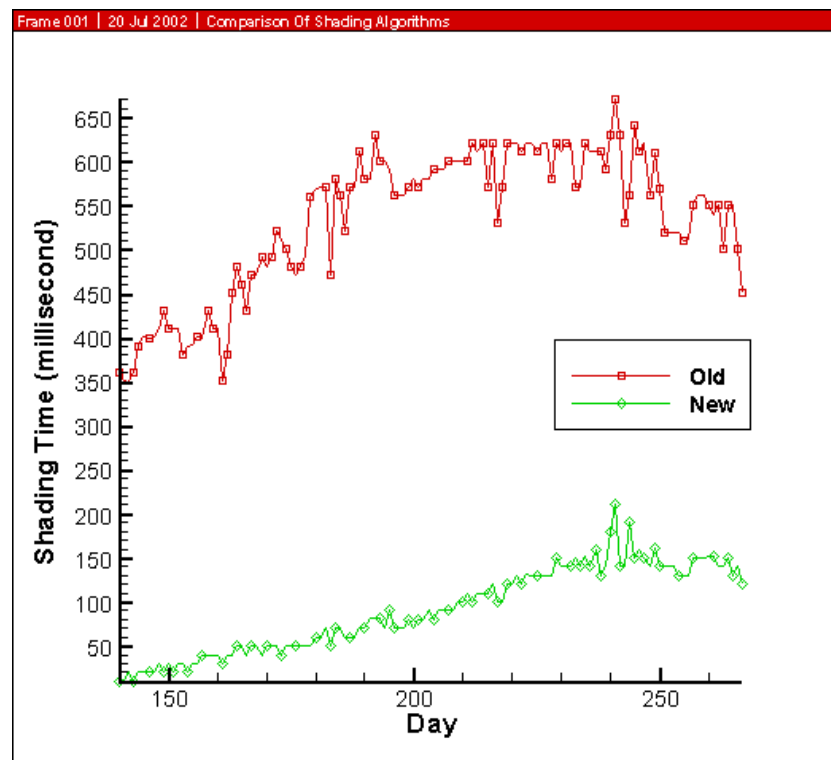


Figure 5.5 Daily Shading time comparison between DCBA and old algorithm

From the figure we can see that:

- (1) Both algorithms need more shading time when the leaf number on a tree increases. This is as expected.
- (2) The DCBA spends only 25% time of old algorithm does. At neighborhood of bud set day, when a tree has maximum number of leaves, the average shading time of DCBA is 150 milliseconds per day. While the average shading time of old algorithm is 600 milliseconds per day.

A two-year simulation described above was attempted. But a possible bug in the old shading algorithm makes a two-year simulation unnecessary. The reason follows.

We can do simulations of the first growing season with both shading algorithms (old one and DCBA) to see how different ScanDensities will affect the model predictions of total leaf area. Both simulations have the same parameters and the same environmental conditions; the only difference is the shading algorithm. In addition, leaf-dropping procedure has been switched off; therefore, both simulations have the same prediction on leaf number, 71. The following is the data we get.

ScanDensity	3	8	10	20	50	100
DCBA (Square cm)	1017.20	1214.54	1227.86	1260.7	1289.25	1297.49
Old (Square cm)	914.05	1002.46	949.61	649.91	371.54	274.27

Table 5.2 The comparisons between DCBA and old algorithm of total leaf areas when using different ScanDensities

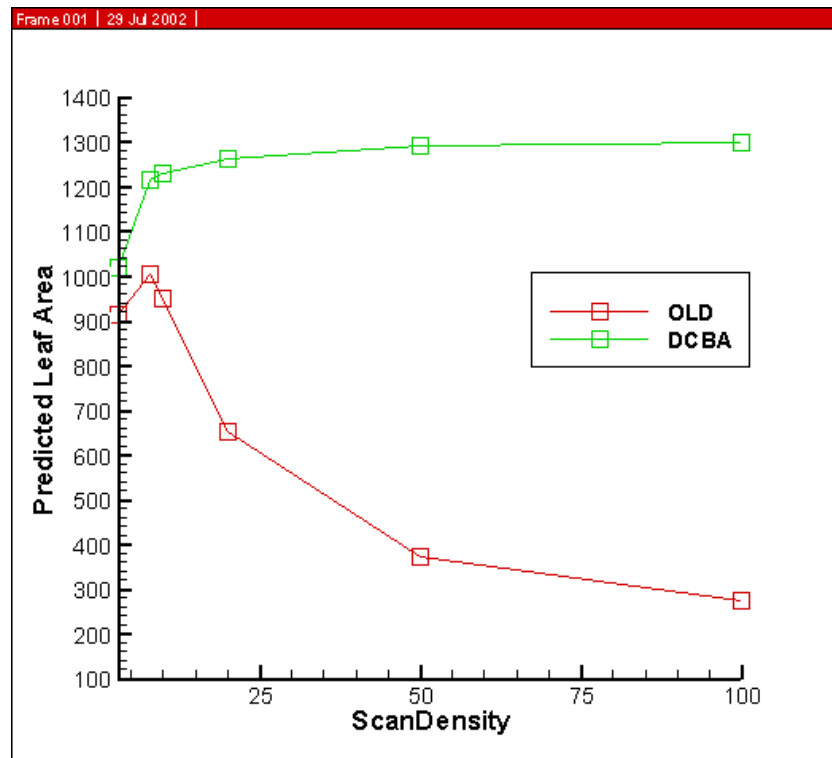


Figure 5.6 The graph of comparisons between DCBA and old algorithm on predications of total leaf area

From the data and above figure, we can see that:

- (1) For DCBA, the predicted leaf area is asymptotically convergent. In addition, the difference between prediction of ScanDensity 10, which is 1227.86 and the prediction of ScanDensity 100, which is 1297.49, is relatively small; the relative error is less than 1%. This indicates that ScanDensity 10 is sufficient for our simulation. Also, the average single leaf area predicted by DCBA is about 17 square cm. This is consistent with our experience.

(2) For old algorithm, the bigger ScanDensity, the lower the prediction of total leaf area. And the differences between different predictions of different ScanDensities are big and therefore, unacceptable. In addition, the average leaf area predicted by ScanDensity 100 is only 4 square cm. This is not true in real life.

The above simulations suggest that the DCBA algorithm is convergent for large ScanDensities. The simulations also suggest that a possible bug exists in the old algorithm. When we keep ScanDensity less than 10, the old algorithm only needs one canvas square [16] and everything is ok. Although the prediction of total leaf area is low, it is still acceptable. This also validates that our first year comparison of the speed between DCBA and the old algorithm is reasonable. However, when we increase the ScanDensity to larger numbers, the old algorithm needs more than one canvas square. The projection of leaves crossing the canvas squares may be the source of error.

In answer to our original question, when we do a two-year simulation the old algorithm will need more than one canvas square. So the same error will happen.

5.3 DCBA vs. Bit-DCBA

As part of the work of this thesis, the Bit-DCBA algorithm has been encoded. The following figures provide comparisons of the speed between DCBA and Bit-DCBA. We can see that Bit-DCBA is faster than DCBA, especially when used in conjunction with the quick sort algorithm.

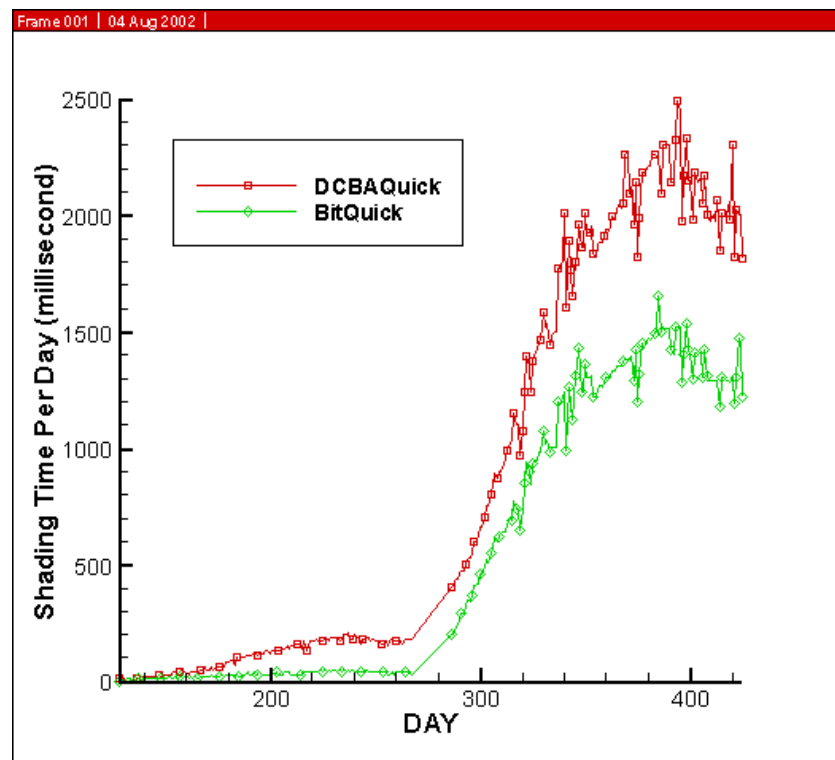


Figure 5.7 The daily shading time comparison between DCBA and Bit-DCBA when implementing quick sort

This figure shows the difference between DCBA and Bit-DCBA after we implement the quick sort algorithm and use one bit to denote each pixel. The following figure shows the difference between DCBA and Bit-DCBA when we implement the simple select sort algorithm and use a character to denote each pixel.

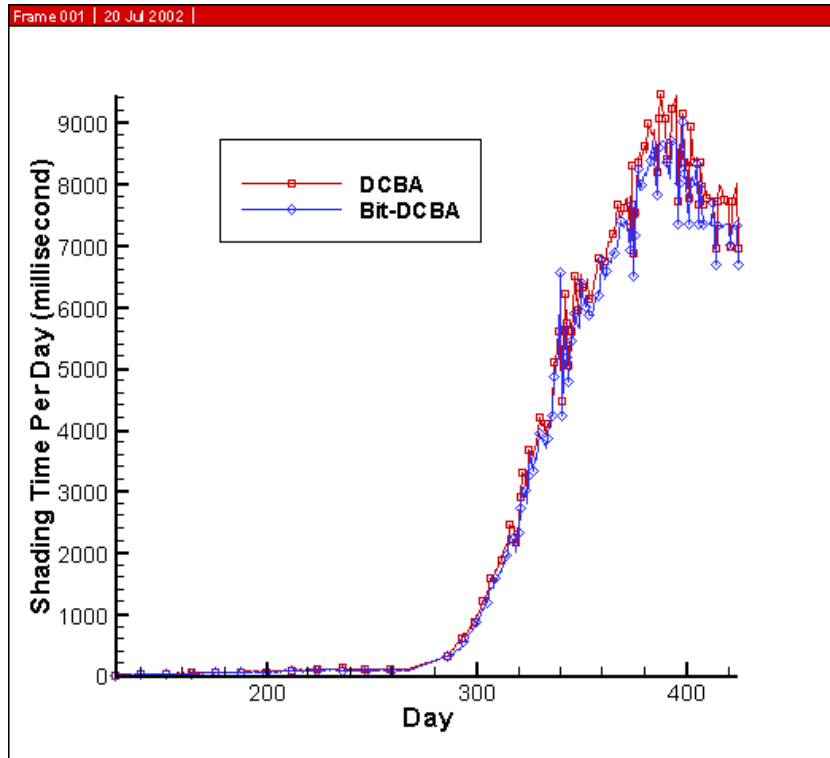


Figure 5.8 The daily shading time comparison between DCBA and Bit-DCBA when implementing simple select sort

We can imagine that the Bit-DCBA has a promising future in ECOPHYS, because it is not only memory saving, but is also faster. Also by comparing these two figures, we can see big improvements by using the better sorting algorithm. This will be discussed in the next section.

5.4 Quick Sort vs. Shading Speed

At the very beginning of the development of the DCBA algorithm, we use the simple select sorting algorithm. Although DCBA is faster than the old shading algorithm, the observation of a quadratic relationship between shading times and total leaf number motivates us to implement a quick sort algorithm in the sorting step of DCBA. The following two figures (Figure 5.9 and Figure 5.10) are comparisons between the simple select sort and quick sort.

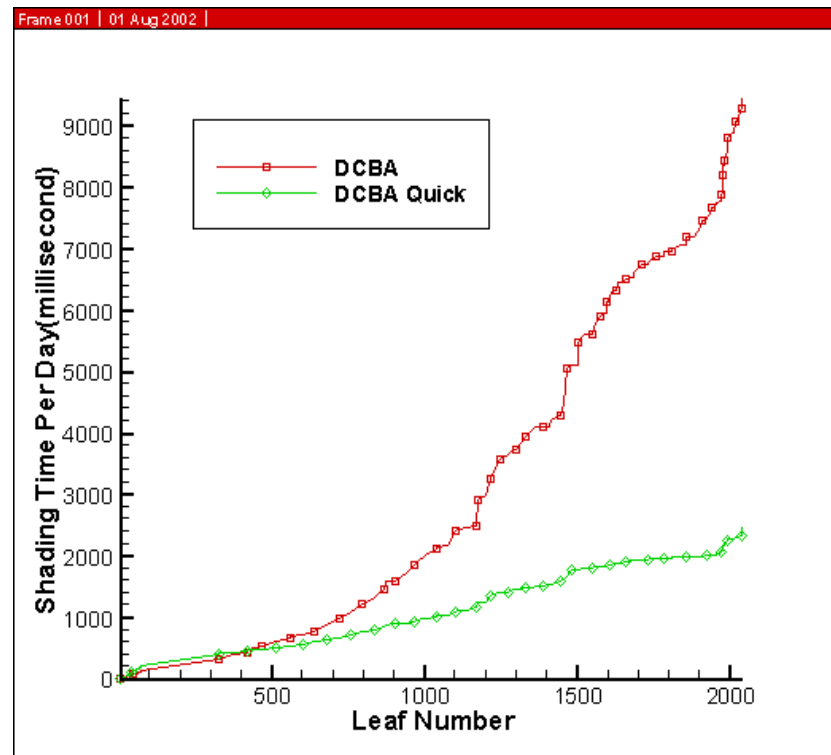


Figure 5.9 Quadratic relationship between daily shading time and leaf number when implementing simple select sort and linear relationship when implementing quick sort (two-year simulations).

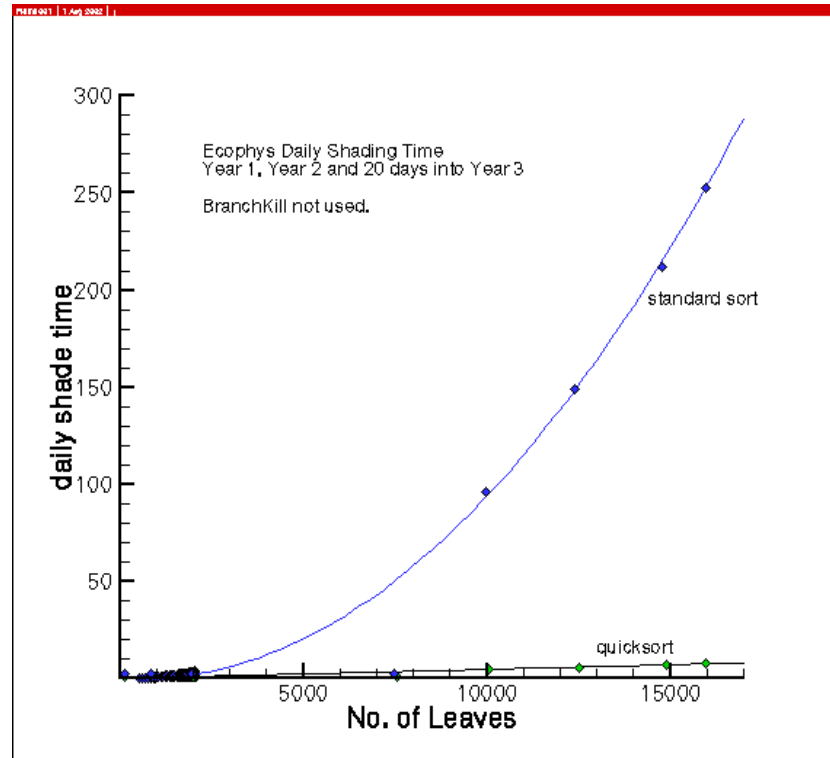


Figure 5.10 Quadratic relationship between daily shading time and leaf number when implementing simple select sort and linear relationship when implementing quick sort (three-year simulations).

We can see clearly from the above two figures that when we use simple select sort, the shading time and total leaf number has a quadratic relationship, i.e. the shading time grows quadratically when the leaf number increases, while when we use the quick sort algorithm, the shading time and total leaf number has a simple linear relationship.

Our observations have a plausible explanation. We know that the complexity of the simple select sort algorithm is $O(n^2)$ and this can explain our quadratic observations.

Furthermore, it also tells us that when the leaf number gets larger, the sorting step, rather than scan conversion algorithm, is the “bottle neck” of DCBA. The complexity of the quick sort algorithm is $O(n \log n)$, which is not linear. But consider that our recorded shading times are whole day shading times. In chapter 3, we already discussed that from hour to hour, the relative leaf positions have only changed slightly. Hence when we do the sorting for the leaf pointers, we may only need to change a very small number of them, except for the first sunlit hour of a day. This may be the reason that we observe a linear relationship between shading time and leaf number. The following figure includes a regression linear function of shading time and leaf number. The following figure includes a regression linear function of shading time and leaf number.

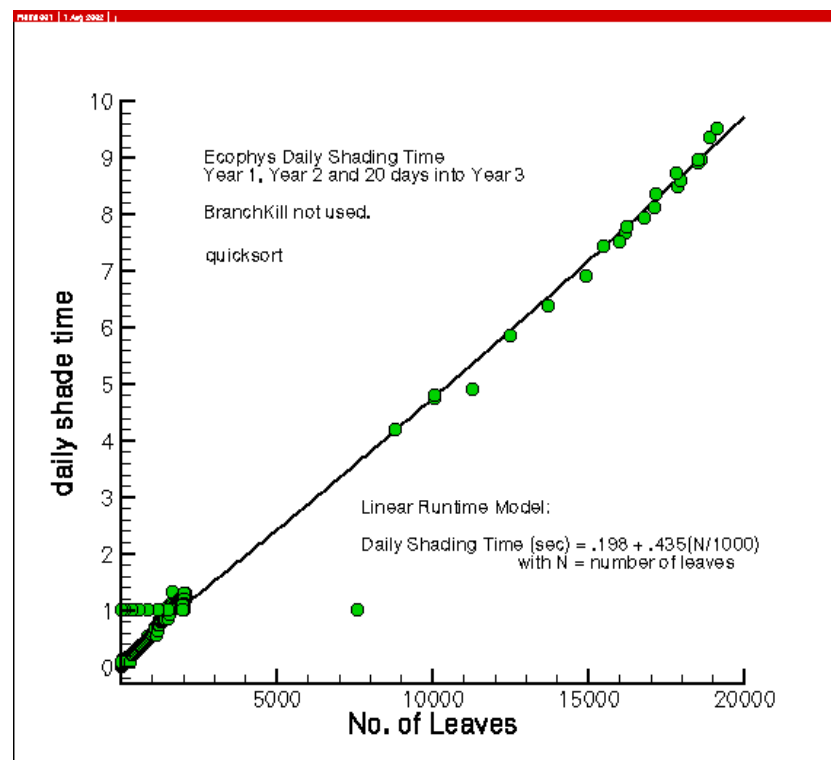


Figure 5.11 Linear relationship between daily shading time and leaf number

The implementation of the quick sort algorithm into DCBA increases its performance dramatically, especially when the total number of leaves is large. The following figures give us a general idea of the speed improvement.

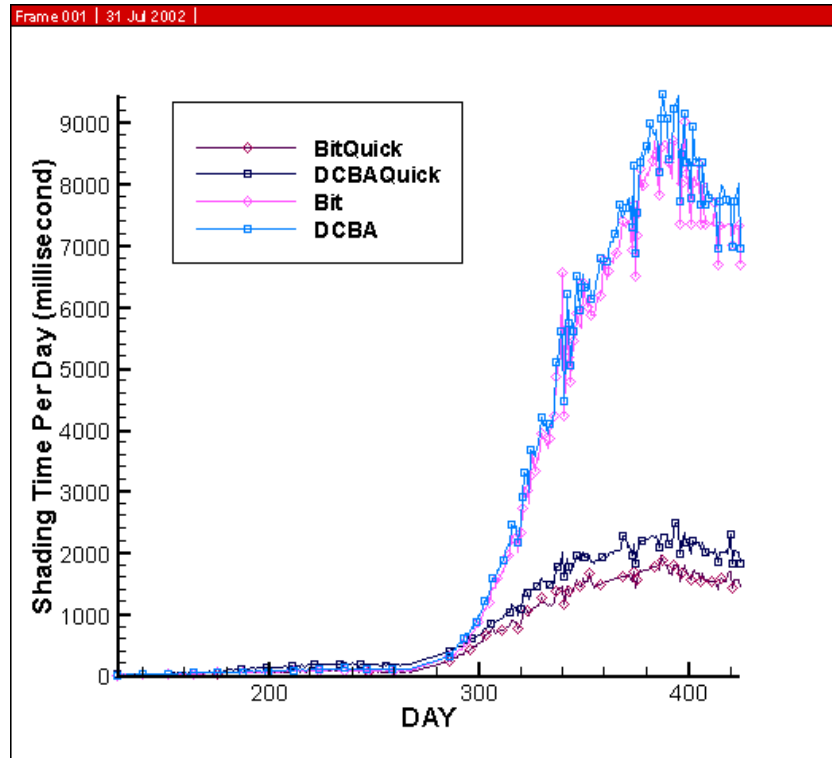


Figure 5.12 Comparisons between DCBA and Bit-DCBA when using different sorting algorithms

Figure 5.12 is a two-year simulation of DCBA and Bit-DCBA with two different sorting algorithms. The top two curves are the results using the simple select sort algorithm while the lower two curves are the results using the quick sort algorithm. We can see roughly that DCBA with quick sort uses only 25% of the time that simple select sort does.

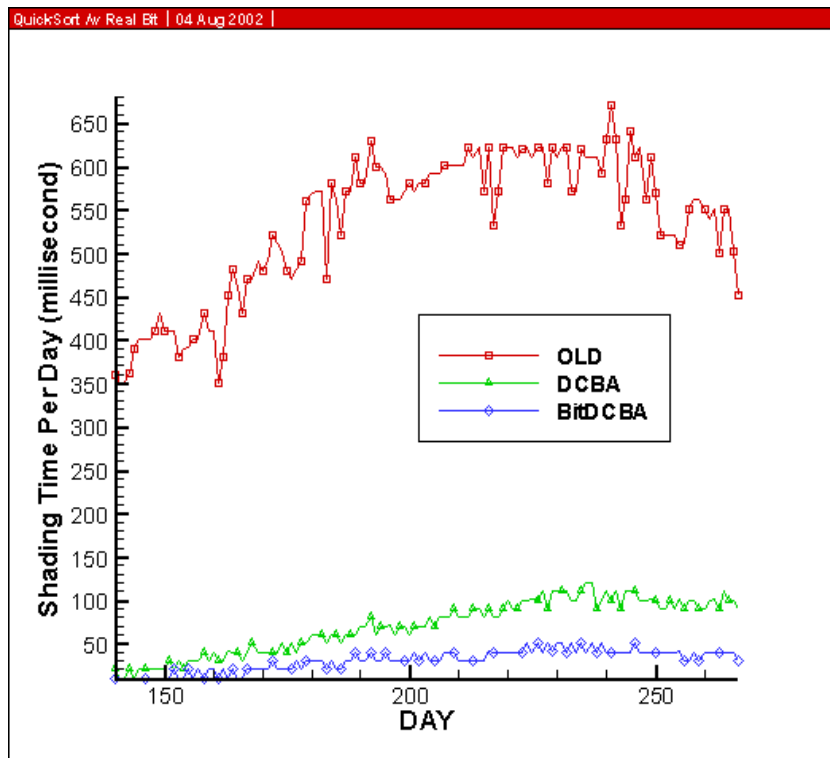


Figure 5.13 Daily shading time comparison between old, DCBA and Bit-DCBA

We can see from Figure 5.13 that DCBA with simple select sort and Bit-DCBA with quick sort both dramatically improved shading speed. Particularly, we can see that roughly the Bit-DCBA with quick sort uses only 7% of the old shading time.

For comparison, we do two-year simulations of the DCBA and Bit-DCBA with quick sort using different ScanDensities. The following is the data we get. We use the same computer described in Section 5.1.

ScanDensities	8	10	12	14	16
DCBA (second)	222	266	312	366	427
Bit-DCBA (second)	203	218	236	258	282

Table 5.3 Simulation time comparisons between DCBA and Bit-DCBA

Combining Table 5.1 and Table 5.3, we have the following figure.

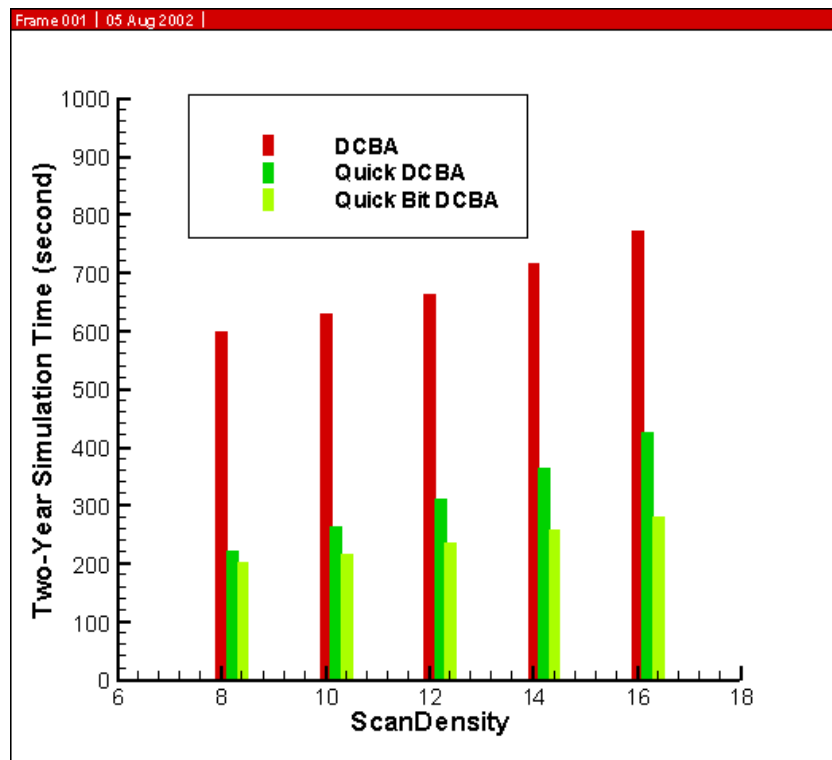


Figure 5.14 Comparison of DCBA and Bit-DCBA

In short, the quick sort algorithm dramatically improves the speed of DCBA, especially when we have a large number of leaves.

Chapter 6. Conclusions and Further Study

In this thesis, the contributions that have been made can be classified into two categories: coding and modeling.

In the coding part, three different versions of codes have been developed. Each version has its special strengths. The COM-less version is friendly for those who are familiar with C++ but not COM. The COM version is designed for parallel computation based on DCOM and preparing an interface for weather simulation. Linux version has been developed for the future parallel computation using the UMD Beowulf Cluster. Huge memory leaking in the previous code has been corrected. A new clearer and cleaner simulation module has been written. Code structures have also been analyzed.

The contributions to the modeling part come into two varieties: new shading algorithms (DCBA and Bit-DCBA) and a new differential equation motivated carbon allocation strategy. A careful comparison based on simulations of DCBA and old algorithm suggests that the old algorithm may have bugs. Bit-DCBA is good at both speed and memory saving compare to DCBA. The transportation matrices model of internode downward carbon transportation has been thoroughly studied. A class of simple differential equation models of internode growth has been studied. As a result, and after several brainstorming meetings with G. Host, J. Isebrands, M. Kubiske, E. McDonald, K. Lenz and H. Stech, a carbon allocation strategy called “constant allocation” has been developed and implemented into the ECOPHYS multi-year simulation. It solves the previously observed non-monotonicity in internode growth that exists in the matrices model.

The topics for the future study can also be classified into two categories: coding and modeling.

For the coding part, some interesting, yet challenging topics are listed below:

- (1) Implementation of parallel computation on Beowulf Clusters.
- (2) Packaging the DCBA and / or Bit-DCBA algorithm into COM.
- (3) Dr. Lenz [19] points out a bug that we may be losing photosynthate after budset. In particular, we are sending photosynthate to leaves with LPI 0-4, even though after branch budset there may be no leaves with these LPI values.

For the modeling part, possible topics include:

- (1) For the differential equation model with a constant carbon intake, we assume that carbon intake and respiration are functions of only one of three choices (internode length, internode lateral surface area and internode volume). It would be interesting to discuss the situation in which allocation and respiration are functions of more than one choice. The general equation would have the form $\frac{dr}{dt} = k_1 \cdot \frac{1}{r} + k_2 + k_3 r$. The study of how the parameters decide the stable equilibrium points may shed additional light on our modeling of internode carbon allocation and internode growth. Furthermore, it would be interesting to consider similar allocation / respiration growth models for leaves and roots.

(2) The estimation of diffuse sunlight (EDS). In the present model, the diffuse sunlight is estimated by the LAI and Beer's law. However, the current estimation of leaf area "above" any particular leaf is faulty, relying on a notion of "higher" defined by the recursive traversal of the canopy, rather than on actual height. A possible alternative estimation method is to calculate how much blue sky can be seen from a leaf through the canopy. The leaf position, i.e. the viewpoint in this estimation, can be measured roughly by a branch ID and an internode ID. Therefore, we can attempt to generate an empirical formula to estimate the diffuse sunlight very quickly. Actually, it was this idea that motivated me to find a quick shading algorithm, because we need a very quick shading algorithm to fulfill the need of extensive shading computations required to estimate the portion of blue sky.

(3) Right now our model is a deterministic model. When and where new leaves emerge and all leaves' orientations are all predetermined. This makes model simulation simple. But a stochastic model may be more appropriate for modeling a life form like a tree. In stochastic models, everything has a distribution and everything follows stochastic rules. And there is no need for a scan-conversion shading algorithm; every leaf's shading pattern would follow stochastic properties. The identification of a reliable stochastic shading model is, however, a significant challenge.

REFERENCE

1. The ECOPHYS User's Manual, By George E. Host et al. The Microcomputer Scientific Software Series 6, 1990.
2. H.M Rauscher, J.G.Isebrands, G.E.Host, R.E. Dickson, D.I. Dickmann, T.R. Crow and D.A. Michael. ECOPHYS: An Ecophysiological Growth process Model For Juvenile Poplar. *Tree Physiology*. 7: 255-281, 1990.
3. J.G. Isebrands, G.E. Host, K. McTavish and K. E. Lenz. Modeling Short-Rotation Forest Plantations Using Hierarchical, Parallel Computing: A Developmental History Of ECOPHYS, 1997.
4. G.Campbell, J. Norman, An Intro To Environmental Biophysics, 2nd Edition, Springer – Verlag, 1998.
5. Guoqiang Zang, Development Of Regulatory Components For ECOPHYS. M. S. Thesis, University of Minnesota, Duluth. 1999.
6. J. J.Lansberg, M. R. Kaufmann, D. Binkley, J. Isebrands and P.G. Jarvis. Evaluating Progress Toward Closed Forest Models Based On Fluxes Of Carbon, Water And Nutrients. *Tree Physiology* 9, 1-15. 1991.
7. Isebrands, J.G.; Nelson, N.C. Distribution of [14C]-Labeled Photosynthates Within Intensively Cultured *Populus* Clones During The Establishment Year. *Physiologia Plantarum*. 59: 9-18. 1983.
8. Guy Eddon, Henry Eddon. Inside Microsoft COM+ Base Service. Microsoft Press, 1999.
9. Don Box. Essential COM. Addison Wesley Longman, Inc. Massachusetts, 1998.

10. Grimes Richard, Alex Stockton, George Reilly, Julian Templeman. Beginning ATL COM Programming. Wrox Press Ltd, 1998.
11. David J. Kruglinski, George Shepherd and Scot Wingo. Programming Microsoft Visual C++, Fifth Edition, Microsoft Press, 1998.
12. Jeff Prorise, Programming Windows With MFC, Second Edition, Microsoft Press, 1999.
13. Bruce Eckel, Thinking In C++, Second Edition, Prantice Hall Inc, 2000.
14. Gang Wu, A Parallel Implementation of Numerical Experiments Investigating The Shading Characteristics Of *Populus Eugenei*. M. S Thesis, University of Minnesota, Duluth, 1999.
15. J.G. Isebrands, G.E. Host, K. Lenz, G. Wu and H. W. Stech, Process modeling Responses Of Forest Plantations To Interacting Multiple Stresses: A Hierarchical, Parallel Computing Strategy Using COM, For Proceedings International Workshop On Forest Ecosystem Modeling Antwerp, Belgium, 1999.
16. Chandramouli Balasubramaniam, Hierarchical Parallel and Computation Numerical Shading Experiments. M.S. Project, University of Minnesota, Duluth, 2000.
17. Wanlun Zhao, An Analytical Solution For A Mechanistic Photosynthesis Model. M. S. Thesis, University of Minnesota, Duluth, 2000.
18. H. M. Rauscher, G. E. Host, J. G. Isebrands, Modeling Light Interception Of Angiosperms With Simple Leaf Forms Using A Crown Geometric Approach. Manuscript not published.

19. K. Lenz, personal communication, 2002.
20. Martin, M. J., G. E. Host, K. E. Lenz, and J. G. Isebrands. 2001. Simulating the growth response of aspen to elevated ozone: a mechanistic approach to scaling a leaf-level model of ozone effects on photosynthesis to a complex canopy architecture. *Environmental Pollution* 115:425-436.
21. G. Host, J. Isebrands, M. Kubiske, E. McDonald, personal communication, 2002.

Appendix

1. Makefile for Linux Version

```
=====
#This is the makefile for linux version of ECOPHYS
#Written by Yongtao Guan
#References: www.gnu.org/manual/make/make.html
#          www.opussoftware.com/tutorial/TutMakefile.htm

#macros

CPP = g++
LIBS = -L/usr/X11R6/lib -lGL -lGLU -lXxf86vm -lm

HFILES = BRANCH.h common_structures.h  Edge.h File_io.hpp \
        Geom.hpp  INTERNODE.h Julian.h  leaf.hpp \
        LeafScanConv.h MyShade.h  parameters.hpp root.hpp \
        Simulate.hpp  SITE.h TransCoef.h  TRANSPORT.h \
        Tree.hpp  WEATHDAY.h  WIMOVAC.h glView.h

OFILES = main.o  BRANCH.o  Edge.o  File_io.o \
        Geom.o  INTERNODE.o  Julian.o  Leaf.o \
        LeafScanConv.o MyShade.o  parameters.o \
        Simulate.o  SITE.o  TransCoef.o TRANSPORT.o \
        Tree.o WEATHDAY.o  WIMOVAC.o glView.o

CFILES = main.cpp  BRANCH.cpp Edge.cpp  File_io.cpp \
        Geom.cpp  INTERNODE.cpp Julian.cpp  Leaf.cpp \
        LeafScanConv.cpp MyShade.cpp parameters.cpp Root.cpp \
        Simulate.cpp  SITE.cpp  TransCoef.cpp TRANSPORT.cpp \
        Tree.cpp  WEATHDAY.cpp  WIMOVAC.cpp  glView.cpp

ecolin: $(OFILES)
        $(CPP) -ggdb -o ecolin $(OFILES) $(LIBS)

$(OFILES): $(HFILES) Makefile

all:  ecolin

clean:
        rm -f $(OFILES) ecolin
=====
```

2. Downward Transportation in the Old Code

In Wanlun Zhao's version [17], in the class TransCoef there is a function named `calc_dc()`, which is called by `Transport::getTC()`. In the new code this function has been changed to `SetDownTC()`, a member function of class TRANSPORTATION. It fills the following lower part of the internode transformation matrix ITC, replacing zeros in the shaded region with non-negative numbers. The new entries are the same within columns, but may change from day to day.

TransITC	LPI<5	5	6	7	8	9	10	11
0	0	0.01	0.03	0.02	0.02	0.02	0.02	0.018333
1	0	0.01	0.02	0.02	0.02	0.02	0.02	0.018333
2	0	0.01	0.02	0.02	0.02	0.02	0.01	0.018333
3	0	0.03	0.03	0.02	0.02	0.02	0.02	0.018333
4	0	0.05	0.03	0.02	0.03	0.01	0.01	0.018333
5	0	0.14	0.04	0.03	0.03	0.01	0.02	0.018333
6	0	0	0.1	0.05	0.03	0.01	0.01	0.018333
7	0	0	0	0.05	0.03	0.01	0.02	0.018333
8	0	0	0	0	0.04	0.03	0.01	0.018333
9	0	0	0	0	0	0.02	0.02	0.018333
10	0	0	0	0	0	0	0.01	0.018333
11	0	0	0	0	0	0	0	0.018333
12	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
up.in[i][j]	from j to l	0.25	0.27	0.23	0.24	0.17	0.17	0.219996

This evenly distributes the portion of photosynthesis that the leaf passes downward to internodes to the nearest lower available internodes (at most 10)

5	6	7	8	9	10	11	12	13	14	15	LPI
0.26	0.33	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.35	Internode

TransDTC

3. Code for DCBA With Quick Sort

Part 1 main procedure

```
bool MyShade::DoShading(int NumLeaf, LEAF** LeafPtrArray, double aze, double za)
{
    int i, j, k;
    LEAF * tempL;
    LEAF * A;
    double M1[3][3] = {      {cos(-aze), -sin(-aze), 0},
                             {sin(-aze), cos(-aze), 0},
                             {0,      0,      1}      };

    double M2[3][3] = {      {cos(-za), 0, sin(-za)},
                             {0,      1,      0},
                             {-sin(-za),0,cos(-za)}  };

    double M[3][3] = {{0,0,0},{0,0,0},{0,0,0}};

    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            for(k=0;k<3;k++) M[i][j] += M2[i][k] * M1[k][j];

    for(j = 0; j < NumLeaf; j++)
    {
        A = LeafPtrArray[j];
        for (int i = 0; i < 4; i++)
        {
            A->SCVertices[i].x = M[0][0] * A->p[i].x + M[0][1] * A->p[i].y
                               + M[0][2] * A->p[i].z;
            A->SCVertices[i].x *= ScanDensity;
            A->SCVertices[i].y = M[1][0] * A->p[i].x + M[1][1] * A->p[i].y
                               + M[1][2] * A->p[i].z;
            A->SCVertices[i].y *= ScanDensity;
            A->SCVertices[i].z = M[2][0] * A->p[i].x + M[2][1] * A->p[i].y
                               + M[2][2] * A->p[i].z;
            A->SCVertices[i].z *= ScanDensity;
        }

        A->SCLCP.z = (A->SCVertices[1].z + A->SCVertices[3].z) / 2.0;

        A->xleft = mymin(mymin(A->SCVertices[0].x, A->SCVertices[1].x),
                       mymin(A->SCVertices[2].x, A->SCVertices[3].x));
        if (A->xleft < xLeft) xLeft = A->xleft;
        A->xright = mymax(mymax(A->SCVertices[0].x, A->SCVertices[1].x),
                        mymax(A->SCVertices[2].x, A->SCVertices[3].x));
        if (A->xright > xRight) xRight = A->xright;
        A->ytop = mymax(mymax(A->SCVertices[0].y, A->SCVertices[1].y),
                      mymax(A->SCVertices[2].y, A->SCVertices[3].y));
        if (A->ytop > yTop) yTop = A->ytop;
        A->ybottom = mymin(mymin(A->SCVertices[0].y, A->SCVertices[1].y),
                          mymin(A->SCVertices[2].y, A->SCVertices[3].y));
        if (A->ybottom < yBottom) yBottom = A->ybottom;
    }
}
// above for loop is changing coordinate system and finding out the canvas range;
```

```

qsort(LeafPtrArray, NumLeaf, sizeof(LEAF *), compare); //quick sort

PxIRow = yTop - yBottom + 1;
PxICol = xRight - xLeft + 1;
PxIMatrix = new int[PxIRow * PxICol];

for(i = 0; i < PxIRow * PxICol; i++)
    PxIMatrix[i] = -1;

for(i = 0; i < NumLeaf; i++)
    SetLeafPxI(LeafPtrArray[i], i);

for(i = 0; i < PxIRow; i++)
    for(j = 0; j < PxICol; j++)
        if (PxIMatrix[i * PxICol + j] > -1)
            (LeafPtrArray[PxIMatrix[i * PxICol + j]]->UnShadedPxI)++;

delete []PxIMatrix;
PxIMatrix = NULL;

for(i = 0; i < NumLeaf; i++)
{
    if(LeafPtrArray[i]->TotalPxI !=0)
        LeafPtrArray[i]->sunArea = LeafPtrArray[i]->l_area * (double)
LeafPtrArray[i]->UnShadedPxI / (double) LeafPtrArray[i]->TotalPxI;
    else LeafPtrArray[i]->sunArea = 0;
    LeafPtrArray[i]->shadeArea = LeafPtrArray[i]->l_area - LeafPtrArray[i]->sunArea;
    LeafPtrArray[i]->TotalPxI = 0;
    LeafPtrArray[i]->UnShadedPxI = 0;
}

return true;
}

//compare function for quick sort
inline int compare(const void * A, const void * B)
{
    LEAF * tempA = *(LEAF **) A;
    LEAF * tempB = *(LEAF **) B;
    if (tempA->LCP.z > tempB->LCP.z)
        return 1;
    else if (tempA->LCP.z < tempB->LCP.z)
        return -1;
    else return 0;
}

```

Part 2 Scan Conversion algorithm

```

bool MyShade::SetLeafPxl(LEAF* A, int order) //scan conversion algorithm
{
    register i, j, k;
    Edge *e[4], *etemp;
    e[0] = new Edge(A->SCVertices[0].x, A->SCVertices[0].y,
        A->SCVertices[1].x,A->SCVertices[1].y);
    e[1] = new Edge(A->SCVertices[1].x, A->SCVertices[1].y,
        A->SCVertices[2].x,A->SCVertices[2].y);
    e[2] = new Edge(A->SCVertices[2].x, A->SCVertices[2].y,
        A->SCVertices[3].x,A->SCVertices[3].y);
    e[3] = new Edge(A->SCVertices[3].x, A->SCVertices[3].y,
        A->SCVertices[0].x,A->SCVertices[0].y);

    Edge *e1, *e2;
    for(i=0;i<4;i++)
        for(j=i+1;j<4;j++)
            if(e[i]->ymax < e[j]->ymax)
                {
                    etemp = e[i];
                    e[i] = e[j];
                    e[j] = etemp;
                }

    k=2;
    Edge* actEdge[2]; //at any time, we have only two active edges.
    actEdge[0] = e[0];
    actEdge[1] = e[1];
    for(i = e[0]->ymax; i > e[3]->ymin+1; i--)
    {
        if(i==actEdge[0]->ymin) {actEdge[0] = e[k++]; break;}
        if(i==actEdge[1]->ymin) {actEdge[1] = e[k++]; break;}
        int temp1=actEdge[0]->EdgeScan();
        int temp2=actEdge[1]->EdgeScan();
        for(j = __min(temp1, temp2); j < __max(temp1, temp2); j++)
        {
            PxlMatrix[(i-yBottom) * PxlCol + j-xLeft] = order;
            (A->TotalPxl)++;
        }
    } //take advantage that we have a simple convex leaf

    for(i=0;i<4;i++) delete e[i];

    return true;
}

```

4. Code for Bit-DCBA With Quick Sort

Part 1. main procedure.

```

bool MyShadeBit::DoShading(int NumLeaf, LEAF** LeafPtrArray, float aze, float za)
{
    register i, j, k;
    LEAF * tempL;
    LEAF * A;
    float M1[3][3] = {          {cos(-aze), -sin(-aze), 0},
                              {sin(-aze), cos(-aze), 0},
                              {0,      0,      1}      };

    float M2[3][3] = {          {cos(-za),      0,      sin(-za)},
                              {0,              1,              0},
                              {-sin(-za),      0,      cos(-za)}
                              };

    float M[3][3] = {{0,0,0},{0,0,0},{0,0,0}};

    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            for(k=0;k<3;k++) M[i][j] += M2[i][k] * M1[k][j];

    for(j = 0; j < NumLeaf; j++)
    {
        A = LeafPtrArray[j];
        for (int i = 0; i < 4; i++)
        {
            A->SCVertices[i].x = M[0][0] * A->p[i].x + M[0][1] * A->p[i].y
                               + M[0][2] * A->p[i].z;
            A->SCVertices[i].x *= ScanDensity;
            A->SCVertices[i].y = M[1][0] * A->p[i].x + M[1][1] * A->p[i].y
                               + M[1][2] * A->p[i].z;
            A->SCVertices[i].y *= ScanDensity;
            A->SCVertices[i].z = M[2][0] * A->p[i].x + M[2][1] * A->p[i].y
                               + M[2][2] * A->p[i].z;
            A->SCVertices[i].z *= ScanDensity;
        }

        A->SCLCP.z = (A->SCVertices[1].z + A->SCVertices[3].z) / 2.0;
        A->xleft = __min(__min(A->SCVertices[0].x, A->SCVertices[1].x),
                       __min(A->SCVertices[2].x, A->SCVertices[3].x));
        if (A->xleft < xLeft) xLeft = A->xleft;
        A->xright = __max(__max(A->SCVertices[0].x, A->SCVertices[1].x),
                        __max(A->SCVertices[2].x, A->SCVertices[3].x));
        if (A->xright > xRight) xRight = A->xright;
        A->ytop = __max(__max(A->SCVertices[0].y, A->SCVertices[1].y),
                      __max(A->SCVertices[2].y, A->SCVertices[3].y));
        if (A->ytop > yTop) yTop = A->ytop;
        A->ybottom = __min(__min(A->SCVertices[0].y, A->SCVertices[1].y),
                          __min(A->SCVertices[2].y, A->SCVertices[3].y));
        if (A->ybottom < yBottom) yBottom = A->ybottom;
    }

    // above for loop is changing coordinate system and finding the range of canopy;
}

```

```

qsort(LeafPtrArray, NumLeaf, sizeof(LEAF *), compare); //quick sort

PxlRow = yTop - yBottom + 1;
PxlCol = xRight - xLeft + 1;

int actual = (int) (PxlRow * PxlCol * 1.0 / 8.0) + 1; // we use unsigned char to denote 8 bits.
PxlMatrix = new unsigned char[actual];
for (i = 0; i < actual; i++)
    PxlMatrix[i] = 0;

for(i = 0; i < NumLeaf; i++)
    SetLeafPxl(LeafPtrArray[i], i);

delete []PxlMatrix;
PxlMatrix = NULL;

return true;
}

//compare function used in quick sort.
inline int compare(const void * A, const void * B)
{
    LEAF * tempA = *(LEAF **) A;
    LEAF * tempB = *(LEAF **) B;
    if (tempA->LCP.z > tempB->LCP.z)
        return -1;
    else if (tempA->LCP.z < tempB->LCP.z)
        return 1;
    else return 0;
}

```

Part 2. Scan conversion algorithm

```

bool MyShadeBit::SetLeafPxl(LEAF * A, int order)
{
    register i, j, k;
    Edge *e[4], *etemp;
    e[0] = new Edge(A->SCVertices[0].x, A->SCVertices[0].y,
                   A->SCVertices[1].x, A->SCVertices[1].y);
    e[1] = new Edge(A->SCVertices[1].x, A->SCVertices[1].y,
                   A->SCVertices[2].x, A->SCVertices[2].y);
    e[2] = new Edge(A->SCVertices[2].x, A->SCVertices[2].y,
                   A->SCVertices[3].x, A->SCVertices[3].y);
    e[3] = new Edge(A->SCVertices[3].x, A->SCVertices[3].y,
                   A->SCVertices[0].x, A->SCVertices[0].y);

    Edge *e1, *e2;
    for(i=0; i<4; i++)
        for(j=i+1; j<4; j++)
            if(e[i]->ymax < e[j]->ymax)
                {
                    etemp = e[i]; e[i] = e[j]; e[j] = etemp;
                }

    k=2;
    Edge* actEdge[2]; //any time, we have only two active edges.
    actEdge[0] = e[0];
    actEdge[1] = e[1];
    unsigned char mask[8] = { 128, 64, 32, 16, 8, 4, 2, 1 }; //from left to right;
    for(i = e[0]->ymax; i > e[3]->ymin+1; i--) //take advantage that we have a simple convex leaf
    {
        if(i==actEdge[0]->ymin) {actEdge[0] = e[k++]; break;}
        if(i==actEdge[1]->ymin) {actEdge[1] = e[k++]; break;}
        int temp1=actEdge[0]->EdgeScan();
        int temp2=actEdge[1]->EdgeScan();
        int xmin = __min(temp1, temp2);
        int xmax = __max(temp1, temp2);
        for(j = xmin; j < xmax; j++)
        {
            int temp = (i-yBottom) * PxlCol + j-xLeft;
            div_t location = div(temp, 8);
            if(int (PxlMatrix[location.quot] & mask[location.rem]) == 0)
                {
                    PxlMatrix[location.quot] = PxlMatrix[location.quot] | mask[location.rem];
                    A->UnShadedPxl++;
                }
        }
        A->TotalPxl += (xmax - xmin);
    }

    if(A->TotalPxl !=0)
        A->sunArea = A->l_area * (float) A->UnShadedPxl / (float) A->TotalPxl;
    else A->sunArea = 0;
    A->shadeArea = A->l_area - A->sunArea;
    A->TotalPxl = 0;
    A->UnShadedPxl = 0;
    for(i=0; i<4; i++) delete e[i];
    return true;
}

```

5. Constant allocation code

```
BRANCH *tempBrn1; //pointer to different branches
INTERNODE *tempI1; //pointer to different internodes
templength = 0; //total length of active internode chain
tempBrn = pBrn->CBrnParentBrn(); //find the mother branch of current branch attached to
if(tempBrn)
{
    tempI = pBrn->CBrnGetParentIntnd(); //find the internode that current branch on
    while (tempBrn && tempI)
    {
        templength += tempBrn->CBrnActiveLength(tempI);
        tempI = tempBrn->CBrnGetParentIntnd();
        tempBrn = tempBrn->CBrnParentBrn();
    }
}
//above calculate the total length of the active internode chain.
tI = pBrn->m_pBsIntnd;
while (tI)
{
    if (! tI->L || tI->L->DownToIntnd <=0.0)
    {
        templength += tI->i_length;
        tI = tI->m_pNextIntnd;
        continue;
    }
    else
    {
        templength += tI->i_length;
        tempamt = tI->L->DownToIntnd / templength; // "constant"
        INTERNODE * pItemp = tI;
        while(pItemp)
        {
            pItemp->i_psyn += tempamt * pItemp->i_length;
            pItemp = pItemp->back;
        }
        tempBrn1 = pBrn->CBrnParentBrn();
        tempI1 = NULL;
        if (tempBrn1)
        {
            tempI1 = pBrn->CBrnGetParentIntnd();
            while (tempBrn1 && tempI1)
            {
                tempBrn1->CBrnPushDown(tempI1, tempamt);
                tempI1 = tempBrn1->CBrnGetParentIntnd();
                tempBrn1 = tempBrn1->CBrnParentBrn();
            }
        }
        tI = tI->m_pNextIntnd;
    }
} //allocating the carbon to internode chain.
```